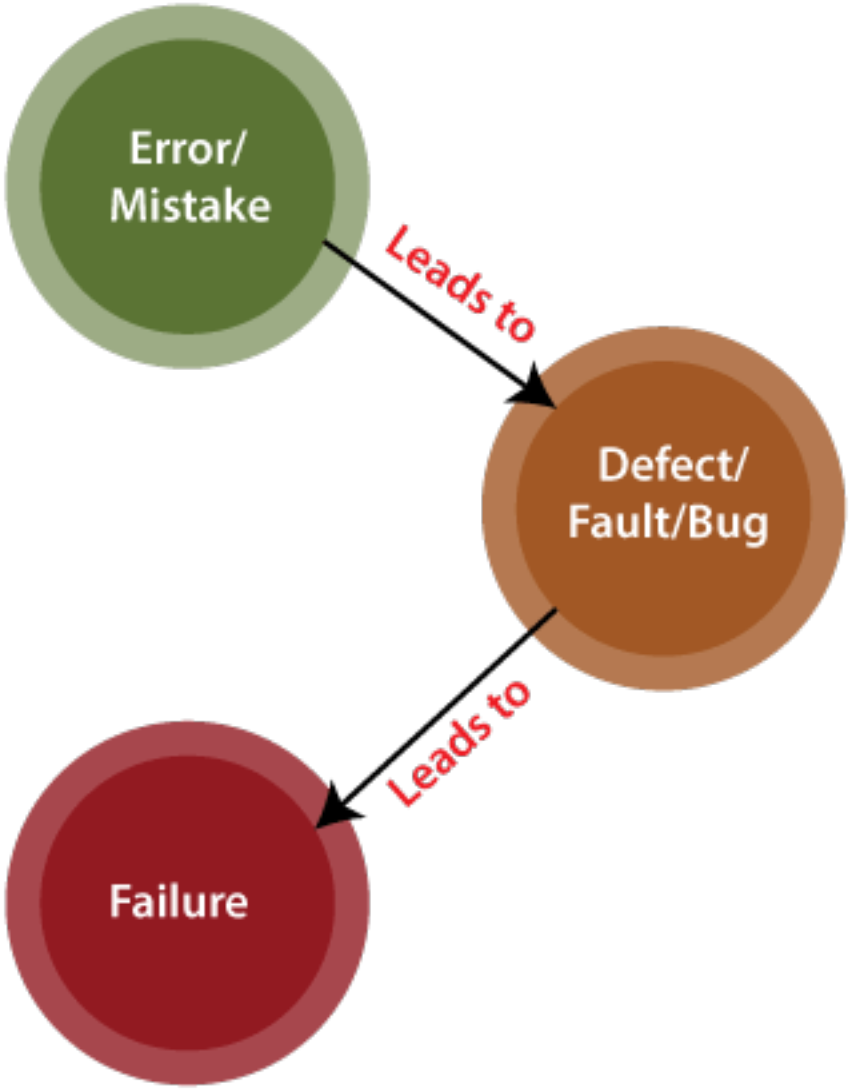
Software Testing

Reference Book:
Software Engineering, Ian Sommerville
10th / Global Edition, Chapter 8
Pearson Publishers

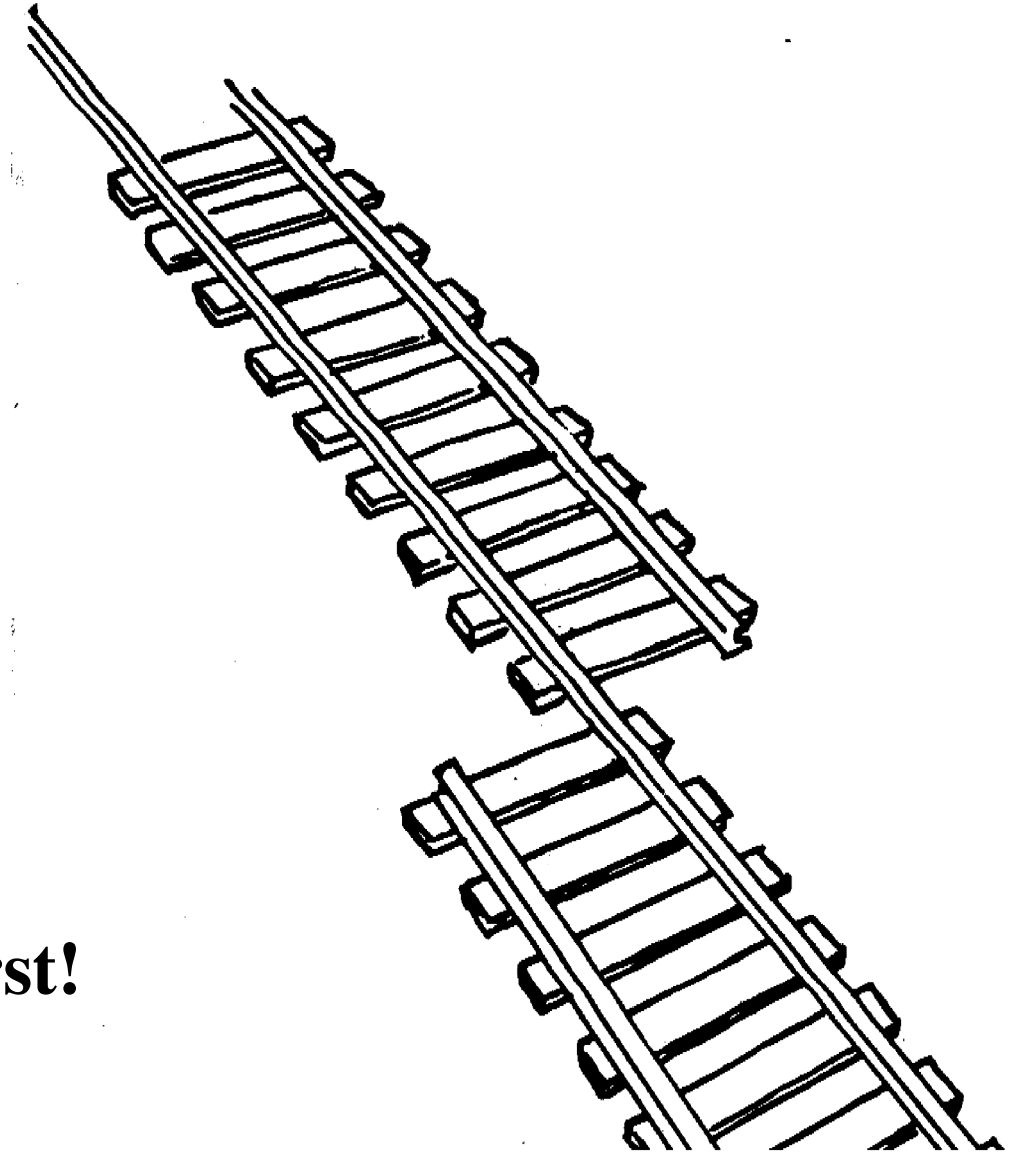
Software Error, Fault and Failure

- **Error/Mistake:** An error is a mistake made in the code, that's why we cannot execute of compile code.
- **Fault/Defect/Bug:** The fault is a state that causes the software to fail to accomplished its essential function.
- **Failure:** Software failure is a loss specifies a fatal issue in software or in its module, which makes the system unresponsive or broken.



Software Error, Fault and Failure

What is this?



A failure?

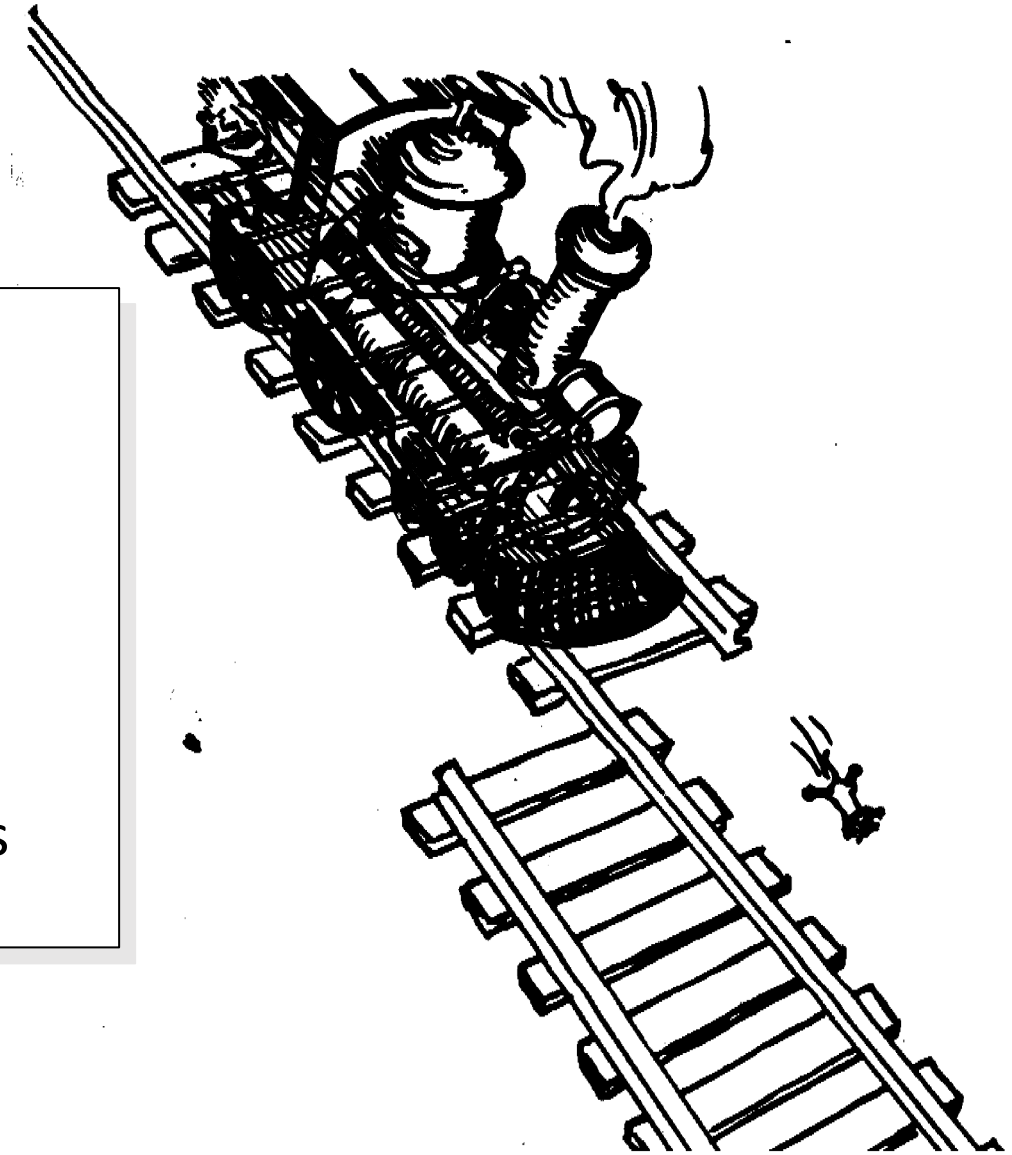
An error?

A fault?

Need to specify
the desired behavior first!

Erroneous State ("Error")

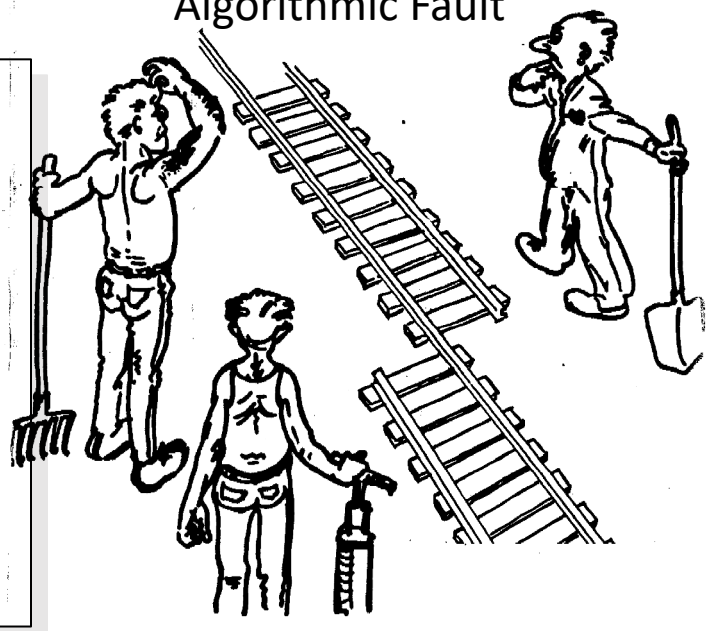
- Errors
  - Stress or overload errors
  - Capacity or boundary errors
  - Timing errors
  - Throughput or performance errors



Examples of Faults

- Faults in the Interface specification
  - Mismatch between what the client needs and what the server offers
  - Mismatch between requirements and implementation

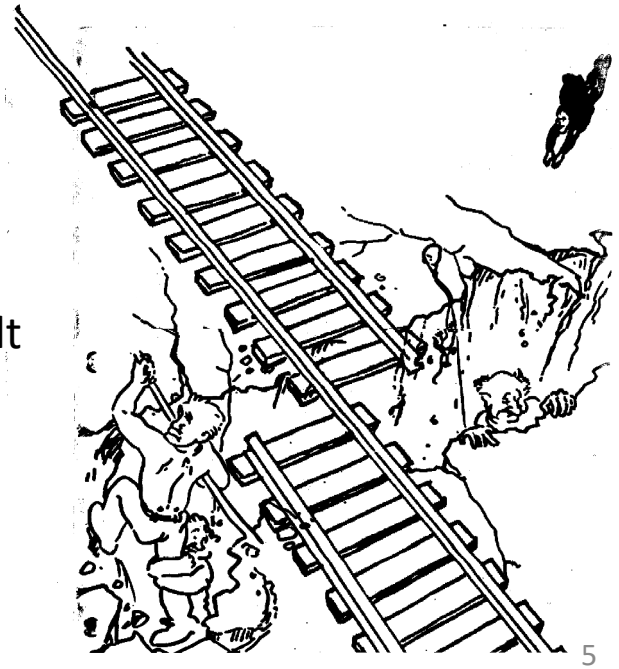
Algorithmic Fault



- Mechanical Faults (very hard to find)
  - Documentation does not match actual conditions or operating procedures

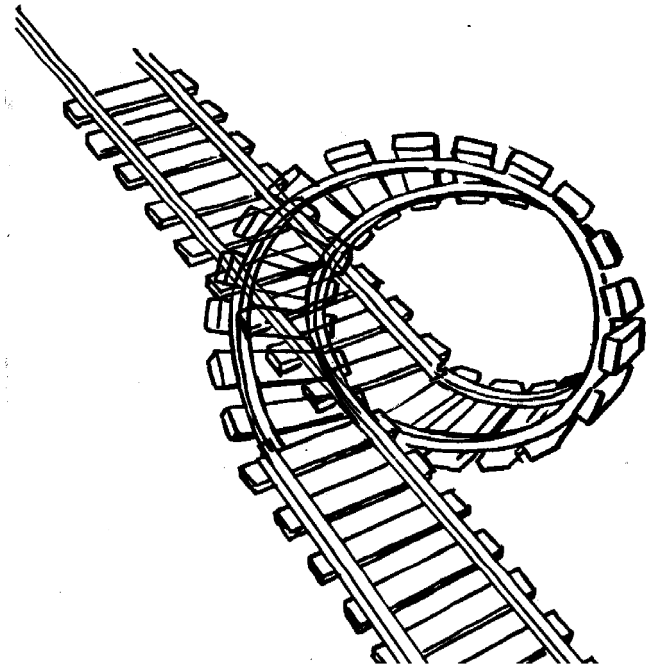
- Algorithmic Faults
  - Missing initialization
  - Branching errors (too soon, too late)
  - Missing test for nil

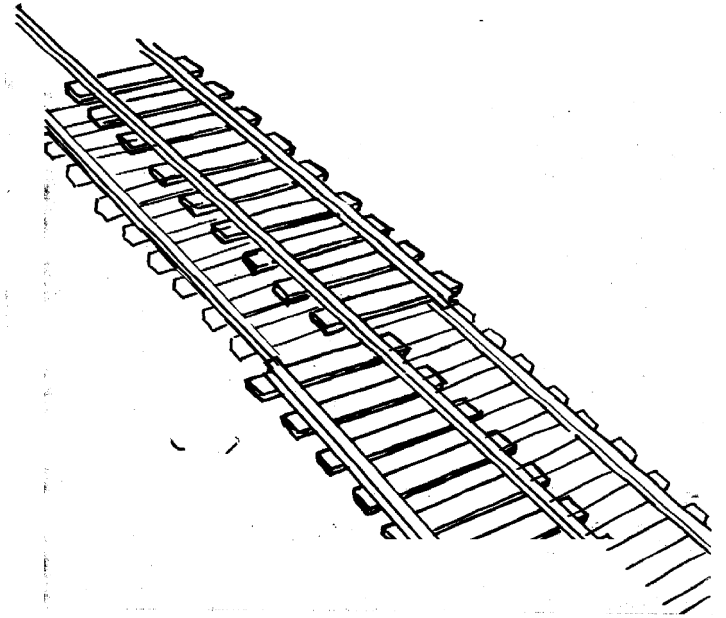
Mechanical Fault



Dealing with Errors

- **Verification:**
  - Assumes hypothetical environment that does not match real environment
  - Proof might be buggy (omits important constraints; simply wrong)





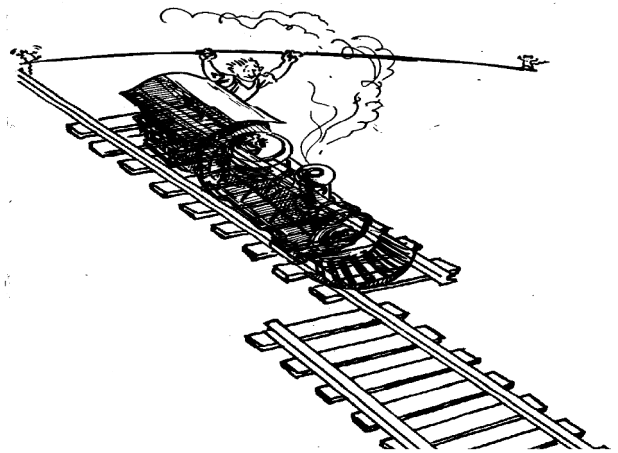
Modular redundancy:

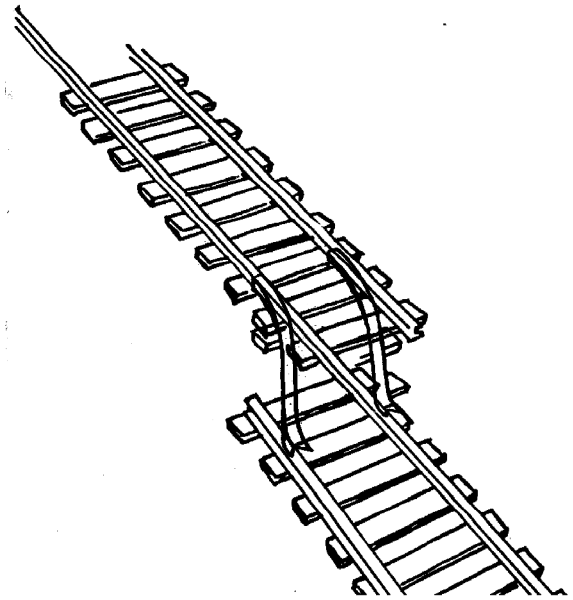
Expensive

Dealing with Errors

Declaring a bug to be a “feature”

Bad practice



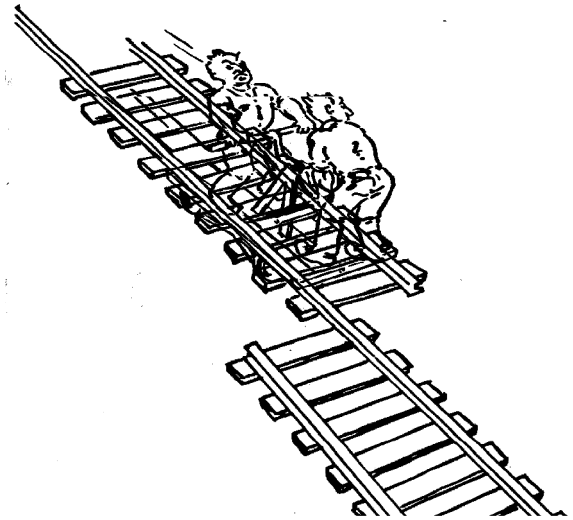


Patching

Slows down performance

Testing (this lecture)

Testing is never good enough



Another View on How to Deal with Errors

Error prevention (before the system is released):

- ✓Use good programming methodology to reduce complexity
- ✓Use version control to prevent inconsistent system
- ✓Apply verification to prevent algorithmic bugs

Error detection (while system is running):

- ✓ Testing: Create failures in a planned way
- ✓ Debugging: Start with an unplanned failures
- ✓ Monitoring: Deliver information about state. Find performance bugs

Error recovery (recover from failure once the system is released):

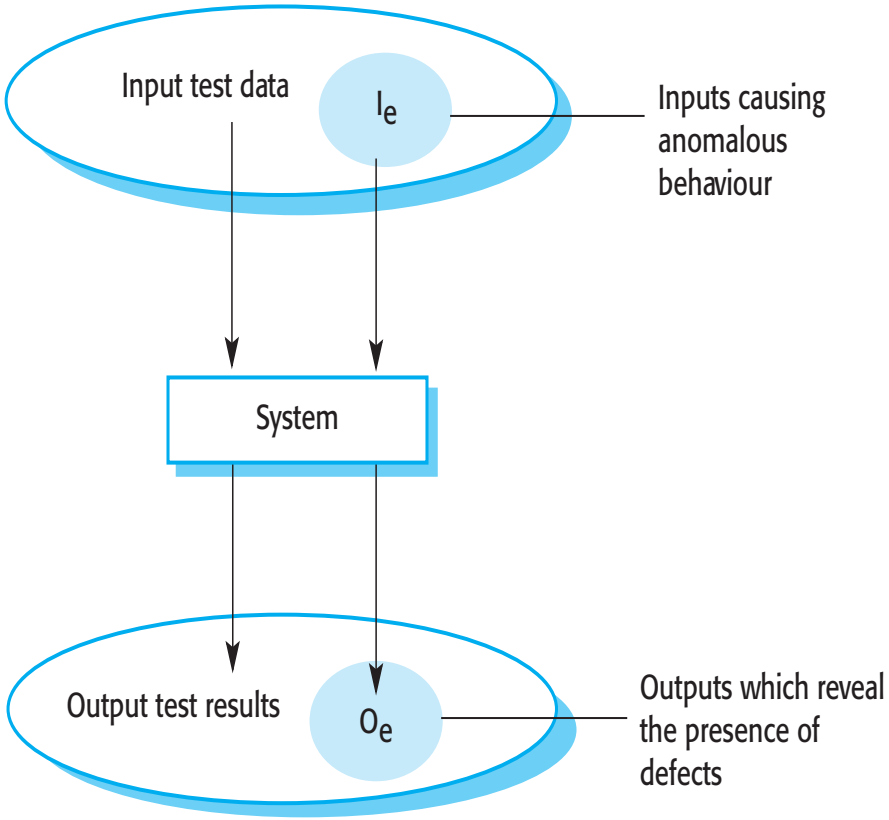
- ✓ Data base systems (atomic transactions)
- ✓ Modular redundancy
- ✓ Recovery blocks

Testing Takes Creativity

- Testing often viewed as dirty work.
- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Knowledge of the testing techniques
  - Skill to apply these techniques in an effective and efficient manner
- Testing is done best by independent testers
  - We often develop a certain mental attitude that the program should in a certain way when in fact it does not.
- Programmer often stick to the data set that makes the program work
  - "Don’t mess up my code!"
- A program often does not work when tried by somebody else.
  - Don't let this be the end-user.

Basics of Software Testing

"Testing can only show the presence of errors, not their absence" – Dijkstra, 1972



Purpose of Software Testing:

- **Validation Testing**: To demonstrate to the developer and the customer that the software meets its requirements.
  - ✓ For custom software, at least one test for every requirement in the requirements document.
  - ✓ For generic software products, test for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- **Defect Testing**: To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - ✓ Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

Basics of Software Testing

Verification and Validation (V & V)

❖ **Verification**:

- ✓ Building the product right
- ✓ The software conforms to its specifications.

❖ **Validation**:

- ✓ Building the right product.
- ✓ The software does what the user really requires.

Aim of V & V is to establish confidence that the system is 'fit for purpose'.

User Expectations and Market Environment:

➢ **Software Purpose**:

- ✓ The level of Confidence depends on how critical the software is to an organization.

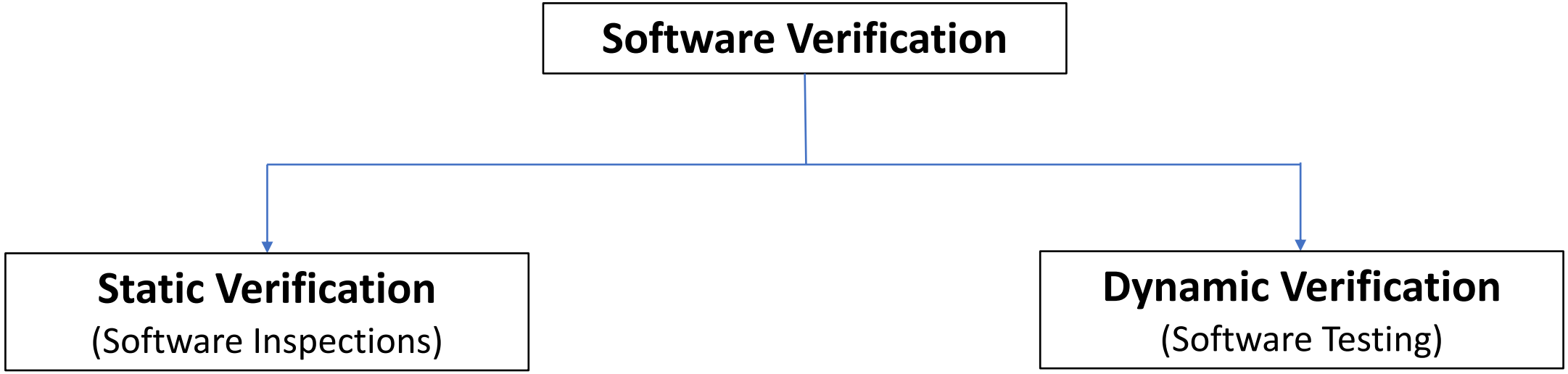
➢ **User Expectations**:

- ✓ Users may have low expectations of certain kinds of software.

➢ **Market Environment**:

- ✓ Getting a product to market early may be more important than finding defects in the program.

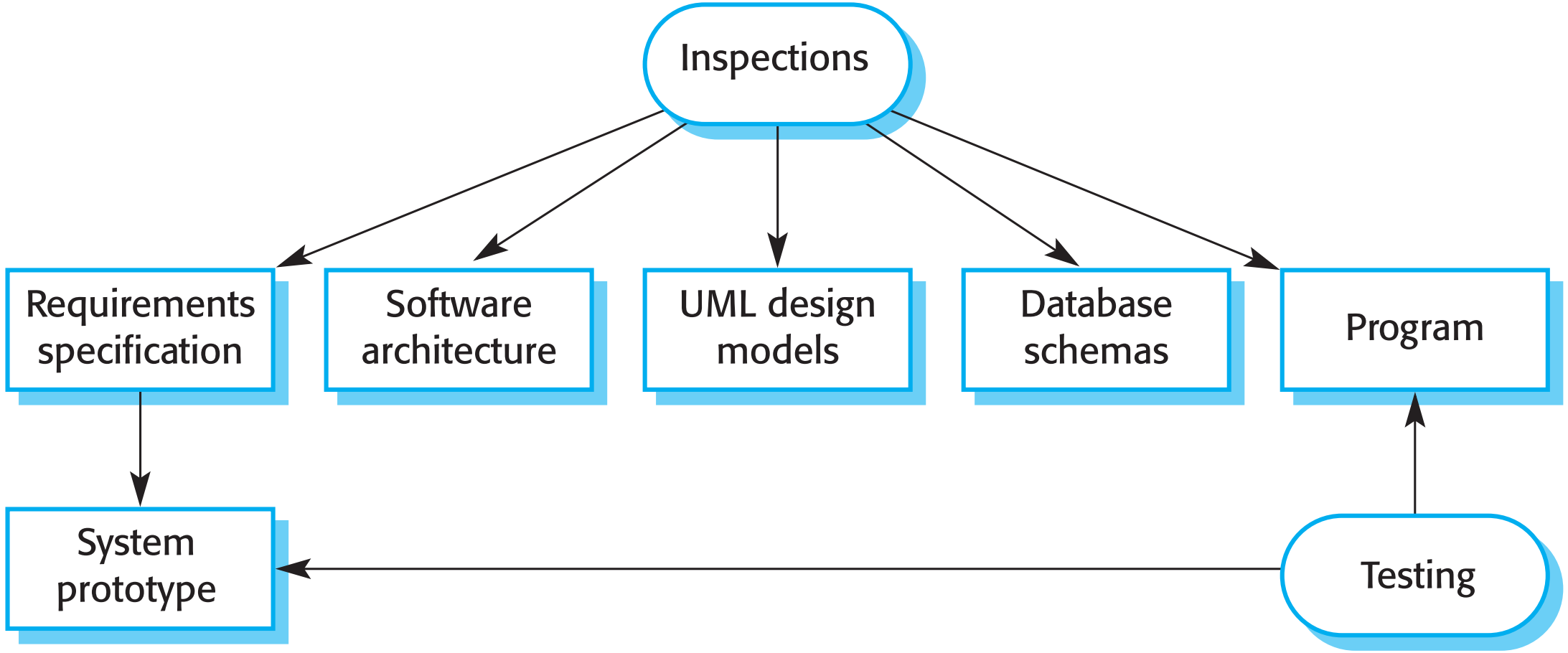
Basics of Software Testing



- ✓ Analysis of the system representation to discover problems.
- ✓ May be supplement by tool-based document and code analysis.

- ✓ Concerned with exercising and observing product behaviour.
- ✓ The system is executed with test data and its operational behaviour is observed.

Software Inspections and Testing



Software Inspections

- Examining the source and system representation (requirements, design, configuration data, test data, etc.) with the aim of discovering anomalies and defects.
- Doesn't require execution of a system, so may be used before implementation.
- An effective technique for discovering program errors.

Software Inspections

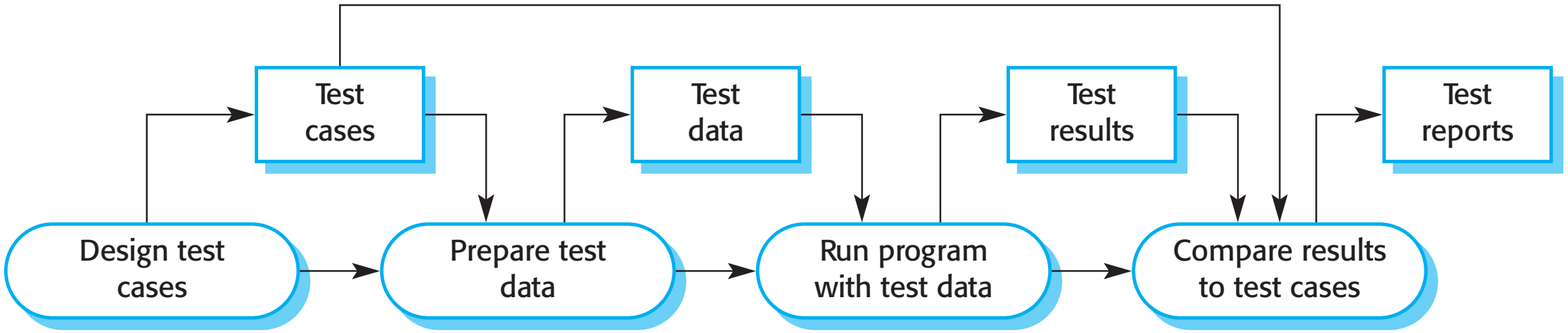
Advantages:

- Because of being a static process, don't have to be concerned with interactions between errors.
  - ✓ During testing, errors can mask (hide) other errors.
- Incomplete versions of a system can be inspected without additional costs.
  - ✓ If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- Besides searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

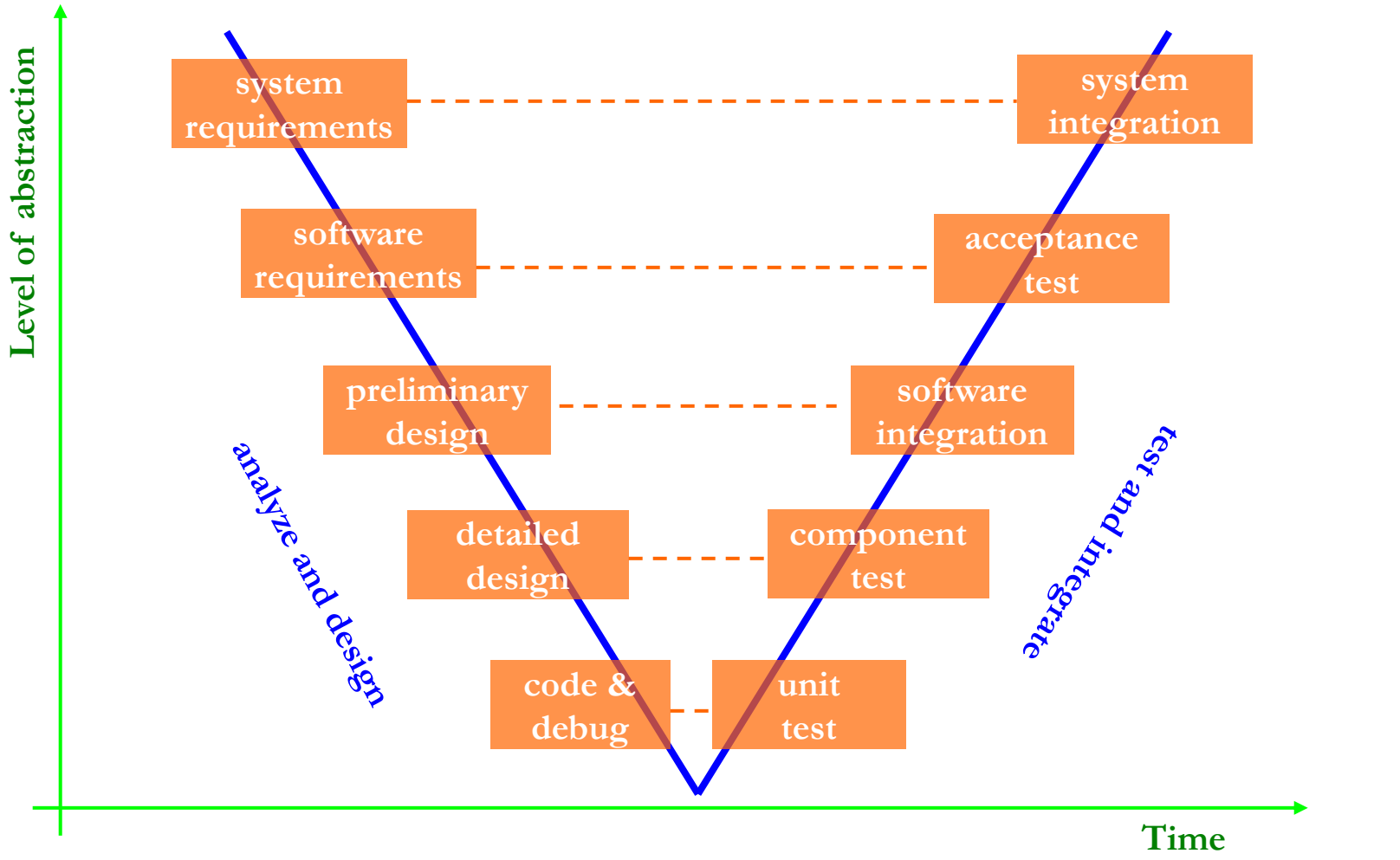
Limitations:

- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

A Model for Software Testing Process

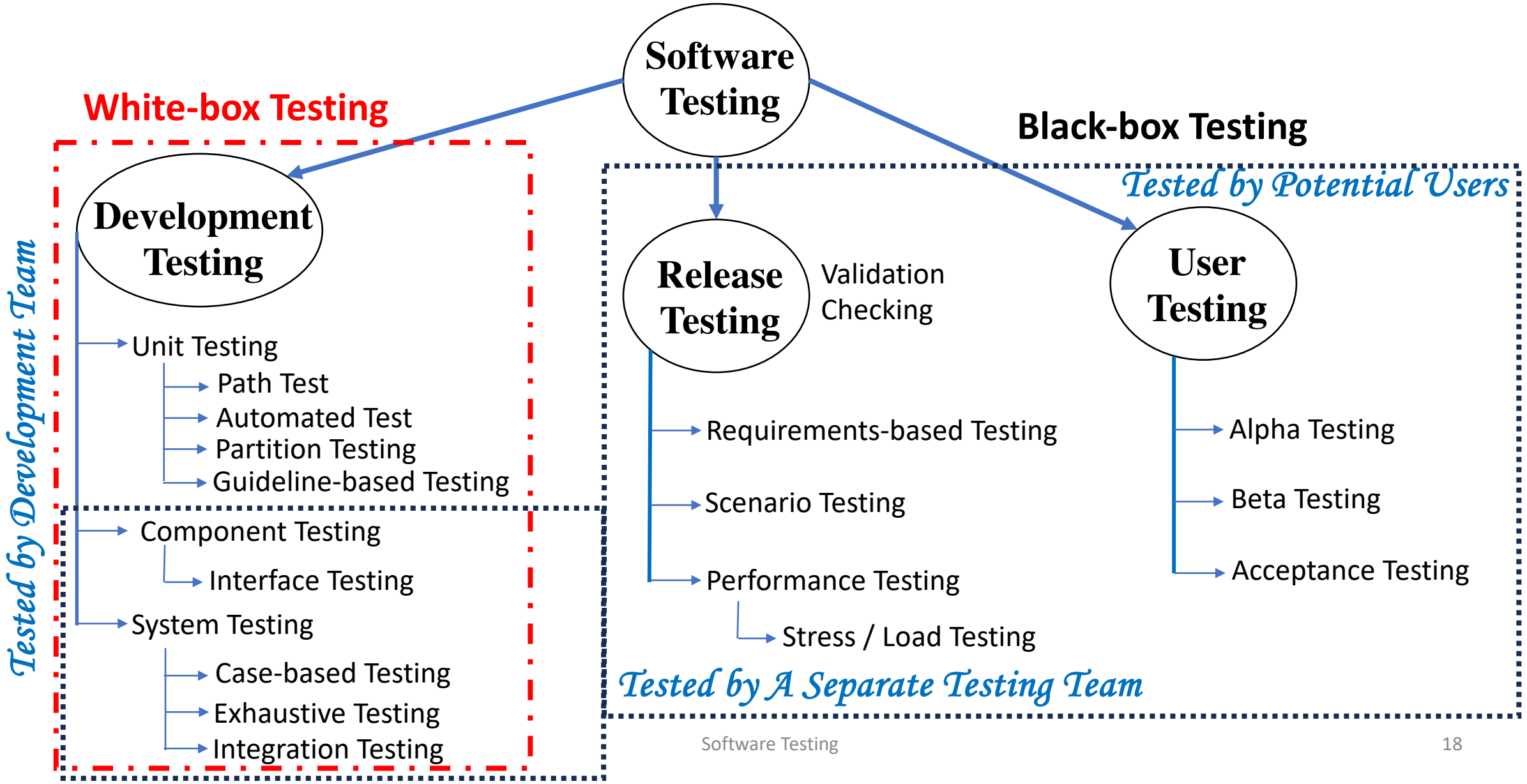


Levels of Testing in V Model

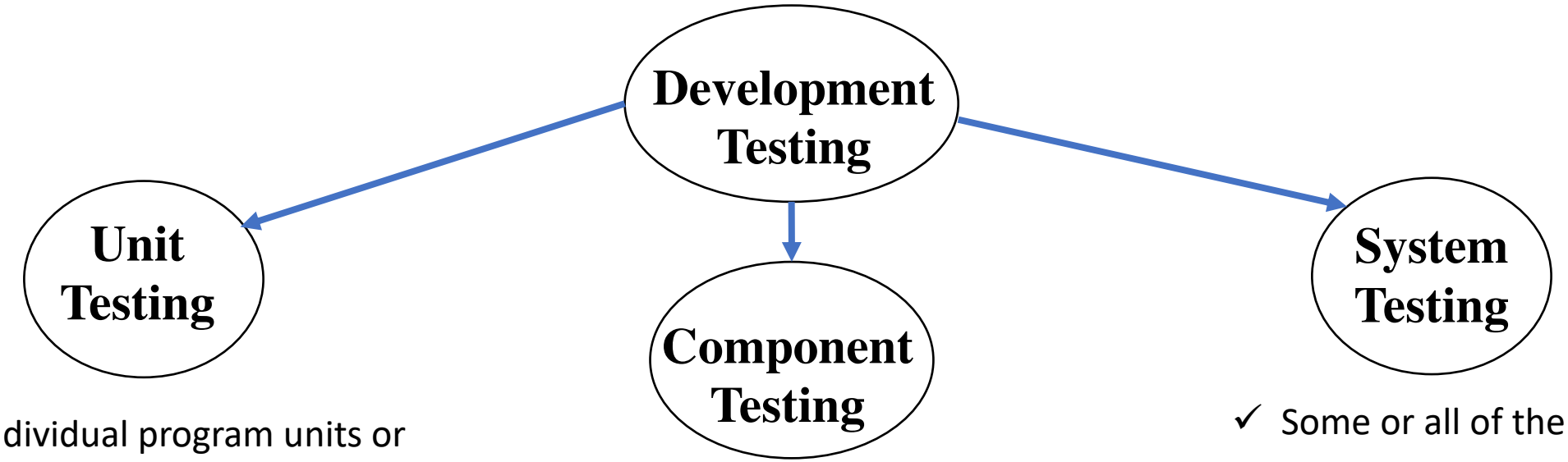


N.B.: component test vs. unit test; acceptance test vs. system integration

Different Types of Software Testing



Development Testing



- ✓ Individual program units or object classes are tested.
- ✓ Unit testing should focus on testing the functionality of objects or methods.

- ✓ Several individual units are integrated to create composite components.
- ✓ Component testing should focus on testing component interfaces.

- ✓ Some or all of the components in a system are integrated and the system is tested as a whole.
- ✓ System testing should focus on testing component interactions.

Unit Testing

Unit Testing

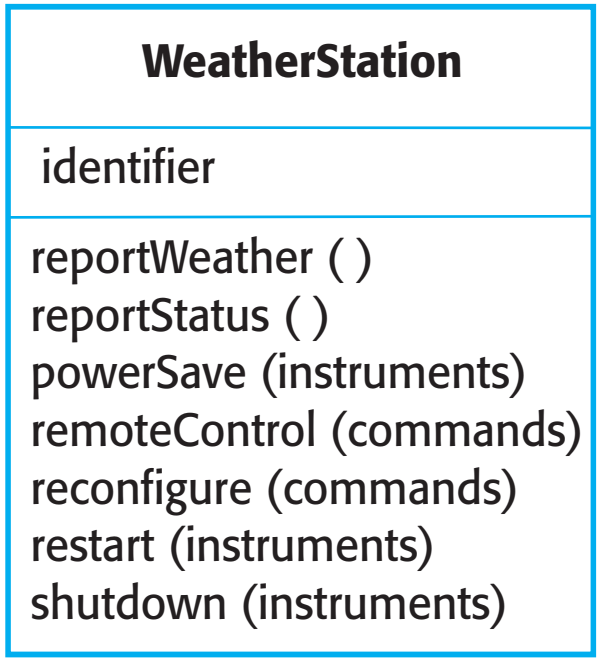
- Unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- Units may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components with defined interfaces used to access their functionality.

- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states.
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

An Example of Unit Testing

Weather Station Testing

- Need to define test cases for reportWeather, calibrate, test, startup and shutdown.
- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- For example:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing -> Transmitting -> Running
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running



Choosing Unit Test Cases

Two types of unit test case:

- The first of these should reflect normal operation of a program and should show that the component works as expected.
- The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

Path Testing

- To exercise every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once.
- All conditional statements are tested for both true and false cases.
- In an object-oriented development process, path testing may be used to test the methods associated with objects.

Automated Testing

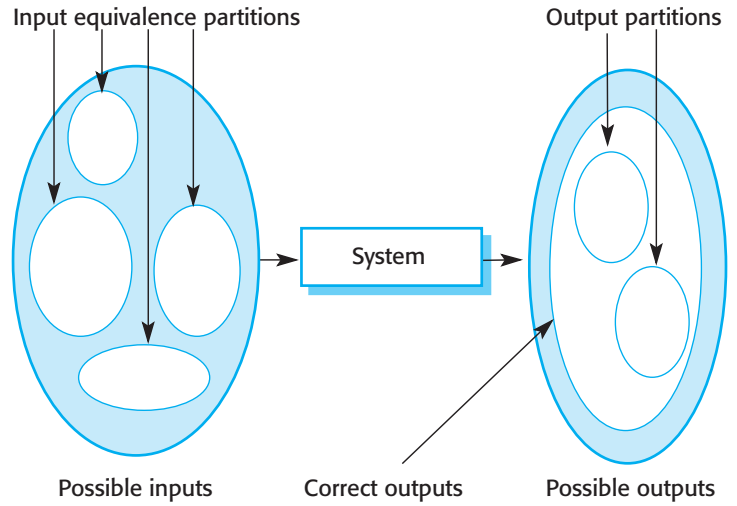
- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- A test automation framework (such as JMeter) is used to write and run program tests.
- Unit testing frameworks provide generic test classes that can be extended to create specific test cases.

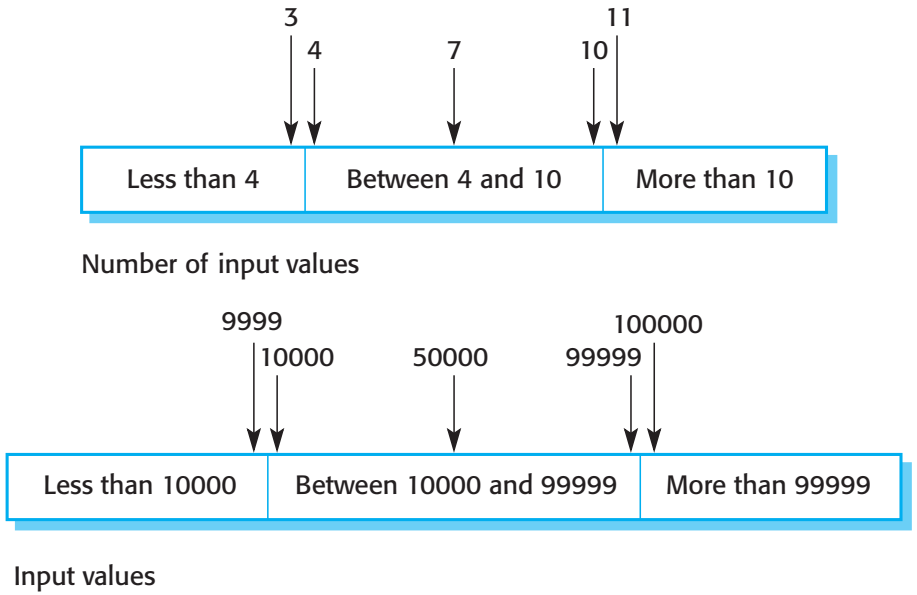
Automated Test Components:

- **A setup part,** where you initialize the system with the test case, namely the inputs and expected outputs.
- **A call part,** where you call the object or method to be tested.
- **An assertion part,** where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

Equivalence Partition Testing

- Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
  - ✓ You should choose tests from within each of these groups.
- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.





Guideline-based Testing

Guideline-based testing, where you use testing guidelines to choose test cases.

- ✓ These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Testing Guidelines for Sequences:

- ➢ Test software with sequences which have only a single value.
- ➢ Use sequences of different sizes in different tests.
- ➢ Derive tests so that the first, middle and last elements of the sequence are accessed.
- ➢ Test with sequences of zero length.

Guideline-based Testing

Some General Testing Guidelines:

- ✓Choose inputs that force the system to generate all error messages
- ✓Design inputs that cause input buffers to overflow
- ✓Repeat the same input or series of inputs numerous times
- ✓Force invalid outputs to be generated
- ✓Force computation results to be too large or too small.

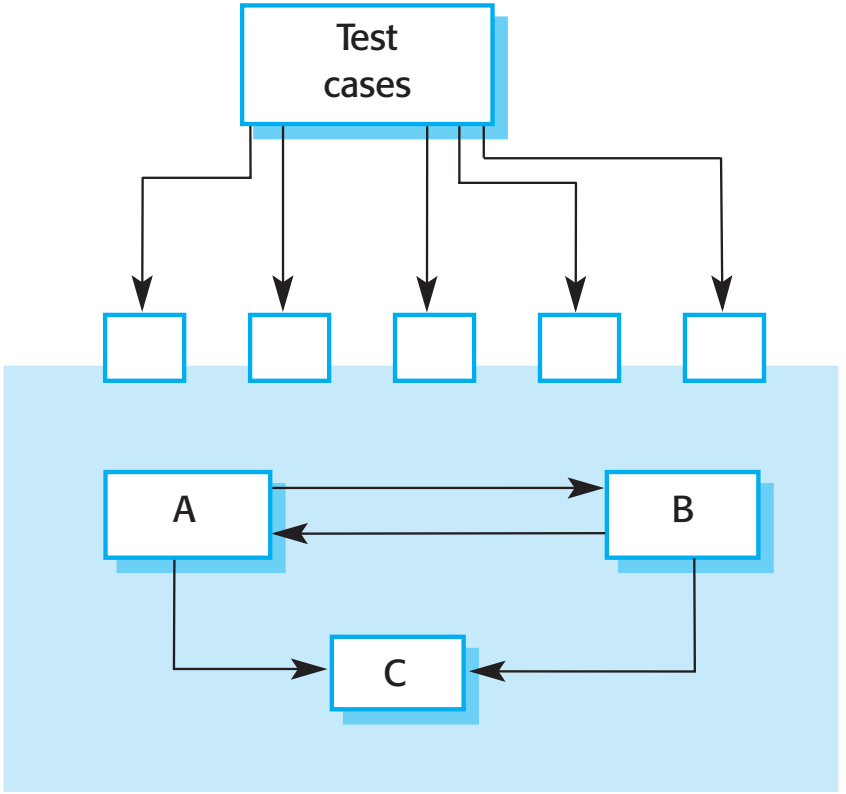
Component Testing

Component Testing

- Software components are often composite components that are made up of several interacting objects.
  - ✓ For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - ✓ You can assume that unit tests on the individual objects within the component have been completed.

Interface Testing

- Objectives are to detect faults due to *interface errors* or *invalid assumptions about interfaces*.
- Interface types
  - **Parameter interfaces**: Data passed from one method or procedure to another.
  - **Shared memory interfaces**: Block of memory is shared between procedures or functions.
  - **Procedural interfaces**: Sub-system encapsulates a set of procedures to be called by other sub-systems.
  - **Message passing interfaces**: Sub-systems request services from other sub-systems



Interface Errors

- Interface misuse
  - ✓ A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding
  - ✓ A calling component embeds assumptions about the behaviour of the called component which are incorrect.
- Timing errors
  - ✓ The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface Testing Guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

System Testing

System Testing

- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing the interactions between components.
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing tests the emergent behavior of a system.

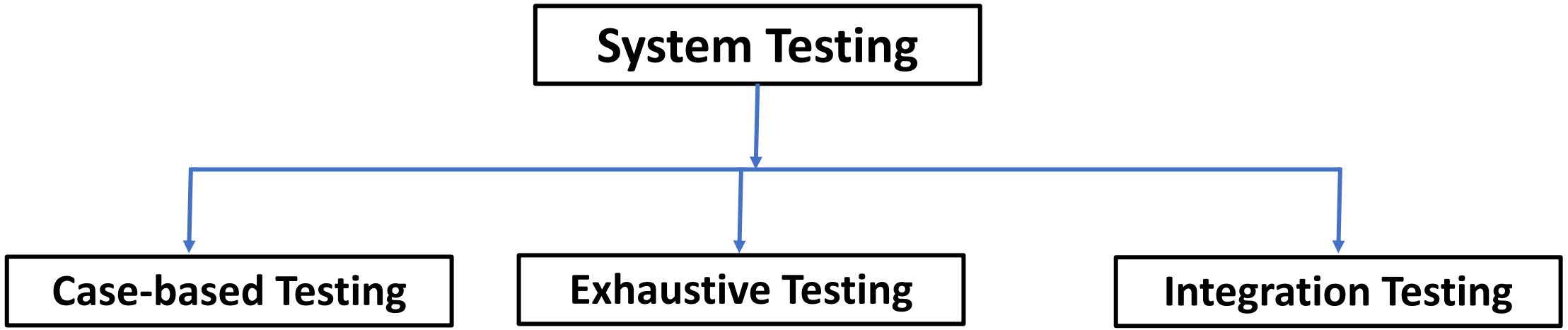
System Testing Examples:

- ✓ Function testing
- ✓ Structure Testing
- ✓ Performance testing
- ✓ Acceptance testing
- ✓ Installation testing

System and Component Testing

- During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
  - ✓In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

System Testing

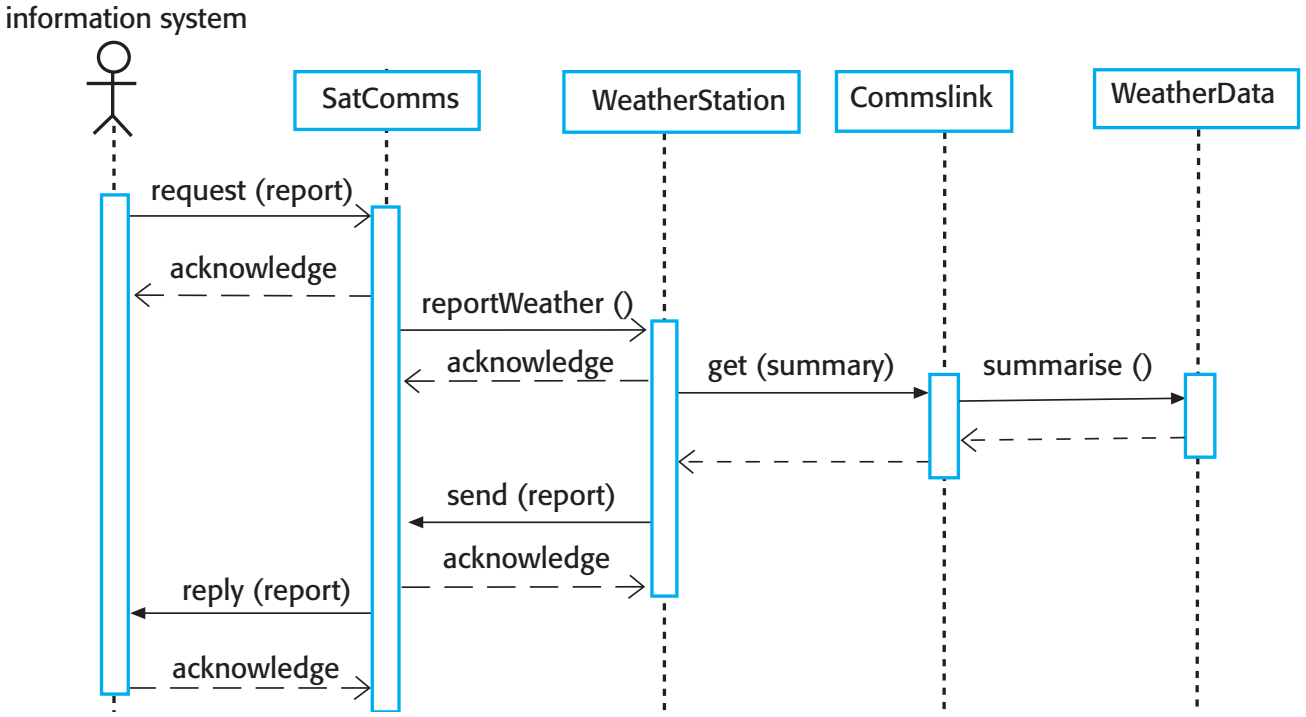


- ✓ Testing is performed for specific cases like operation, interface, acknowledgement, etc.
- ✓ Testing of all components and all cases.
- ✓ Really impossible!
- ✓ Integrate system incrementally and then perform testing

Case-based Testing

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- The sequence diagrams associated with the use case documents the components and interactions that are being tested.

An Example of Case-based Testing



A Sequence Diagram for generating report in a Weather Station

- An input of a request for a report should have an associated acknowledgement. A report should ultimately be returned from the request.
  - ✓ You should create summarized data that can be used to check that the report is correctly organized.
- An input request for a report to WeatherStation results in a summarized report being generated.
  - ✓ Can be tested by creating raw data corresponding to the summary that you have prepared for the test of SatComms and checking that the WeatherStation object correctly produces this summary. This raw data is also used to test the WeatherData object.

Exhaustive Testing

- Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- Examples of testing policies:
  - ✓ All system functions that are accessed through menus should be tested.
  - ✓ Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - ✓ Where user input is provided, all functions must be tested with both correct and incorrect input.

Integration Testing

- System testing involves integrating different components, then testing the integrated system.
- You should always use an incremental approach to integration and testing where you integrate a component, test the system, integrate another component, test again, and so on.
- If problems occur, they are probably due to interactions with the most recently integrated component.
- Incremental integration and testing is fundamental to agile methods, where regression tests are run every time a new increment is integrated.

Release Testing

Release Testing

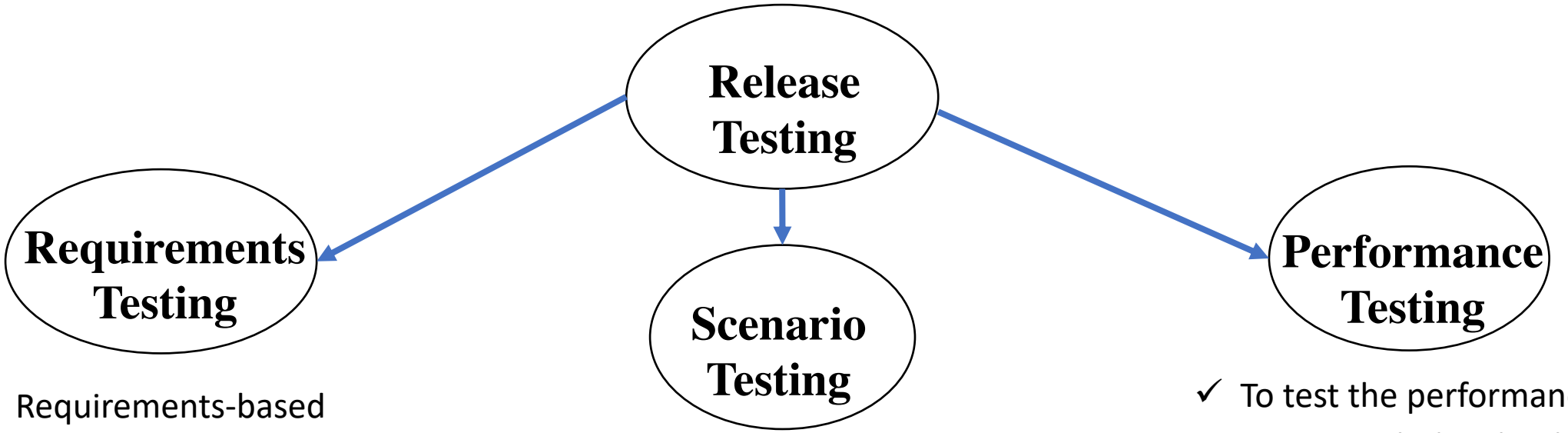
- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
  - ✓ Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a black-box testing process where tests are only derived from the system specification.

Release Testing vs. System Testing

- Release testing is a form of system testing.

System Testing	Release Testing
System Testing is performed by the Development Team.	A separate team that has not been involved in the system development, should be responsible for release testing.
Defect Testing: System Testing should focus on discovering bugs in the system.	**Validation Testing**: The objective of release testing is to check that the system meets its requirements and is good enough for external use.

Release Testing



- ✓ Requirements-based testing involves examining each requirement and developing a test or tests for it.
- ✓ To test whether a scenario is working perfectly or not
- ✓ To test the performance of the system including load, dependability, reliability, latency, etc.

An Example of Requirements-based Testing

Mentcare system requirements:

- ✓ If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
- ✓ If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

Test Plan:

- Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
- Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system.
- Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
- Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
- Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

An Example of Scenario Testing

A usage scenario for the Mentcare system

George is a nurse who specializes in mental healthcare. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients that he visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. George looks up Jim's record and is prompted for his key phrase to decrypt the record. He checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so he notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. Jim agrees so George enters a prompt to call him when he gets back to the clinic to make an appointment with a physician. George ends the consultation and the system re-encrypts Jim's record.

After, finishing his consultations, George returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for George of those patients who He has to contact for follow-up information and make clinic appointments.

An Example of Scenario Testing

Features tested by scenario:

- ✓Authentication by logging on to the system.
- ✓Downloading and uploading of specified patient records to a laptop.
- ✓Home visit scheduling.
- ✓Encryption and decryption of patient records on a mobile device.
- ✓Record retrieval and modification.
- ✓Links with the drugs database that maintains side-effect information.
- ✓The system for call prompting.

Performance Testing

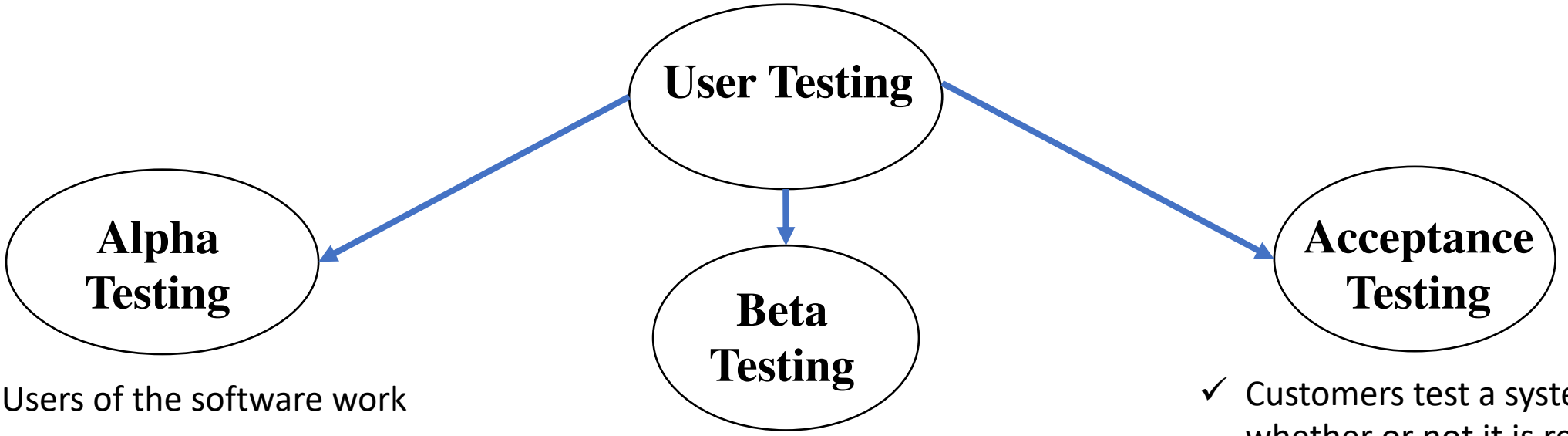
- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the profile of use of the system.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.

User Testing

User Testing

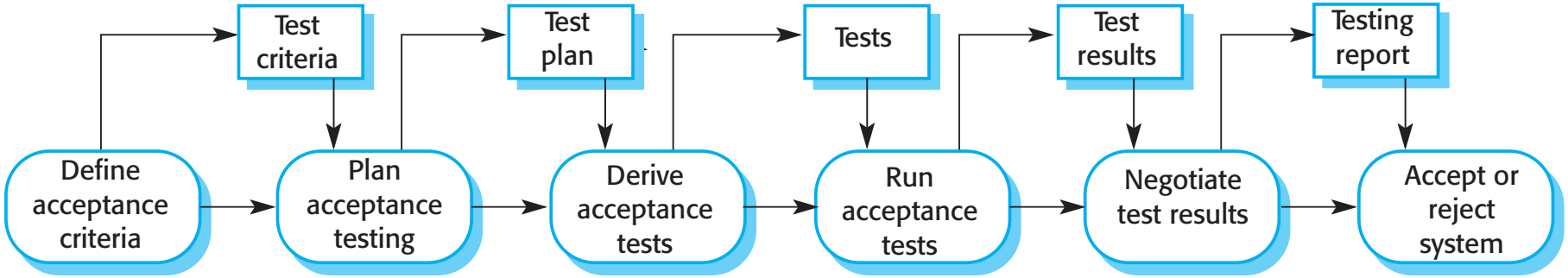
- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
  - ✓ The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

User Testing



- ✓ Users of the software work with the development team to test the software at the developer's site.
- ✓ A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- ✓ Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

Acceptance Testing Process

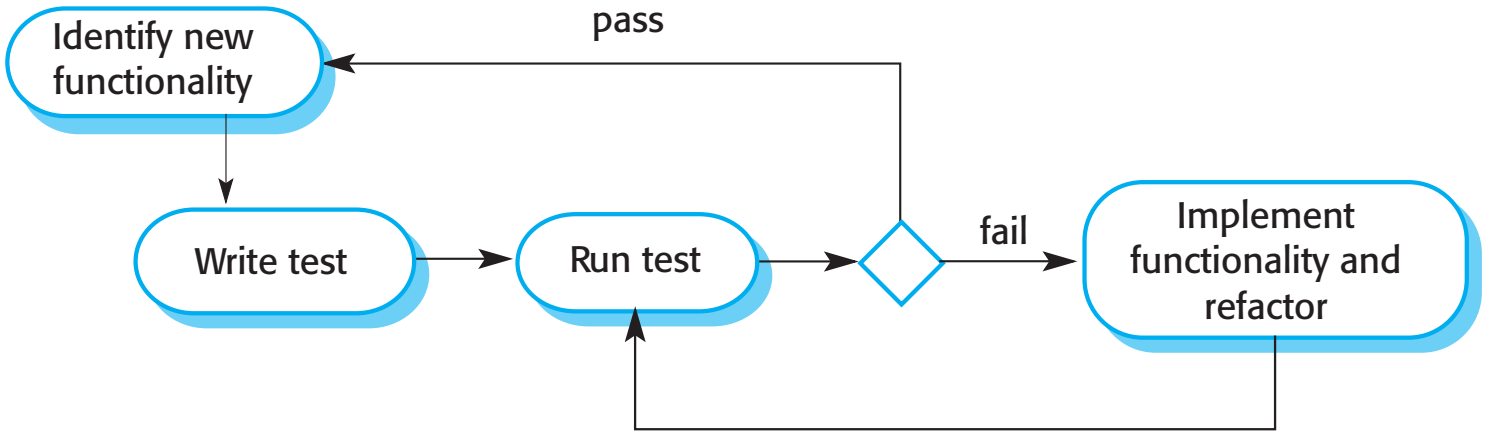


Acceptance Testing in Agile Methods:

- ✓ In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- ✓ Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- ✓ There is no separate acceptance testing process.
- ✓ Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.

Test-Driven Development (TDD)

Test-Driven Development (TDD)



- ✓Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- ✓Tests are written before code and 'passing' the tests is the critical driver of development.
- ✓You develop code incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test.
- ✓TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

TDD Process Activities

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and re-run the test.
- Once all tests run successfully, you move on to implementing the next chunk of functionality.

Benefits of TDD

- Code coverage
  - ✓ Every code segment that you write has at least one associated test so all code written has at least one test.
- Regression testing
  - ✓ A regression test suite is developed incrementally as a program is developed.
- Simplified debugging
  - ✓ When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- System documentation
  - ✓ The tests themselves are a form of documentation that describe what the code should be doing.

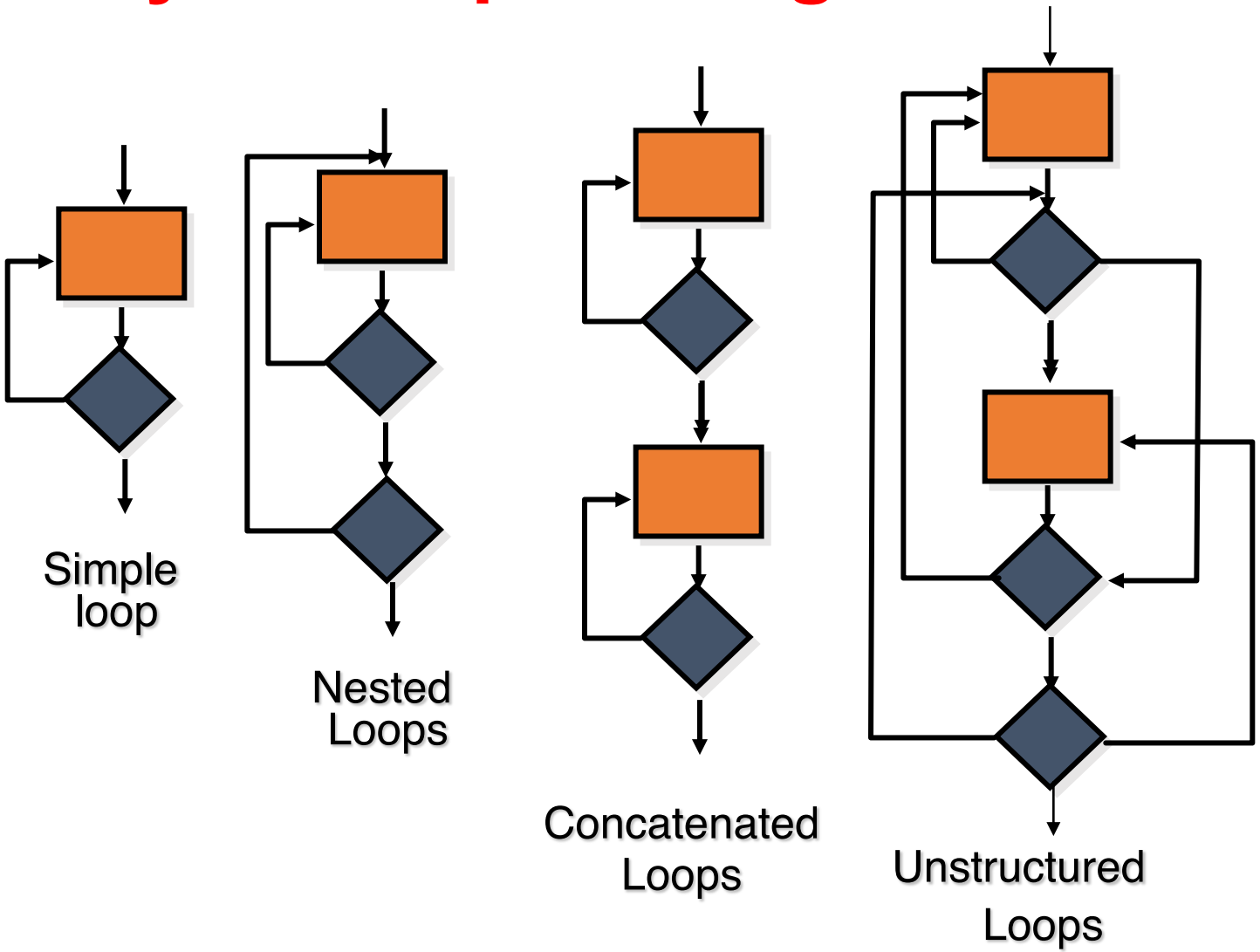
Regression Testing

- Regression testing is testing the system to check that changes have not ‘broken’ previously working code.
- In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program.
- Tests must run ‘successfully’ before the change is committed.

White-box Testing

- **Focus on Thoroughness (Coverage)**: Every statement in the component is executed at least once.
- **Four types of white-box testing**
  - **Statement Testing (Algebraic Testing)**: Test single statements.
  - **Loop Testing**:
    - ✓ Cause execution of the loop to be skipped completely. (Exception: Repeat loops)
    - ✓ Loop to be executed exactly once
    - ✓ Loop to be executed more than once
  - **Path Testing**: Make sure all paths in the program are executed.
  - **Branch Testing (Conditional Testing)**: Make sure that each possible outcome from a condition is tested at least once.

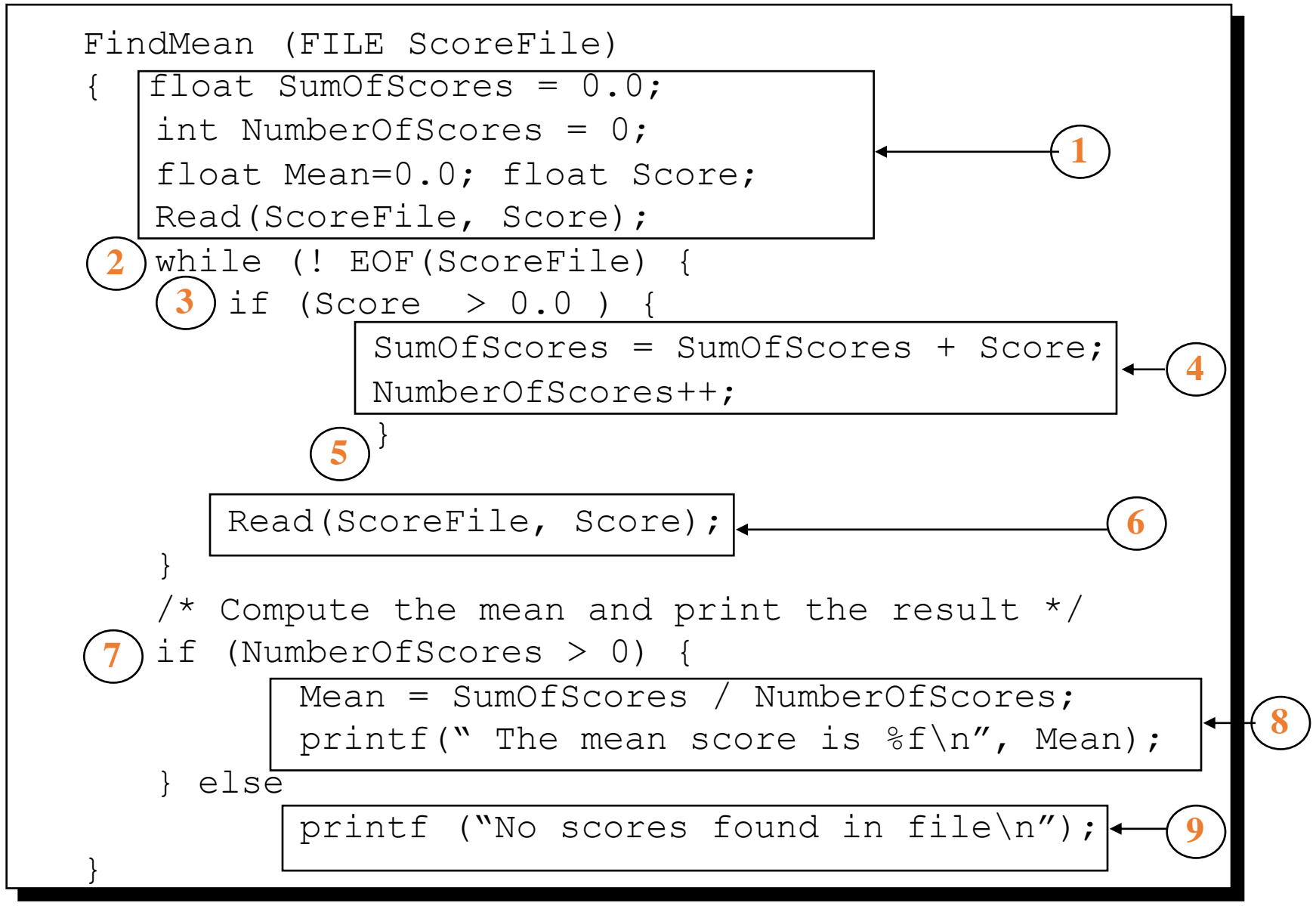
Complexity of Loop Testing



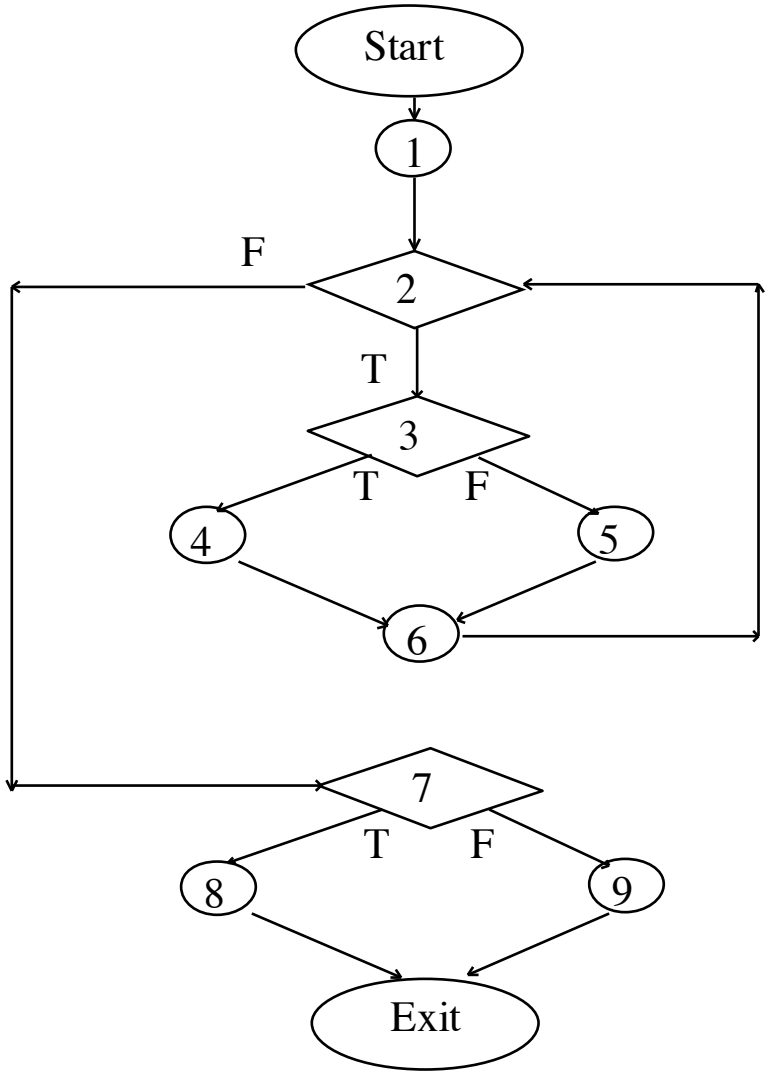
White-box Testing Example

```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
         if ( Score > 0.0 ) {
               SumOfScores = SumOfScores + Score;
               NumberOfScores++;
          }
          Read(ScoreFile, Score);
 }
 /* Compute the mean and print the result */
 if (NumberOfScores > 0 ) {
             Mean = SumOfScores/NumberOfScores;
             printf("The mean score is %f \n",  Mean);
 } else
             printf("No scores found in file\n");
}
```

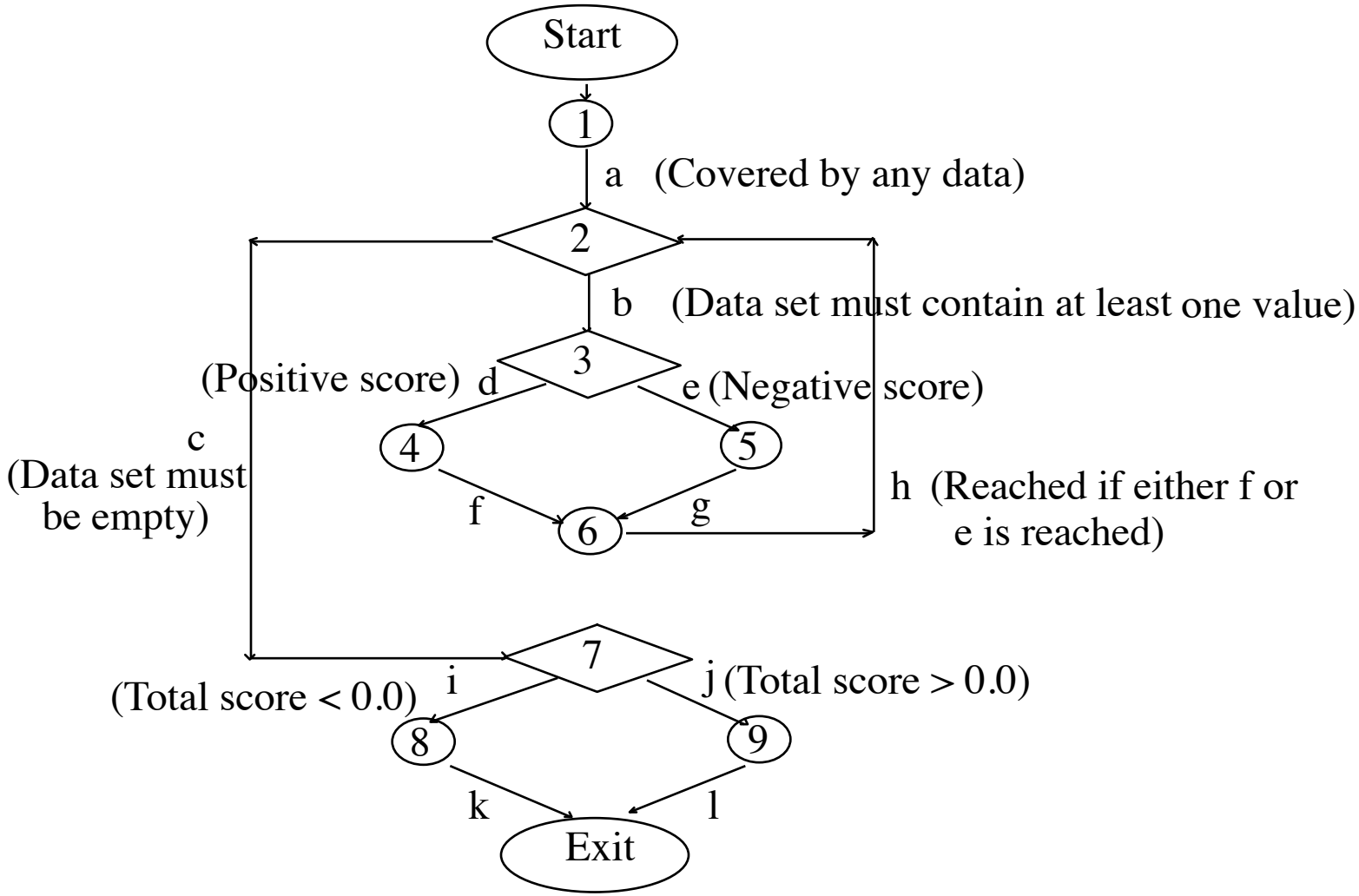
White-box Testing Example: Determining the Paths



White-box Testing Example: Constructing the Logic Flow Diagram



White-box Testing Example: Finding the Test Cases



Black-box Testing

- **Focus on I/O behavior:** If for any given input, we can predict the output, then the module passes the test.
  - ✓ Almost always impossible to generate all possible inputs ("test cases")

- **Goal:** Reduce number of test cases by equivalence partitioning:
  - ✓ Divide input conditions into equivalence classes
  - ✓ Choose test cases for each equivalence class. (Example: If an object is supposed to accept a negative number, testing one negative number is enough)

Black-box Testing: Reducing inputs

- Selection of equivalence classes (***No*** rules, only guidelines):
  - ✓ Input is valid across range of values. Select test cases from 3 equivalence classes:
    - Below the range
    - Within the range and
    - Above the range.
  - ✓ Input is valid if it is from a discrete set. Select test cases from 2 equivalence classes:
    - Valid discrete value and
    - Invalid discrete value.
- Another solution to select only a limited amount of test cases:
  - ✓ Get knowledge about the inner workings of the unit being tested => white-box testing

White-box vs. Black-box Testing

➢ **White-box Testing:**

- ✓ Potentially infinite number of paths have to be tested
- ✓ White-box testing often tests what is done, instead of what should be done
- ✓ Cannot detect missing use cases

➢ **Black-box Testing:**

- ✓ Potential combinatorical explosion of test cases (valid & invalid data)
- ✓ Often not clear whether the selected test cases uncover a particular error
- ✓ Does not discover extraneous use cases ("features")

➢ Both types of testing are needed

➢ White-box testing and black box testing are the extreme ends of a testing continuum.

➢ Any choice of test case lies in between and depends on the following:

- Number of possible logical paths
- Nature of input data
- Amount of computation
- Complexity of algorithms and data structures