SOFTWARE TESTING STRATEGIES

A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is a set of activities that can be planned in advance and conducted systematically. A strategy for software testing is developed by the project manager, software engineers, and testing specialists. Testing begins "in the small" and progresses "to the large." By this we mean that early testing focuses on a single component or on a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the component(s). After components are tested they must be integrated until the complete system is constructed. At this point, a series of high-order tests are executed to uncover errors in meeting customer requirements. As errors are uncovered, they must be diagnosed and corrected using a process that is called debugging.

A number of **software testing strategies** have been proposed ,all have the following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews . By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). ***Verification*** refers to the set of tasks that ensure that software correctly implements a specific function. ***Validation*** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way:

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

Software Testing Strategy—The Big Picture

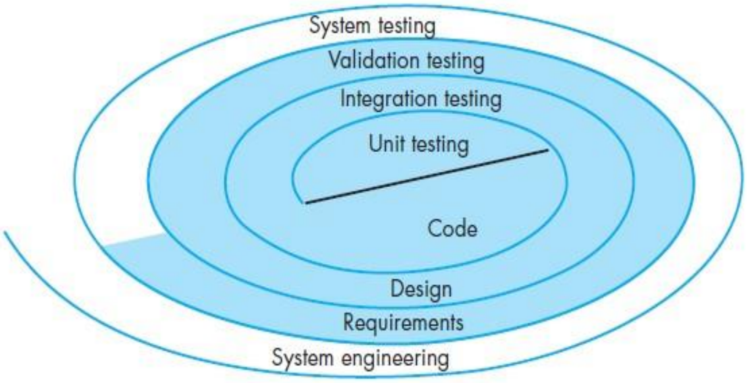


Figure Testing Strategy

A strategy for software testing may also be viewed in the context of the spiral (Figure).

- *Unit testing* begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code.
- Testing progresses by moving outward along the spiral to *integration testing*, where the focus is on design and the construction of the software architecture.
- Taking another turn outward on the spiral, you encounter *validation testing*, where requirements established as part of requirements modeling are validated against the software that has been constructed.
- Finally, you arrive at *system testing*, where the software and other system elements are tested as a whole. To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn.
- Initially, tests focus each component individually, ensuring that it functions properly as a unit. Hence, the name ***unit testing***. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.
- Next, components must be assembled or integrated to form the complete software package. *Integration testing* addresses the issues associated with the dual problems of verification and program construction. Testcase design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths.
- After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria (established during requirements analysis) must be evaluated. *Validation testing* provides final assurance that software meets all functional, behavioral, and performance requirements.

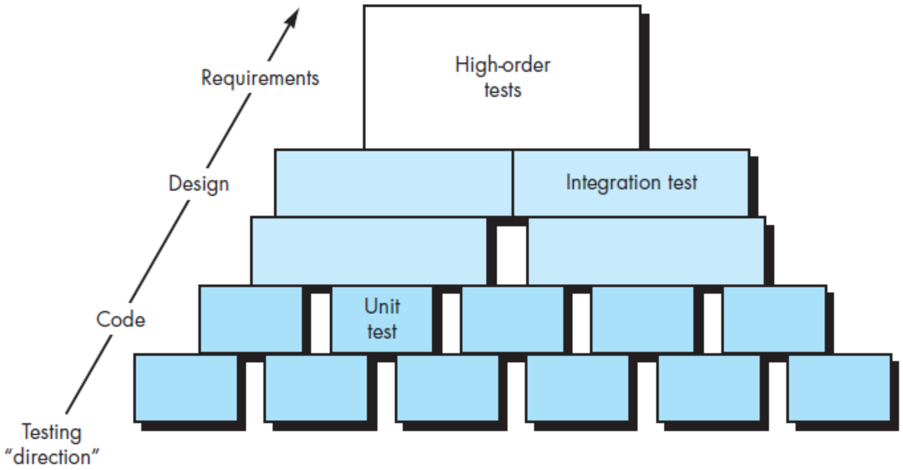


Figure : Software Testing steps

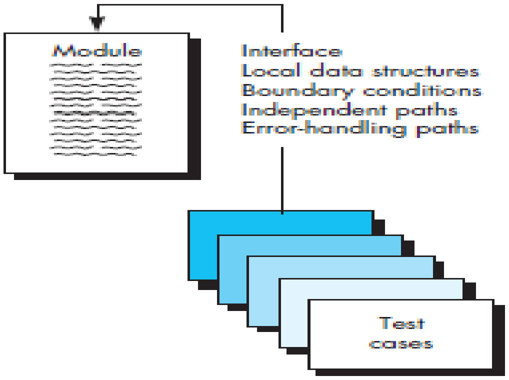
- The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases).
- *System testing* verifies that all elements mesh properly and that overall system function/performance is achieved.

Unit Testing

- *Unit testing* focuses verification effort on the smallest unit of software design—the software component or module.
- Using the component-level design description, important control paths are tested to uncover errors within the boundary of the module.
- The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel formultiple components.

Unit Test Considerations.

- Unit tests are illustrated schematically in Figure.
- The module interface is tested to ensure that information properly flows into and out of the program unit under test.
- Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error handling paths are tested.

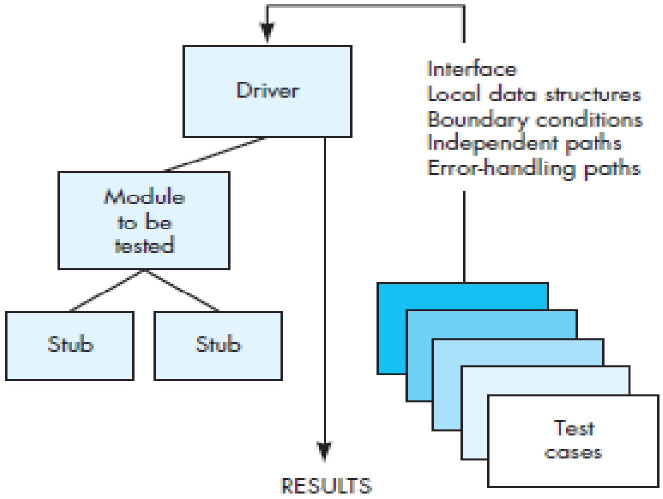


- Data flow across a component interface is tested before any other testing is initiated.
- If data do not enter and exit properly, all other tests are moot.
- In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.
- Selective testing of execution paths is an essential task during the unit test.

- Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.
- Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n* th element of an *n* -dimensional array is processed, when the *i*th repetition of a loop with *I* passes is invoked, when the maximum or minimum allowable value is encountered.
- Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.
- A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur -*antibugging*.
- Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible,
  (2) error noted does not correspond to error encountered,
  (3) error condition causes system intervention prior to error handling,
  (4) exception-condition processing is incorrect, or
  (5) error description does not provide enough information to assist in the location of the cause of the error.

<u>Unit-Test Procedures.</u>

- Unit testing is normally considered as an adjunct tothe coding step. The design of unit tests can occur before coding begins or after source code has been generated.
- A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories .
- Each test case should be coupled with a set of expected results.



- Because a component is not a stand-alone program, driver and/or stub softwaremust often be developed for each unit test.
- The unit test environment is illustrated in Figure .
- In most applications a *driver* is nothing more than a “main program” that accepts test-case data, passes such data to the component(to be tested), and prints relevant results.
- *Stubs* serve to replace modules that are subordinate (invoked by) the component to be tested.

- A stub or “dummy subprogram” uses the subordinate module’s interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- Drivers and stubs represent testing “overhead.” That is, both are software that must be coded (formal design is not commonly applied) but that is not delivered with the final software product.
- If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with “simple” overhead software.
- In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Integration Testing

- Components must be assembled or integrated to form the complete software package. where the focus is on design and the construction of the software architecture.
- *Integration testing* addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths.
- Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.
- To construct the program using a “**big bang**” approach. All components are combined in advance and the entire program is tested as a whole. Errors are encountered, but correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.
- **Incremental integration** is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

Top-Down Integration.

- *Top-down integration testing* is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth first or breadth-first manner. Referring to Figure below ,*depth-first integration* integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.
- For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated.
- Then, the central and right-hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

- The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actualcomponents.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the realcomponent.
5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

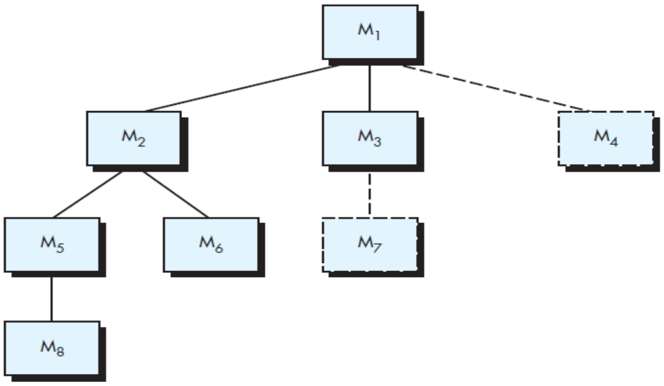


Figure : Top down integration

The process continues from step 2 until the entire program structure is built. The top-down integration strategy verifies major control or decision points earlyin the test process. In a “well-factored” program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. Early demonstration of functional capability is a confidence builder for all stakeholders.

Bottom-Up Integration. *Bottom-up integration testing,* as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
2. A *driver* (a control program for testing) is written to coordinate test-case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

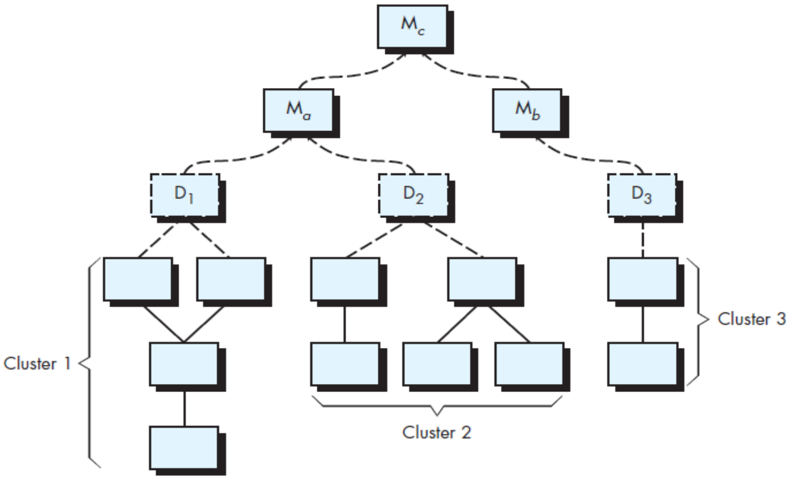


Figure : Bottom up approach

Integration follows the pattern illustrated in Figure . Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M*a* . Drivers D1 and D2 are removed and the clusters are interfaced directly to M*a* . Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb.

Both M*a* and M*b* will ultimately be integrated with component M*c* , and so forth. As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

Regression Testing.

- Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. Side effects associated with these changes may cause problems with functions that previously worked flawlessly.
- Regression testing refers to a type of software testing that is used to verify any modification or update in a software without affecting the overall working functionality of the said software**.**
- Test cases are re-executed to check the previous functionality of the application is working fine, and the new changes have not produced any bugs.
- In the context of an integration test strategy, ***regression testing*** is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools.
- ***Capture/playback tools*** enable the software engineer to capture test cases and results for subsequent playback and comparison.

The ***regression test suite*** (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.

Smoke Testing.

- ***Smoke testing*** is an integration testing approach that is commonly used when product software is developed.
- Smoke testing is a process where the software build is deployed to quality assurance environment and is verified to ensure the stability of the application. Smoke Testing is also known as ***Confidence Testing* or *Build Verification Testing*.**
- **Smoke Testing** is a software testing process that determines whether the deployed software build is stable or not. Smoke testing is a confirmation for QA team to proceed with further software testing. It consists of a minimal set of tests run on each build to test software functionalities.
- It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.
- **Smoke Testing** is a software testing process that determines whether the deployed software build is stable or not. Smoke testing is a confirmation for QA team to proceed with further software testing.
- It consists of a minimal set of tests run on each build to test software functionalities. Smoke testing is also known as "**Build Verification Testing" or "Confidence Testing**."

In essence, the smoke-testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show-stopper" errors that have the highest likelihood of throwing the software project behind schedule.
3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

- The smoke test should exercise the entire system from end to end
- Smoke testing provides a number of benefits when it is applied on complex,time-critical software projects:
- ***Integration risk is minimized***. Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- ***The quality of the end product is improved***. Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as

architectural and component-level design errors. If these errors are corrected early, better product quality will result.

- ***Error diagnosis and correction are simplified***. Like all integration testing approaches, errors uncovered during smoke testing are likely to be associatedwith “new software increments”—that is, the software that has justbeen added to the build(s) is a probable cause of a newly discovered error.
- ***Progress is easier to assess***. With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

TEST STRATEGIES FOR WEBAPPS

The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems.

The following steps summarize the approach:

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Each functional component is unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for errors.

TEST STRATEGIES FOR MOBILEAPPS

The strategy for testing mobile applications adopts the basic principles for all software testing. However, the unique nature of MobileApps demands the consideration of a number of specialized testing approaches:

- *User-experience testing.* Users are involved early in the development process to ensure that the MobileApp lives up to the usability and accessibility expectations of the stakeholders on all supported devices.
- *Device compatibility testing.* Testers verify that the MobileApp works correctly on all required hardware and software combinations.
- *Performance testing.* Testers check nonfunctional requirements unique to mobile devices (e.g., download times, processor speed, storage capacity, power availability).
- *Connectivity testing.* Testers ensure that the MobileApp can access any needed networks or Web services and can tolerate weak or interrupted network access.

- *Security testing* . Testers ensure that the MobileApp does not compromise the privacy or security requirements of its users.
- *Testing-in-the-wild* . The app is tested under realistic conditions on actual user devices in a variety of networking environments around the globe.
- Certification testing. Testers ensure that the MobileApp meets the standards established by the app stores that will distribute it.

C .VALIDATION TESTING

- The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.
- Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.
- Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At thevalidation or system level, the distinction between different software categories disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

C1 :Validation-Test Criteria

- Software validation is achieved through a series of tests that demonstrate conformity with requirements.
- A test plan outlines the classes of tests to be conducted, and a test procedure define specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).
- If a deviation from specification is uncovered, a *deficiency list* is created. A method for resolving deficiencies (acceptable to stakeholders) must be established.

C2 :Alpha and Beta Testing

- When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.
- Conducted by the end user rather than software engineers, an acceptance test can range from an informal “test drive” to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.
- The ***alpha test*** is conducted at the developer’s site by a representative group of end users. The software is used in a natural setting with the developer “looking over the shoulder” of

the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

- The ***beta test*** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a “live” application of the software in an environment that cannot be controlled by the developer.
- The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.
- A variation on beta testing, called ***customer acceptance testing***, is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer.

D: SYSTEM TESTING

D.1 :Recovery Testing

- Many computer-based systems must recover from faults and resume processing with little or no downtime.
- In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.
- ***Recovery testing*** is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

D.2: Security Testing

- Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain.
- Security testing is an integral part of software testing, which is used to discover the weaknesses, risks, or threats in the software application and also help us to stop the nasty attack from the outsiders and make sure the security of our software applications.
- The primary objective of security testing is to find all the potential ambiguities and vulnerabilities of the application so that the software does not stop working. If we perform security testing, then it helps us to identify all the possible security threats and also help the programmer to fix those errors.

- It is a testing procedure, which is used to define that the data will be safe and also continue the working process of the software.

D.3 Stress Testing

- **Stress Testing** is a type of software testing that verifies stability & reliability of software application.
- The goal of Stress testing is measuring software on its robustness and error handling capabilities under extremely heavy load conditions and ensuring that software doesn't crash under crunch situations. It even tests beyond normal operating points and evaluates how software works under extreme conditions.It is also known as ***Endurance Testing, fatigue testing*** or ***Torture Testing***.
- The stress testing includes the **testing beyond standard operational size**, repeatedly to a **breaking point**, to get the outputs.
- It highlights the error handling and robustness under a heavy load instead of correct behavior under regular conditions.
- In other words, we can say that **Stress testing** is used to verify the constancy and dependability of the system and also make sure that the system would not crash under disaster circumstances.
- *Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate 10 interrupts per second, when one or two is the average rate, (2)input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.
- A variation of stress testing is a technique called *sensitivity testing.* In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

D.4 Performance Testing

- Performance testing is a non-functional software testing technique that determines how the stability, speed, scalability, and responsiveness of an application holds up under a given workload.
- For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable.
- Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are

conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis
- **D.5 Deployment Testing**
  - Software must execute on a variety of platforms and under more than one operating system environment.
  - *Deployment testing,* sometimes called *configuration testing,* exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g.,"installers") that will be used by customers, and all documentation that will be used to introduce the software to end users.

E :THE ART OF DEBUGGING

- *Debugging* occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error.
- Although debugging can and should be an orderly process, it is still very much an

E.1 The Debugging Process

Debugging is not testing but often occurs as a consequence of testing, the debugging process begins with the execution of a test case.

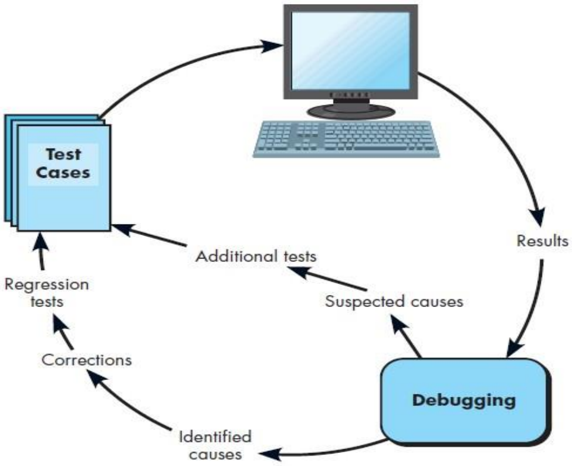


Figure 22.7 Debugging process

Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the non corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

However, a few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components (Chapter 12) exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing Problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real- time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to fi nd the cause also increases. Often, pressure forces a software developer to fi x one error and at the same time introduce two more.

E2 Debugging Strategies

Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

- Debugging has one overriding objective— to find and correct the cause of a software error or defect.
- The objective is realized by a combination of systematic evaluation, intuition, and luck. In general, three debugging strategies have been proposed**: brute force, backtracking, and cause elimination**. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Debugging Tactics.

- The ***brute force*** **category** of debugging is probably the most common and least efficient method for isolating the cause of a software error.
- This is the foremost common technique of debugging however is that the least economical method. During this approach, the program is loaded with print statements to print the intermediate values with the hope that a number of the written values can facilitate to spot the statement in error. This approach becomes a lot of systematic with the utilisation of a symbolic program (also known as a source code debugger), as a result of values of various variables will be simply checked and breakpoints and watch-points can be easily set to check the values of variables effortlessly.

- ***Backtracking***is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.
- The third approach to debugging**—** ***cause elimination***—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.
- A “cause hypothesis”is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Automated Debugging.

- Each of these debugging approaches can be supplemented with debugging tools that can provide you with semiautomated support as debugging strategies are attempted.
- Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation.”
- A wide variety of debugging compilers, dynamic debugging aids (“tracers”), automatic test-case generators, and cross-reference mapping tools are available. However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

SOFTWARE TESTING FUNDAMENTALS

- The goal of testing is to find errors, and a good test is one that has a high probability of finding an error.

The tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

- ✓ **Testability**
- ✓ **Operability**
- ✓ **Observability**.
- ✓ **Controllability**.
- ✓ **Decomposability**.
- ✓ **Simplicity**
- ✓ **Stability**.
- ✓ **Understandability**.

Test Characteristics

- A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
- A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).
- A good test should be "best of breed" .In a group of tests that have a similar intent, time and resource limitations may dictate the execution of only those tests that has the highest likelihood of uncovering a whole class of errors.
- A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately

INTERNAL AND EXTERNAL VIEWS OF TESTING

Any engineered product can be tested in one of two ways:

(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function. :The first test approach takes an external view and is **called black-box testing.**

(2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The second requires an internal view and is termed **white-box testing.**

A:WHITE-BOX TESTING

White-box testing, sometimes called glass-box testing or structural testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, we can derive test cases **that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.**

A1: BASIS PATH TESTING

Basis path testing is a white-box testing technique which enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

A.1.1 Flow Graph Notation

The flow graph depicts logical control flow using the notation illustrated in Figure

FIGURE 23.1 Flow graph notation

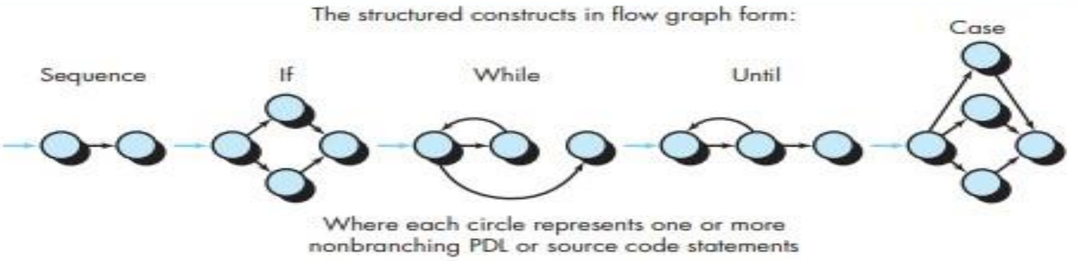
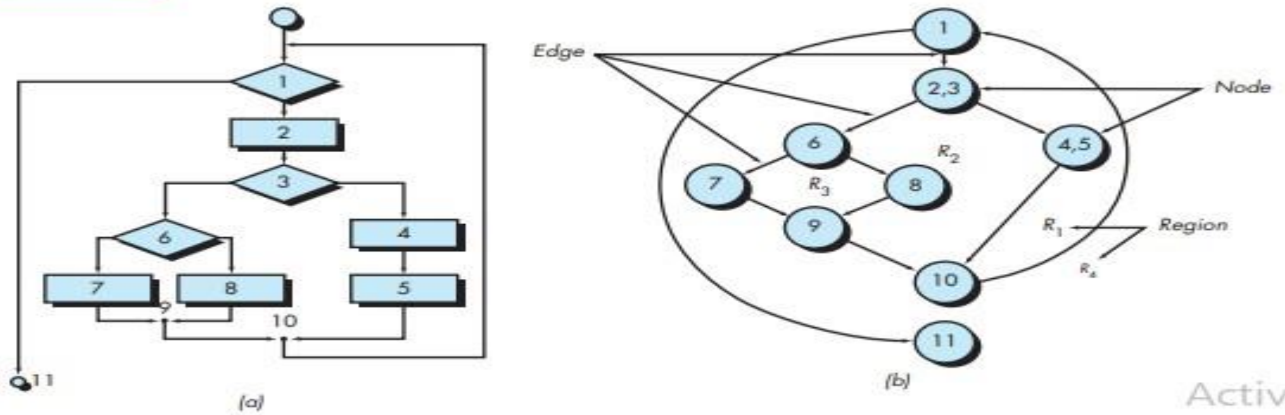

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

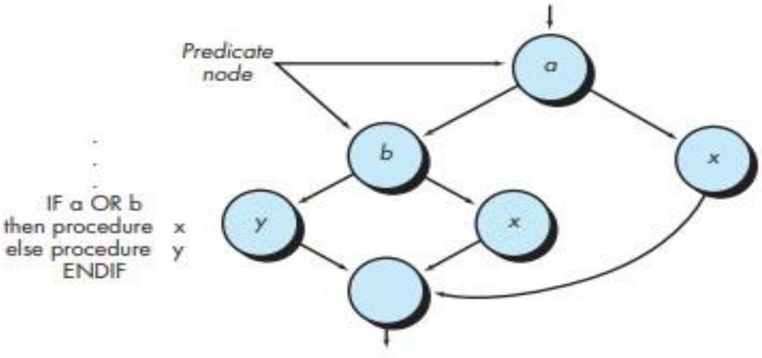
FIGURE 23.2 (a) Flowchart and (b) flow graph



- To illustrate the use of a flow graph, consider the procedural design representation in Figure 23.2a . Here, a flowchart is used to depict program control structure.
- Figure 23.2bmaps the flowchart into a corresponding flow
- Referring to Figure 23.2b, each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node. The arrows on the w graph, called edges or links, represent flow of control and are analogous to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct)
- Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.

When compound conditions are encountered in a procedural design, the generation of aflow graph becomes slightly more complicated. A compound condition occurs when one or more

Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure, the program design language (PDL) segment translates into the flow graph shown. Note that a separate node is created for each of the conditions a and b in the statement IF a OR b. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.



A.1.2 Independent Program Paths

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 23.2 b is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge.

The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

Paths 1 through 4 constitute a basis set for the flow graph in Figure 23.2b . That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:

1. **The number of regions of the flow graph corresponds to the cyclomatic complexity.**

2. Cyclomatic complexity V(G) for a flow graph G is defined as V(G) =E-N+ 2

where E is the number of flow graph edges and N is the number of flow graph nodes.

3. Cyclomatic complexity V(G) for a flow graph G is also defined as V(G) =P+1

where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure 23.2b , the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has **four** regions.

2. V(G) = 11 edges - 9 nodes + 2 = 4.

3. V(G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 23.2bis 4.

More important, the value for V(G) provides you with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements

A.1.3 Graph Matrices

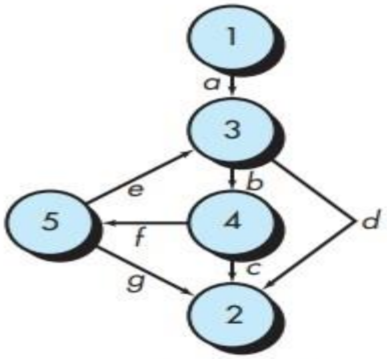
A data structure, called a graph matrix, can be quite useful for developing a software tool that assists in basis path testing.

A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix Figure below.

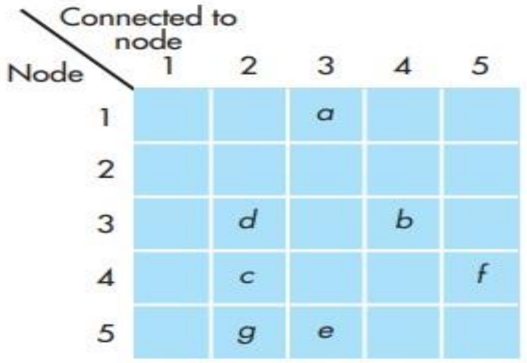
Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge b.

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link
- The memory required during traversal of a link
- The resources required during traversal of a link.



Flow graph



Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

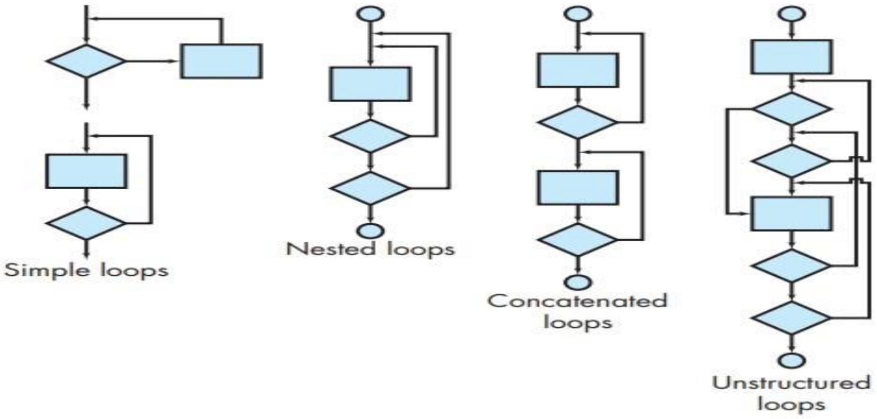
Graph matrix

2. CONTROL STRUCTURE TESTING

The basis path testing technique is one of a number of techniques for control structure testing.

Condition testing is a test-case design method that exercises the logical conditions contained in a program module. Data flow testing selects test paths of a program according to the locations of definitions and uses of variables in the program.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops .



Simple Loops. The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.

2. Only one pass through the loop.

3. Two passes through the loop.

4. m passes through the loop where m < n.

5. n - 1, n, n + 1 passes through the loop.

Nested Loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. Beizer suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.

3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.

4. Continue until all loops have been tested.

Concatenated Loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured Loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

3. BLACK-BOX TESTING

Black-box testing, also called behavioral testing or functional testing focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program

- Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavior or performance errors, and (5) initialization and termination errors.
- Black-box testing is focused on the information domain. Black-box tests are designed to validate functional requirements without regard to the internal workings of a program.

- Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage.

Equivalence partitioning divides the input domain into classes of data that are likely to exercise a specific software function.

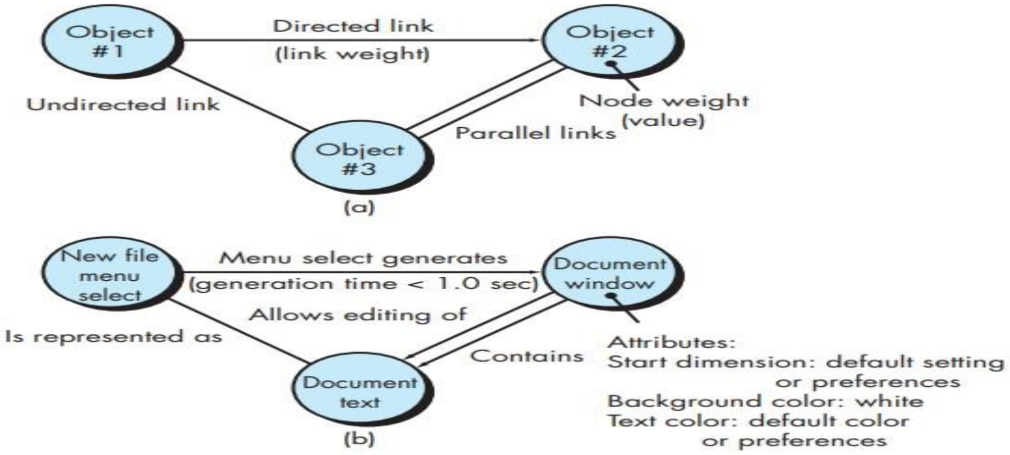
Boundary value analysis probes the program's ability to handle data at the limits of acceptability.

Orthogonal array testing provides an efficient, systematic method for testing systems with small numbers of input parameters.

Model-based testing uses elements of the requirements model to test the behavior of an application.

Graph-Based Testing Methods

The first step in black-box testing is to understand the objects 5 that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another" . Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.



To accomplish these steps, you begin by creating a graph—a collection of nodes that represent objects, links that represent the relationships between objects, node weights that describe the properties of a node (e.g., a specific data value or state behavior), and link weights that describe some characteristic of a link.

The symbolic representation of a graph is shown in Figure a . Nodes are represented as circles connected by links that take a number of different forms. A **directed link** (represented by an arrow) indicates that a relationship moves in only one direction. A **bidirectional link,** also called a symmetric link, implies that the relationship applies in both directions. **Parallel links** are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 23.8b) where

Object #1 = newFile (menu selection)

Object #2 = documentWindow

Object #3 = documentText

Referring to the figure, a menu select on newFile generates a document window. The node weight of documentWindow provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the newFile menu selection and documentText, and parallel links indicate relationships between documentWindow and documentText.

we can then derive test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships. Beizer describes a number of behavioral testing methods that can make use of graphs:

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps

Finite state modeling. The nodes represent different user-observable states of the software (e.g., each of the “screens” that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state input).

Data flow modeling. The nodes are data objects, and the links are the transformations that occur to translate one data object into another

Timing modeling. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

2 Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.

Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present .

An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.

Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifices a range, one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.

4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

3 Boundary Value Analysis

- A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that boundary value analysis (BVA) has been developed as a testing technique.
  - Boundary value analysis leads to a selection of test cases that exercise bounding values.
  - Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

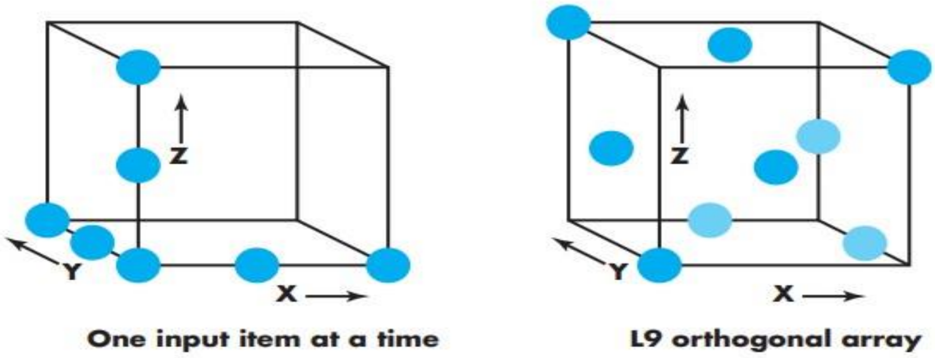
4 Orthogonal Array Testing

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.

The orthogonal array testing method is particularly useful in finding region faults—an error category associated with faulty logic within a software component.

To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, X, Y, and Z.

Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases. Phadke suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure . Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).



When orthogonal array testing occurs, an L9 orthogonal array of test cases is created. The L9 orthogonal array has a "balancing property". That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube in Figure . Test coverage across the input domain is more complete.

To illustrate the use of the L9 orthogonal array, consider the send function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values:

P1 = 1, send it now

P1 = 2, send it one hour later

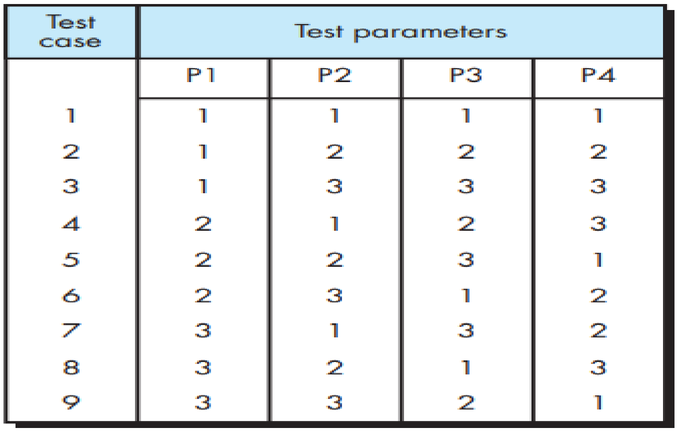
P1 = 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other send functions.

If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1,3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). But these would uncover only single mode faults [Pha97], that is, faults that are triggered by a single parameter.

Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is 3 4 5 81, large but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high.

The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in Figure .



Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Phadke assesses the result of tests using the L9 orthogonal array in the following manner:

Detect and isolate all single mode faults. A single mode fault is a consistent problem with any level of =any single parameter. For example, if all test cases of factor P1 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 [Figure 23.10] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with “send it now” (P1 = 1)] as the source of the error. Such an isolation of fault is important to fix the fault.

Detect all double mode faults. If there exists a consistent problem when specific levels of two parameters occur together, it is called a double mode fault. Indeed, a double mode fault is an indication of pair wise incompatibility or harmful interactions between two test parameters.

Multimode faults. Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multi-mode faults are also detected by these tests.

MODEL -BASED TESTING

Model-based testing (MBT) is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases

In many cases, the model-based testing technique uses UML state diagrams, an element of the behavioral model , as the basis for the design of test cases.

The MBT technique requires five steps:

1. Analyze an existing behavioral model for the software or create one. Recall that a behavioral model indicates how software will respond to external events or stimuli. To create the model, you should perform the steps (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a UML state diagram for the system and (5) review the behavioral model to verify accuracy and consistency.

2. Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state. The inputs will trigger events that will cause the transition to occur.

3. Review the behavioral model and note the expected outputs as the software makes the transition from state to state. Recall that each state transition is triggered by an event and that as a consequence of the transition, some function is invoked and outputs are created. For each set of inputs (test cases) you specified in step 2, specify the expected outputs as they are characterized in the behavioral model.

4. Execute the test cases. Tests can be executed manually or a test script can be created and executed using a testing tool.

5. Compare actual and expected results and take corrective action as required.

MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

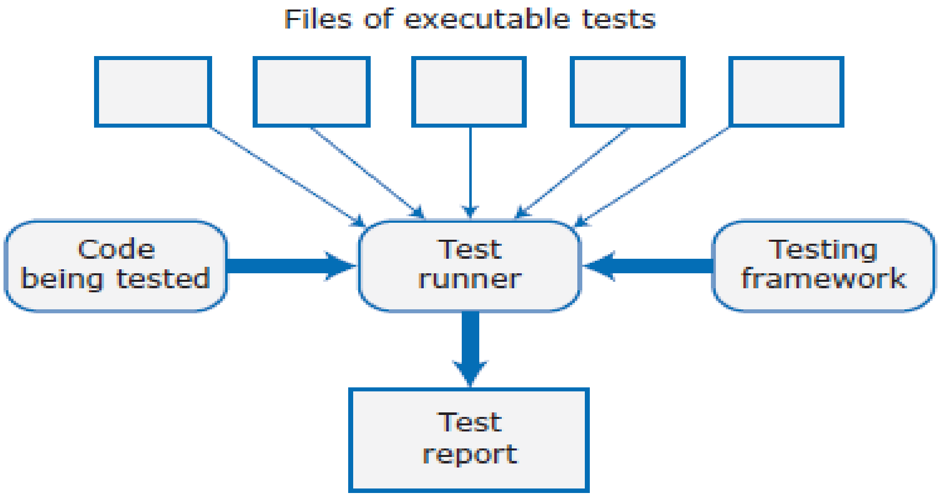
TESTING DOCUMENTATION

Documentation testing can be approached in two phases. The first phase, technical review examines the document for editorial clarity. The second phase, live test, uses the documentation in conjunction with the actual program.

Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed earlier. Graph-based testing can be used to describe the use of the program; equivalence partitioning and boundary value analysis can be used to define various classes of input and associated interactions. MBT can be used to ensure that documented behavior and actual behavior coincide. Program usage is then tracked through the documentation.

Test automation

Figure 9.4 Automated testing



- Automated testing (Figure 9.4) is based on the idea that tests should be executable.
- An executable test includes the input data to the unit that is being tested, the expected result, and a check that the unit returns the expected result.
- we run the test and the test passes if the unit returns the expected result. Normally, we should develop hundreds or thousands of executable tests for a software product.
- The development of automated testing frameworks, such as JUnit for Java in the 1990s, reduced the effort involved in developing executable tests.
- Testing frameworks are now available for all widely used programming languages. A suite of hundreds of unit tests, developed using a framework, can be run on a desktop computer in a few seconds. A test report shows the tests that have passed and failed.
- Testing frameworks provide a base class, called something like "TestCase" that is used by the testing framework. To create an automated test, you define your own test class as a subclass of this TestCase class. Testing frameworks include a way of running all of the tests defined in the classes that are based on TestCase and reporting the results of the tests.

It is good practice to structure automated tests in three parts:

1. *Arrange* You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.

2. *Action* You call the unit that is being tested with the test parameters.

3. *Assert* You make an assertion about what should hold if the unit being tested has executed successfully.

If we use equivalence partitions to identify test inputs, we should have several automated tests based on correct and incorrect inputs from each partition.

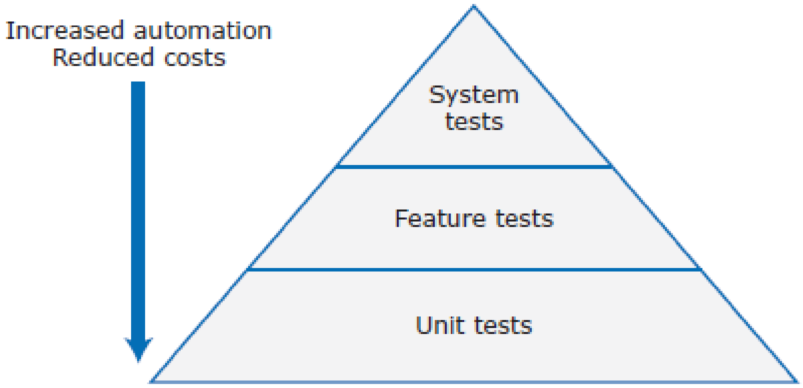
As the point of automated testing is to avoid the manual checking of test outputs, we can't realistically discover test errors by running the tests. Therefore, we have to **use two approaches to reduce the chances of test errors:**

1. Make tests as simple as possible. The more complex the test, the more likely that it will be buggy. The test condition should be immediately obvious when reading the code.

2. Review all tests along with the code that they test. As part of the review process, someone apart from the test programmer should check the tests for correctness.

- Regression testing is the process of re-running previous tests when we make a change to a system.
- This testing checks that the change has not had unexpected side effects. The code change may have inadvertently broken existing code, or it may reveal bugs that were undetected in earlier tests.
- If we use automated tests, regression testing takes very little time. Therefore, after we make any change to your code, even a very minor change, we should always re-run all tests to make sure that everything continues to work as expected.

Figure 9.5 The test pyramid



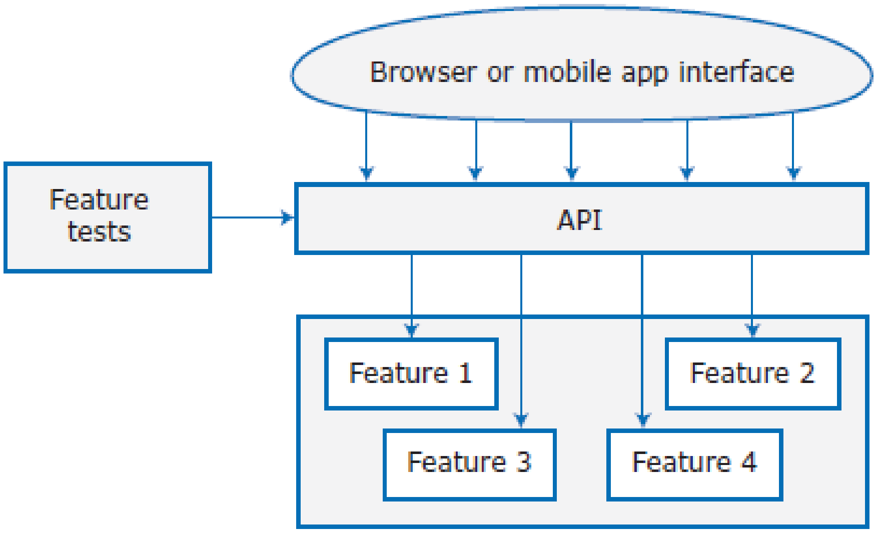
- Unit tests are the easiest to automate, so the majority of your tests should be unit tests. Mike Cohn, who first proposed the test pyramid , suggests that 70% of automated tests should be unit tests, 20% feature tests (he called these service tests), and 10% system tests (UI tests).
- The implementation of system features usually involves integrating functional units into components and then integrating these components to implement the feature. If you have good unit tests, you can be confident that the individual functional units and components that implement the feature will behave as you expect.
- Generally, users access features through the product's graphical user interface (GUI). However, GUI-based testing is expensive to automate so it is best to use an alternative feature testing strategy.
- This involves designing your product so that its features can be directly accessed through an API, not just from the user interface. The feature tests can then access features directly through the API without the need for direct user interaction through the system's GUI (Figure 9.6).

Accessing features through an API has the additional benefit that it is possible to re-implement the GUI without changing the functional components of the software.

For example, a series of API calls may be required to implement a feature that allows a user to share a document with another user by specifying their email address.

These calls collect the email address and the document identification information, check that the access permissions on the document allow sharing, check that the specified email address is valid and is a registered system user, and add the document to the sharing user's workspace

Figure 9.6 Feature testing through an API

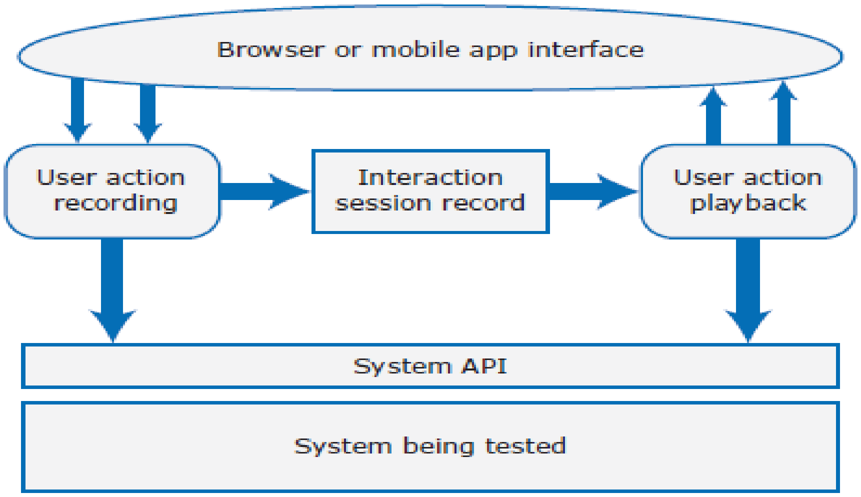


When these calls have been executed, a number of conditions should hold:

- The status of the document is “shared.”
- The list of users sharing the document includes the specified email address.
- There have been no deletions from the list of users sharing the document.
- The shared document is visible to all users in the sharing list.

Manual system testing, when testers have to repeat sequences of actions, is boring and prone to errors. In some cases, the timing of actions is important and is practically impossible to repeat consistently. To avoid these problems, testing tools have been developed to record a series of actions and automatically replay them when a system is retested (Figure 9.7).

Figure 9.7 Interaction recording and playback

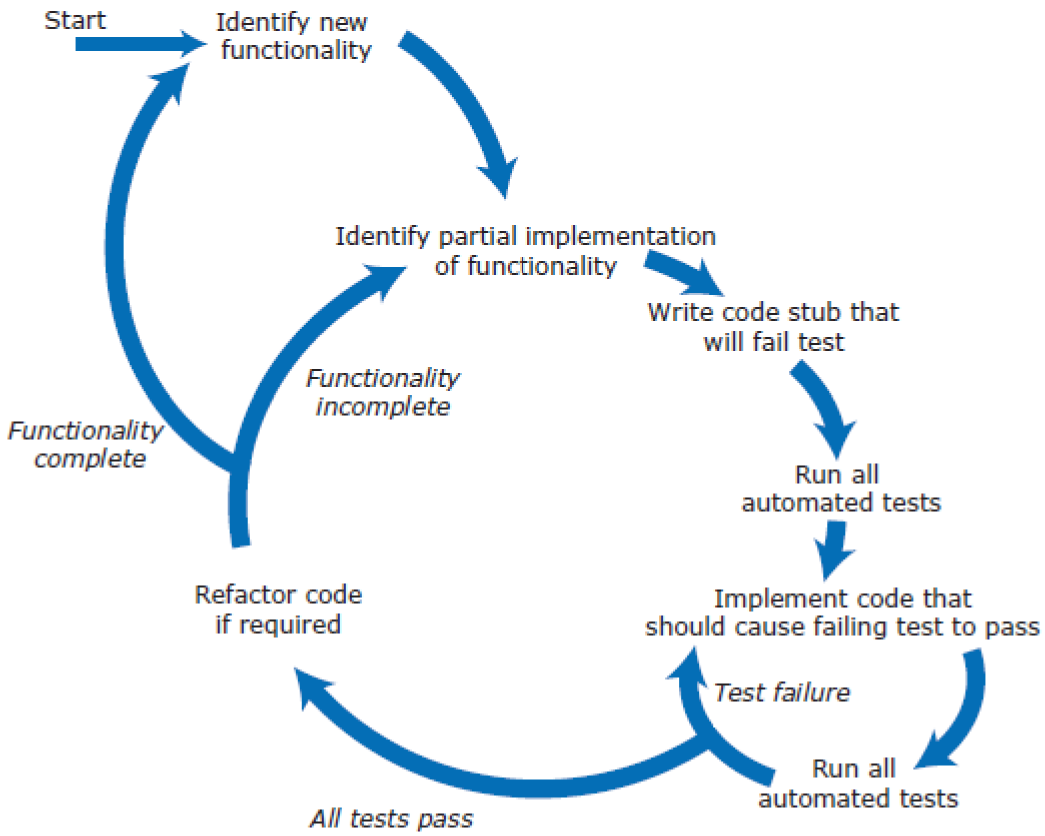


- Interaction recording tools record mouse movements and clicks, menu selections, keyboard inputs, and so on. They save the interaction session and can replay it, sending commands to the application and replicating them in the user's browser interface. These tools also provide scripting support so that you can write and execute scenarios expressed as test scripts. This is particularly useful for cross-browser testing, where you need to check that your software works in the same way with different browsers.
- Automated testing is one of the most important developments in software engineering

<u>Test-driven development</u>

- Test-driven development (TDD) is an approach to program development that is based on the general idea that we should write an executable test or tests for code that are writing before you write the code.
- TDD was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach. Figure 9.8 is a model of the test-driven development process.

Figure 9.8 Test-driven development


Start
Identify new functionality
Identify partial implementation of functionality
Write code stub that will fail test
Run all automated tests
Implement code that should cause failing test to pass
Test failure
Run all automated tests
All tests pass
Refactor code if required
Functionality incomplete
Functionality complete

- Assume that we have identified some increment of functionality to be implemented.
- Test-driven development relies on automated testing. Every time we add some functionality, we develop a new test and add it to the test suite.
- All of the tests in the test suite must pass before we move on to developing the next increment.
- Test-driven development is an approach in which executable tests are written before the code. Code is then developed to pass the tests.

■ A disadvantage of test-driven development is that programmers focus on the details of passing tests rather than considering the broader structure of their code and algorithms used.

The benefits of test-driven development are:

1. It is a systematic approach to testing in which tests are clearly linked to

section of the program code.

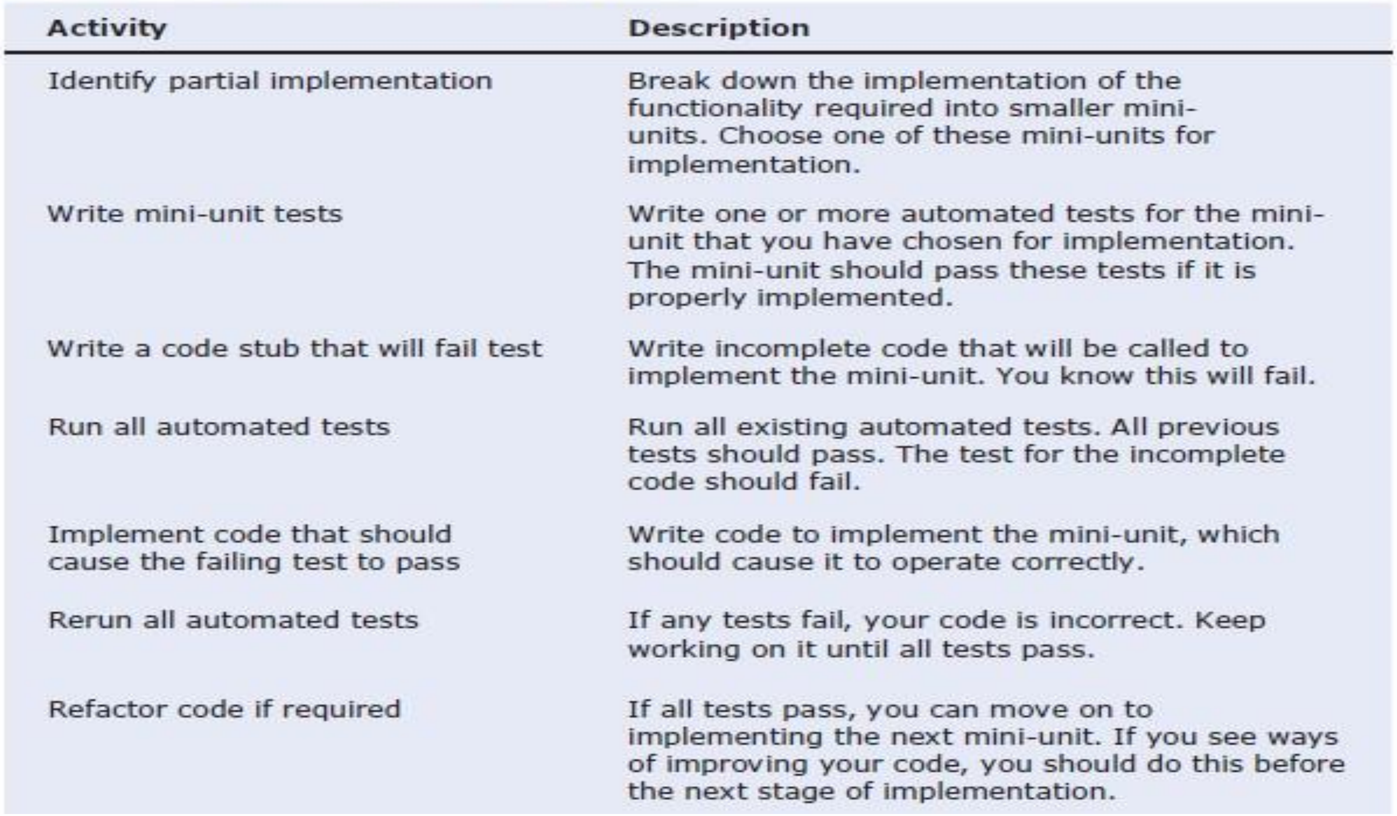
2. The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests..

3. Debugging is simplified because, when a program failure is observed, you

can immediately link this to the last increment of code that you added to the system.

4. It is argued that TDD leads to simpler code, as programmers only write code that's necessary to pass tests. They don't over engineer their code with complex features that aren't needed.

Table 9.9 Stages of test-driven development



Activity	Description
Identify partial implementation	Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.
Write mini-unit tests	Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.
Write a code stub that will fail test	Write incomplete code that will be called to implement the mini-unit. You know this will fail.
Run all automated tests	Run all existing automated tests. All previous tests should pass. The test for the incomplete code should fail.
Implement code that should cause the failing test to pass	Write code to implement the mini-unit, which should cause it to operate correctly.
Rerun all automated tests	If any tests fail, your code is incorrect. Keep working on it until all tests pass.
Refactor code if required	If all tests pass, you can move on to implementing the next mini-unit. If you see ways of improving your code, you should do this before the next stage of implementation.

DevOps and Code Management

- The ultimate goal of software product development is to release a product to customers. Traditionally, separate teams were responsible for software development, software release, and software support (Figure 1).
- The development team passed a "final" version of the software to a release team. That team then built a release version, tested it, and prepared release documentation before releasing the software to customers.

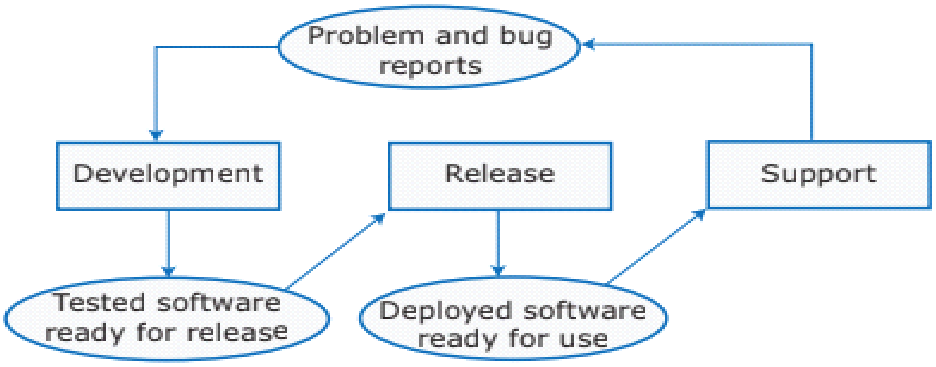

Problem and bug reports
Development
Release
Support
Tested software ready for release
Deployed software ready for use

Fig 1 Development, release, and support

- A third team provided customer support. The original development team was sometimes responsible for implementing software changes. Alternatively, the software may have been maintained by a separate maintenance team.
- In these processes, communication delays between the groups were inevitable.
- Development and operations engineers used different tools, had different skill sets, and often didn't understand the other's problems.
- Even when an urgent bug or security vulnerability was identified, it could take several days for a new release to be prepared and pushed to customers.

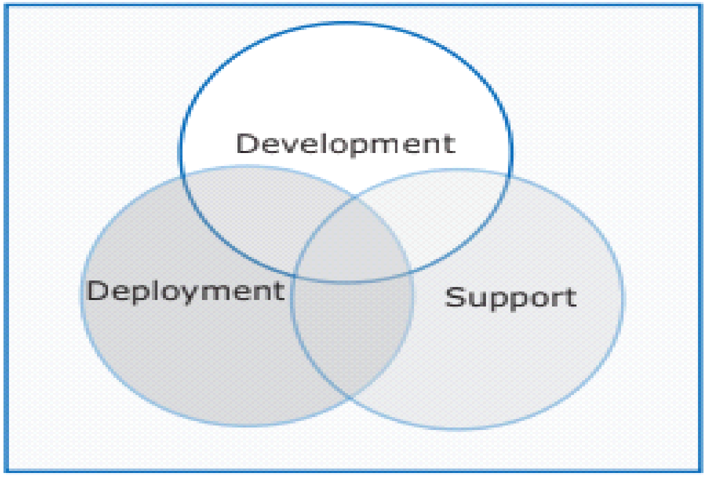
Many companies still use this traditional model of development, release, and support.

- However, more and more companies are using an alternative approach called DevOps.
- DevOps (development + operations) integrates development, deployment, and support, with a single team responsible for all of these activities (Figure 2).
- Three factors led to the development and widespread adoption of DevOps:

1. Agile software engineering reduced the development time for software, but the traditional release process introduced a bottleneck between development and deployment.

2. Amazon re-engineered their software around services and introduced an approach in which a service was both developed and supported by the same team.

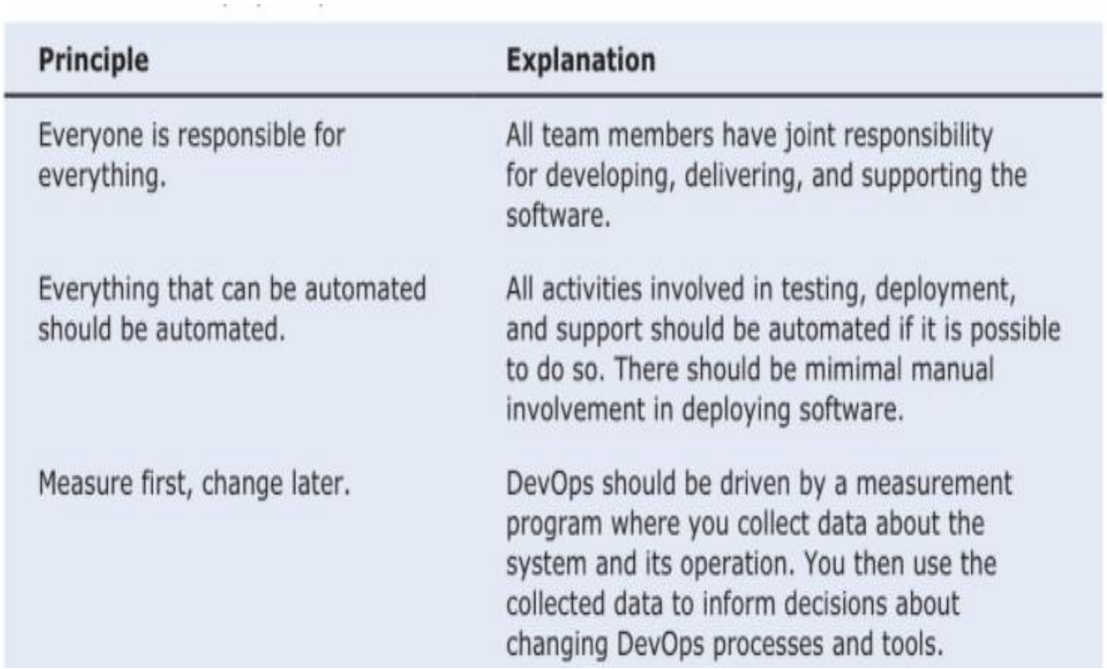
3. It became possible to release software as a service, running on a public or private cloud. Software products did not have to be released to users on physical media or downloads.



Multi-skilled DevOps team

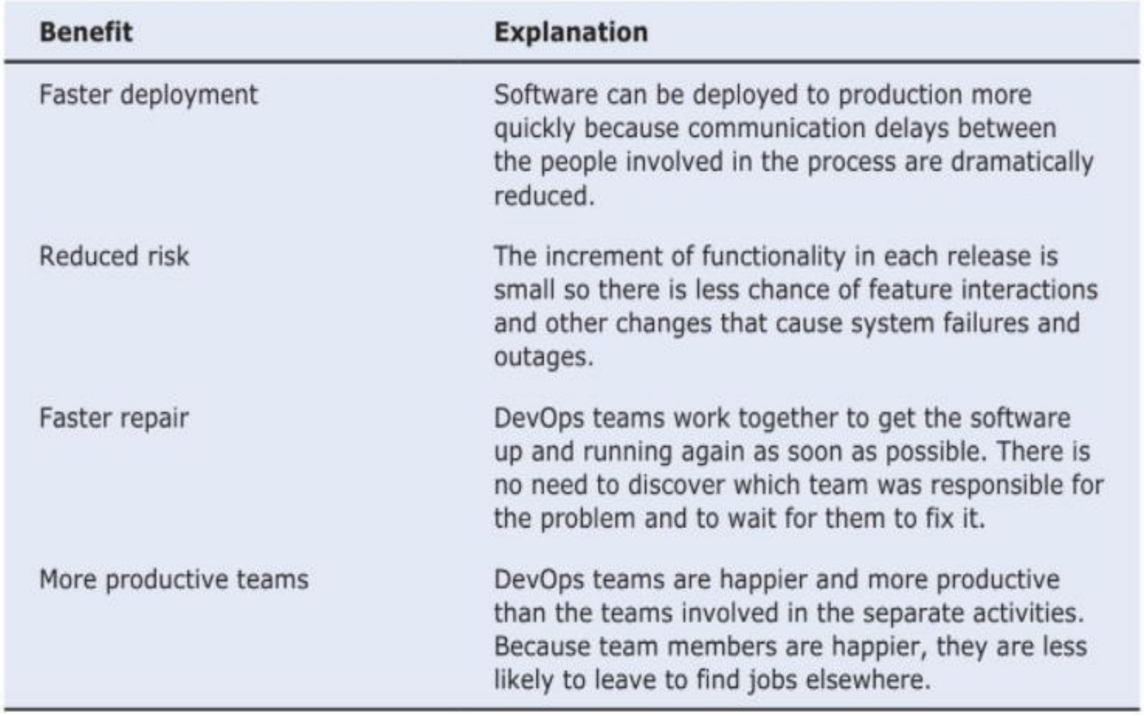
Fig 2 DevOps

<u>DevOps Principle</u>



Principle	Explanation
Everyone is responsible for everything.	All team members have joint responsibility for developing, delivering, and supporting the software.
Everything that can be automated should be automated.	All activities involved in testing, deployment, and support should be automated if it is possible to do so. There should be mimimal manual involvement in deploying software.
Measure first, change later.	DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

Table 1 DevOps principles



Benefit	Explanation
Faster deployment	Software can be deployed to production more quickly because communication delays between the people involved in the process are dramatically reduced.
Reduced risk	The increment of functionality in each release is small so there is less chance of feature interactions and other changes that cause system failures and outages.
Faster repair	DevOps teams work together to get the software up and running again as soon as possible. There is no need to discover which team was responsible for the problem and to wait for them to fix it.
More productive teams	DevOps teams are happier and more productive than the teams involved in the separate activities. Because team members are happier, they are less likely to leave to find jobs elsewhere.

Table 2 Benefits of DevOps

1: CODE MANAGEMENT

- DevOps depends on the source code management system that is used by the entire team.
- Code management is a set of software-supported practices used to manage an evolving codebase.
- We need code management to ensure that changes made by different developers do not interfere with each other and to create different product versions.
- Code management tools make it easy to create an executable product from its source code files and to run automated tests on that product.
- Source code management, combined with automated system building, is critical for professional software engineering.
- In companies that use DevOps, a modern code management system is a fundamental requirement for “auto- mating everything.” Not only does it store the project code that is ultimately deployed, but it also stores all other information that is used in DevOps processes.
- DevOps automation and measurement tools all interact with the code management system (Figure 3).

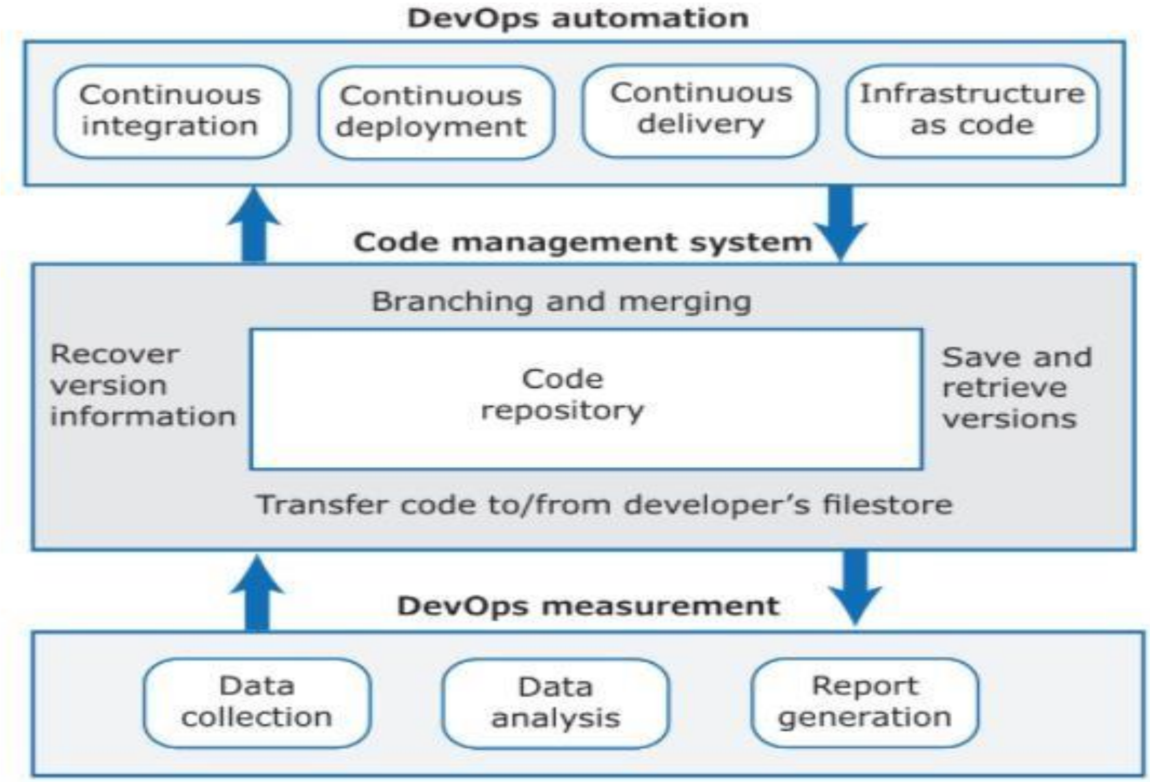


Fig 3: Code management and DevOps

1.1 Fundamentals of source code management

Source code management systems are designed to manage an evolving project codebase to allow different versions of components and entire systems to be stored and retrieved.

Developers can work in parallel without interfering with each other and they can integrate their work with that from other developers.

The code management system provides a set of features that support four general areas:

1. Code transfer Developers take code into their personal file store to work on it; then they return it to the shared code management system.

2. Version storage and retrieval Files may be stored in several different versions, and specific versions of these files can be retrieved.

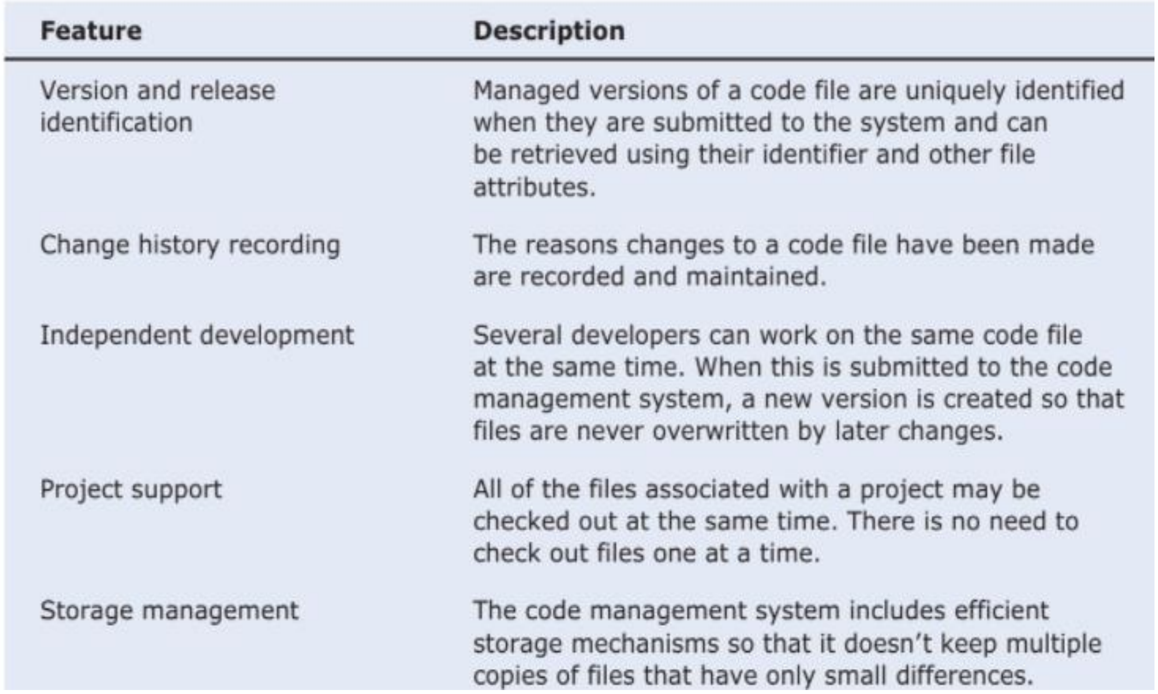
3. Merging and branching Parallel development branches may be created for concurrent working. Changes made by developers in different branches may be merged.

4. Version information Information about the different versions maintained in the system may be stored and retrieved.

All source code management systems have the general form shown in Figure 3, with a shared repository and a set of features to manage the files in that repository:

1. All source code files and file versions are stored in the repository, as are other artifacts such as configuration files, build scripts, shared libraries, and versions of tools used. The repository includes a database of information about the stored files, such as version information, information about who has changed the files, what changes were made at what times and so on.

2. The source code management features transfer files to and from the repository and update the information about the different versions of files and their relationships. Specific versions of files and information about these versions can always be retrieved from the repository.



Feature	Description
Version and release identification	Managed versions of a code file are uniquely identified when they are submitted to the system and can be retrieved using their identifier and other file attributes.
Change history recording	The reasons changes to a code file have been made are recorded and maintained.
Independent development	Several developers can work on the same code file at the same time. When this is submitted to the code management system, a new version is created so that files are never overwritten by later changes.
Project support	All of the files associated with a project may be checked out at the same time. There is no need to check out files one at a time.
Storage management	The code management system includes efficient storage mechanisms so that it doesn't keep multiple copies of files that have only small differences.

Table 4 Features of source code management systems

- In 2005, Linus Torvalds, the developer of Linux, revolutionized source code management by developing a distributed version control system (DVCS) called Git to manage the code of the Linux kernel.
- Git was geared to supporting large-scale open-source development.
- It took advantage of the fact that storage costs had fallen to such an extent that most users did not have to be concerned with local storage management.
- Instead of only keeping the copies of the files that users are working on, Git maintains a clone of the repository on every user's computer (Figure 5).
- A fundamental concept in Git is the “master branch,” which is the current master version of the software that the team is working on.
- we create new versions by creating a new branch, In Figure 5, we can see that two branches have been created in addition to the master branch.
- When users request a repository clone, they get a copy of the master branch that they can work on independently.

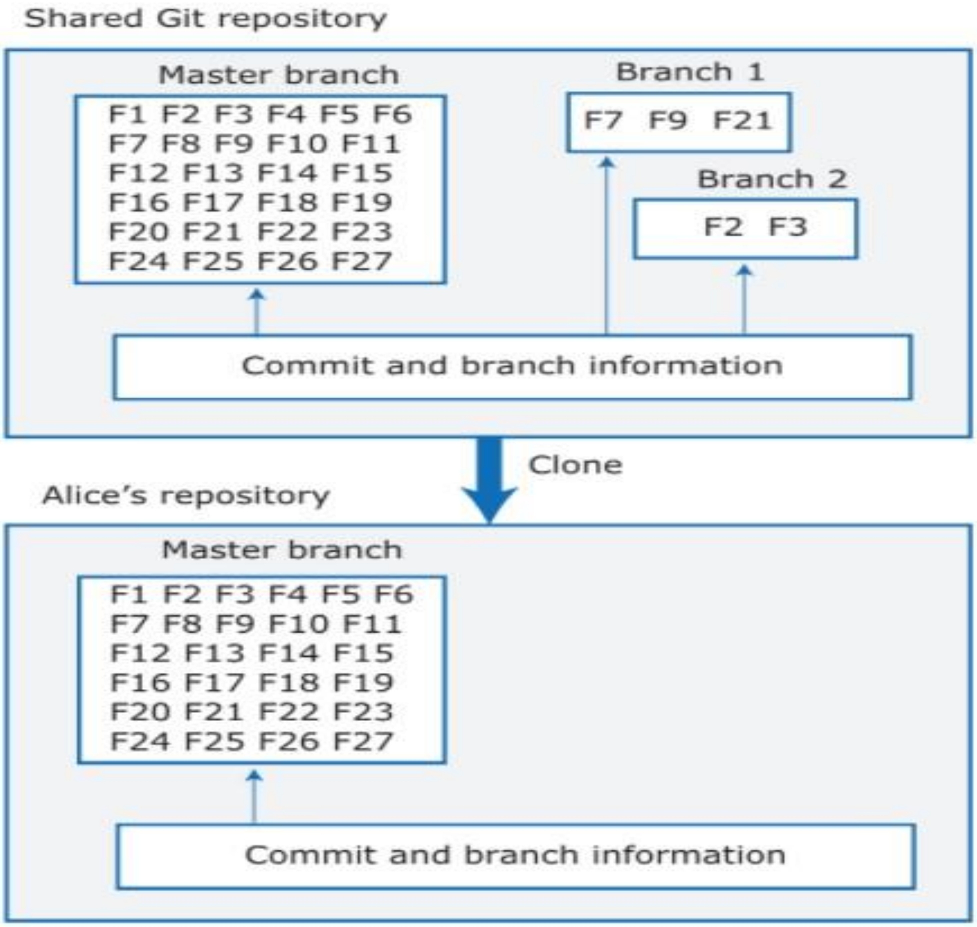


Figure 5 Repository cloning in Git

Git and other distributed code management systems have several advantages over centralized systems:

1. *Resilience* Everyone working on a project has their own copy of the repository. If the shared repository is damaged or subjected to a cyber attack, work can continue, and the clones can be used to restore the shared repository. People can work offline if they don't have a network connection.

2. Speed Committing changes to the repository is a fast, local operation and does not need data to be transferred over the network.

3. Flexibility Local experimentation is much simpler. Developers can safely try different approaches without exposing their experiments to other project members. With a centralized system, this may only be possible by working outside the code management system.

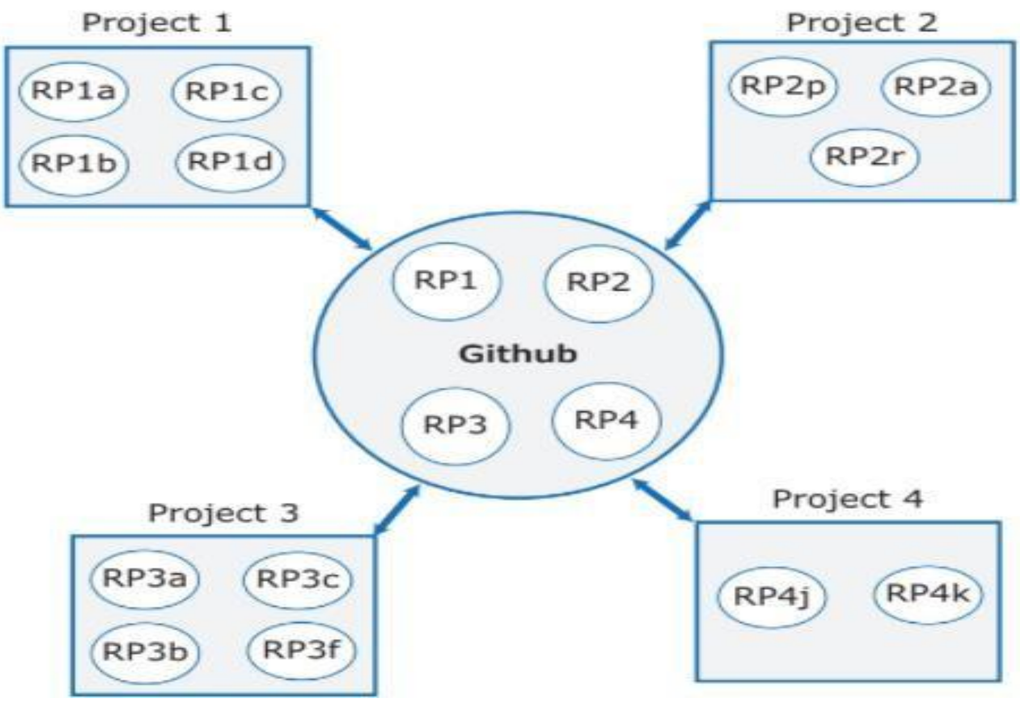


Figure 6 Git repositories

- Most software product companies now use Git for code management.
- For teamwork, Git is organized around the notion of a shared project repository and private clones of that repository held on each developer's computer (Figure 6).
- A company may use its own server to run the project repository. However, many companies and individual developers use an external Git repository provider.
- Several Git repository hosting companies, such as Github and Gitlab, host thousands of repositories on the cloud.

Figure 6 shows four project repositories on Github, RP1–RP4. RP1 is the repository for project 1, RP2 is the repository for project 2, and so on. Each of the developers on each project is identified by a letter (a, b, c, etc.) and has an individual copy of the project repository.

Developers may work on more than one project at a time, so they may have copies of several Git repositories on their computer.

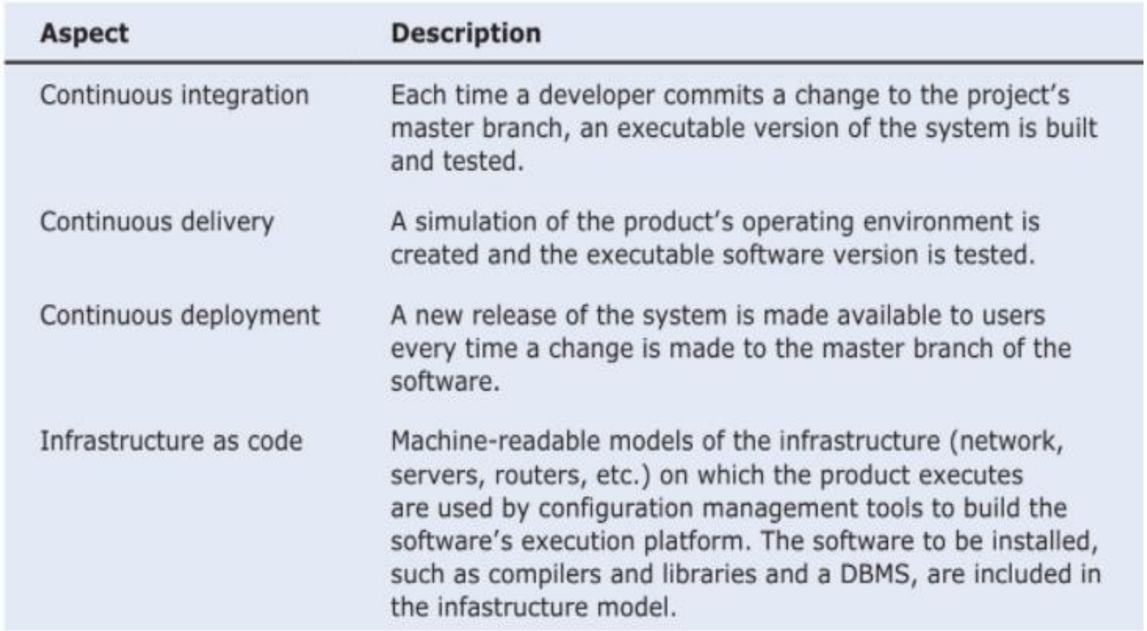
For example, developer a works on Project 1, Project 2, and Project 3, so has clones of RP1, RP2, and RP3.

2: DevOps automation

- Historically, the processes of integrating a system from independently developed parts, deploying that system in a realistic testing environment, and releasing it were time consuming and expensive.
- By using DevOps with automated support, however, we can dramatically reduce the time and costs for integration, deployment, and delivery.

- “Everything that can be should be automated” is a fundamental principle of DevOps.
- In addition to reducing the costs and time required for integration, deployment, and delivery, automation makes these processes more reliable and reproducible.

Figure 3 showed the four aspects of DevOps automation



Aspect	Description
Continuous integration	Each time a developer commits a change to the project’s master branch, an executable version of the system is built and tested.
Continuous delivery	A simulation of the product’s operating environment is created and the executable software version is tested.
Continuous deployment	A new release of the system is made available to users every time a change is made to the master branch of the software.
Infrastructure as code	Machine-readable models of the infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software’s execution platform. The software to be installed, such as compilers and libraries and a DBMS, are included in the infastructure model.

Table Aspects of DevOps automation

2.1 Continuous integration

- System integration (system building) is the process of gathering all of the elements required in a working system, moving them into the right directories, and putting them together to create an operational system. This involves more than compiling the system.
- Continuous integration (CI) means creating an executable version of a software system whenever a change is made to the repository. The CI tool is triggered when a file is pushed to the repo. It builds the system and runs tests on your development computer or project integration server

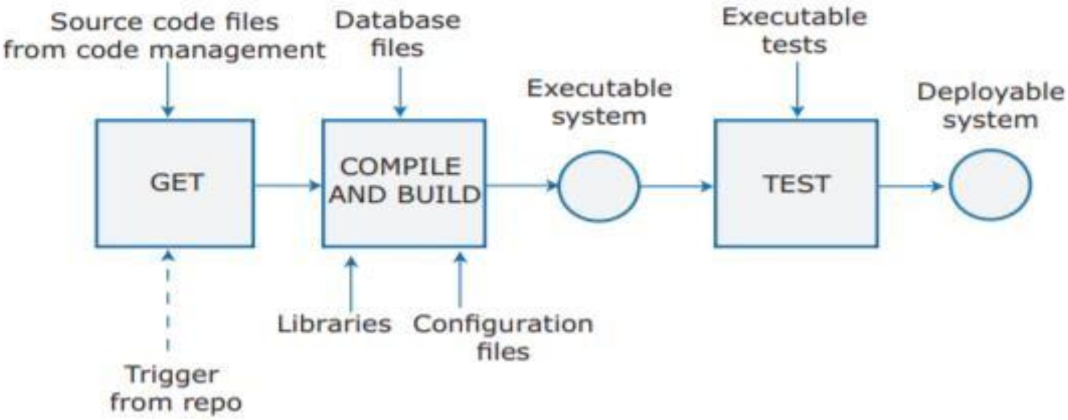


Figure 9 Continuous integration

- Continuous integration simply means that an integrated version of the system is created and tested every time a change is pushed to the system's shared code repository.
- On completion of the push operation, the repository sends a message to an integration server to build a new version of the product (Figure 9).
- The squares in Figure 9 are the elements of a continuous integration pipeline that is triggered by a repository notification that a change has been made to the master branch of the system.

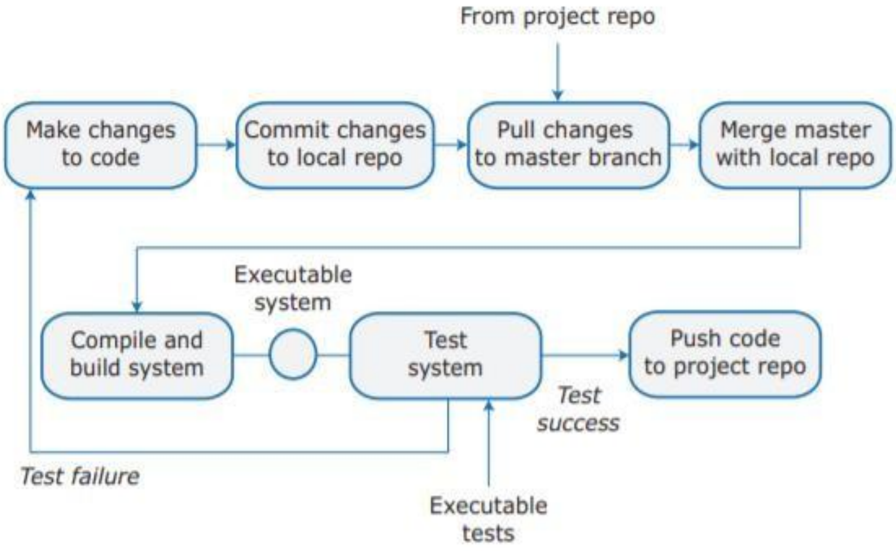


Figure 10 Local integration

- In a continuous integration environment, developers have to make sure that they don’t “break the build.” Breaking the build means pushing code to the project repository, which when integrated, causes some of the system tests to fail. This holds up other developers. If this happens ,our priority is to discover and fix the problem so that normal development can continue.
- To avoid breaking the build, we should always adopt an “integrate twice” approach to system integration. we should integrate and test on our own computer before pushing code to the project repository to trigger the integration server (Figure10).
- The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system. If we make a small change and some system tests then fail, the problem almost certainly lies in the new code that we have pushed to the project repo. we can focus on this code to find the bug that’s causing the problem.
- If we continuously integrate, then a working system is always available to the whole team. This can be used to test ideas and to demonstrate the features of the system to management and customers. Furthermore, continuous integration creates a “quality culture” in a development team. Team members want to avoid the stigma and disruption of breaking the build. They are likely to check their work carefully before pushing it to the project repo.

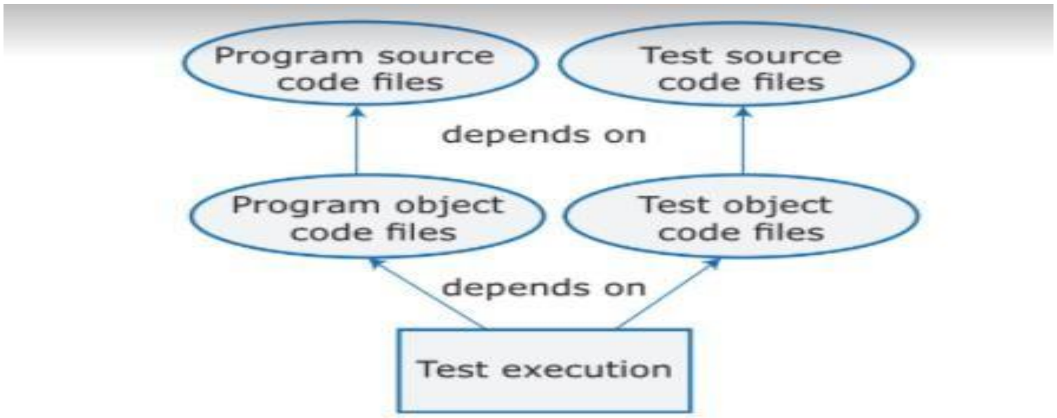


Figure 11 A dependency model

- Continuous integration is effective only if the integration process is fast and developers do not have to wait for the results of their tests of the integrated system.
- However, some activities in the build process, such as populating a database or compiling hundreds of system files, are inherently slow. It is therefore essential to have an automated build process that minimizes the time spent on these activities.
- To understand incremental system building, you need to understand the concept of dependencies.
- Figure 11 is a dependency model that shows the dependencies for test execution.
- An upward-pointing arrow means “depends on” and shows the information required to complete the task shown in the rectangle at the base of the model. Figure 11 therefore shows that running a set of system tests depends on the existence of executable object code for both the program being tested and the system tests.

- In turn, these depend on the source code for the system and the tests that are compiled to create the object code.
- The first time we integrate a system, the incremental build system compiles all the source code files and executable test files. It creates their object code equivalents, and the executable tests are run. Subsequently, however, object code files are created only for new and modified tests and for source code files that have been modified.

2.2 Continuous delivery and deployment

- Continuous delivery means that, after making changes to a system, we ensure that the changed system is ready for delivery to customers.
- This means that you have to test it in a production environment to make sure that environmental factors do not cause system failures or slow down its performance.
- Continuous delivery does not mean that the software will necessarily be released immediately to users for deployment.

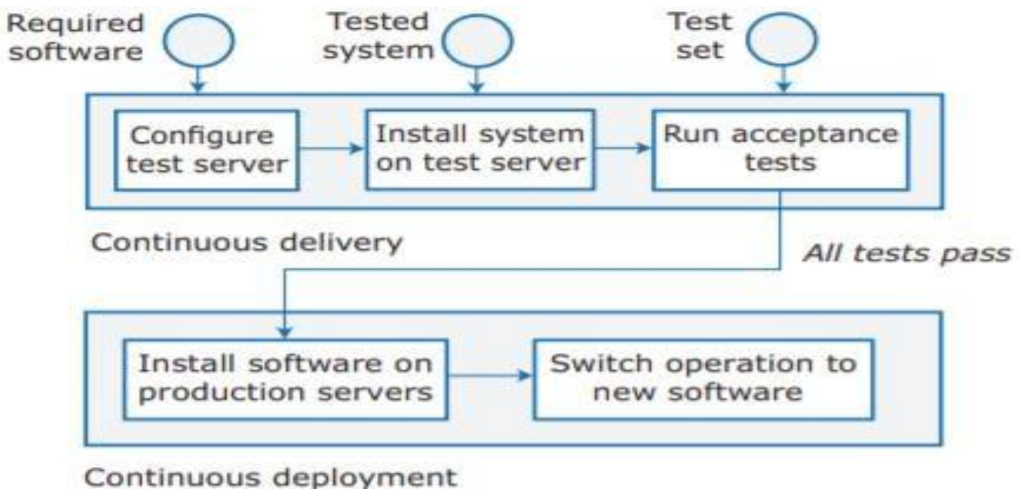
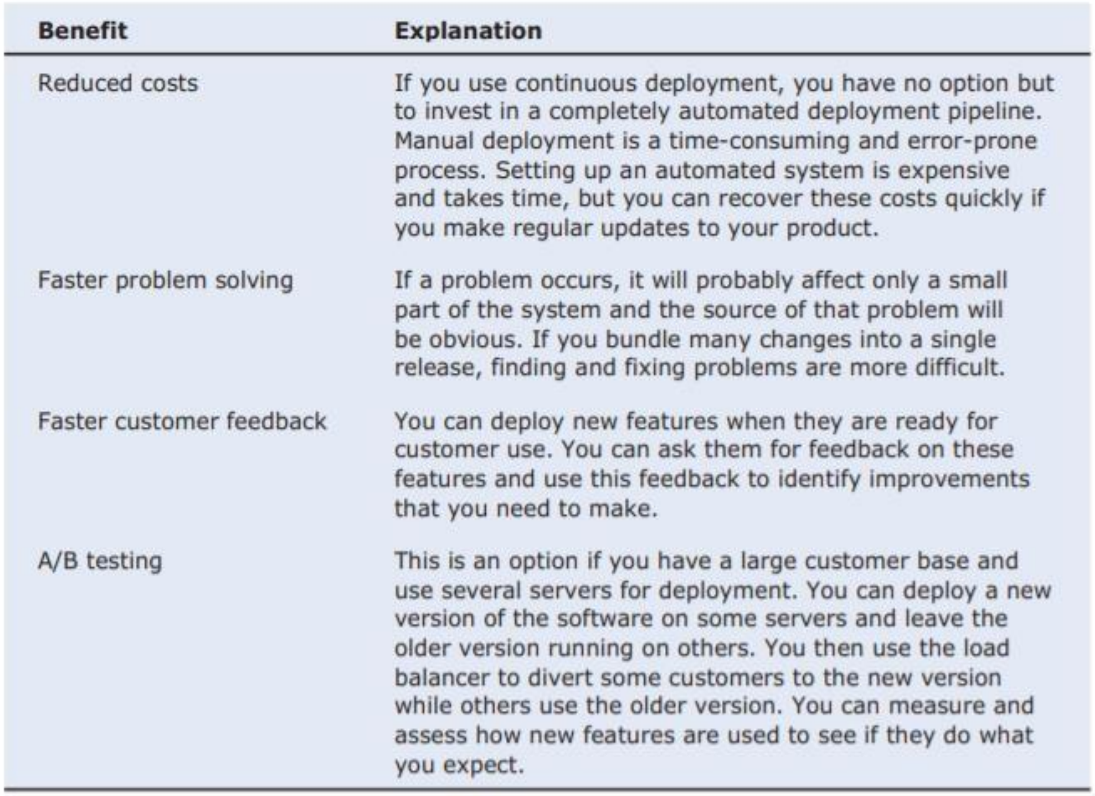


Figure 12 Continuous delivery and deployment

- Figure 12 illustrates a summarized version of this deployment pipeline, showing the stages involved in continuous delivery and deployment.
- After initial integration testing, a staged test environment is created. This is a replica of the actual production environment in which the system will run.
- The system acceptance tests, which include functionality, load, and performance tests, are then run to check that the software works as expected.



Benefit	Explanation
Reduced costs	If you use continuous deployment, you have no option but to invest in a completely automated deployment pipeline. Manual deployment is a time-consuming and error-prone process. Setting up an automated system is expensive and takes time, but you can recover these costs quickly if you make regular updates to your product.
Faster problem solving	If a problem occurs, it will probably affect only a small part of the system and the source of that problem will be obvious. If you bundle many changes into a single release, finding and fixing problems are more difficult.
Faster customer feedback	You can deploy new features when they are ready for customer use. You can ask them for feedback on these features and use this feedback to identify improvements that you need to make.
A/B testing	This is an option if you have a large customer base and use several servers for deployment. You can deploy a new version of the software on some servers and leave the older version running on others. You then use the load balancer to divert some customers to the new version while others use the older version. You can measure and assess how new features are used to see if they do what you expect.

Table 6 Benefits of continuous deployment

2.3 Infrastructure as code

- Rather than manually updating the software on a company's servers, the process can be automated using a model of the infrastructure written in a machine-processable language.
- Configuration management (CM) tools, such as Puppet and Chef, can automatically install software and services on servers according to the infrastructure definition. The CM tool accesses a master copy of the software to be installed and pushes this to the servers being provisioned (Figure 13).
- When changes have to be made, the infrastructure model is updated and the CM tool makes the change to all servers.
- Defining our software infrastructure as code is obviously relevant to products that are delivered as services.
- The product provider has to manage the infrastructure of their services on the cloud. However, it is also relevant if software is delivered through downloads.

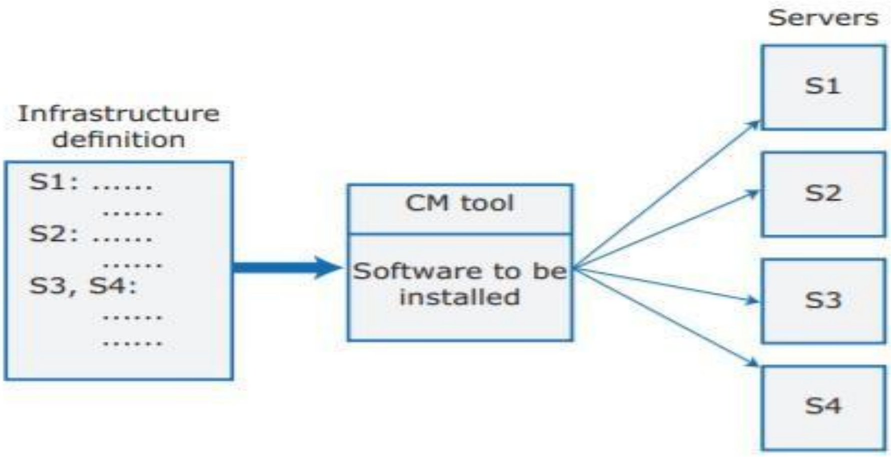


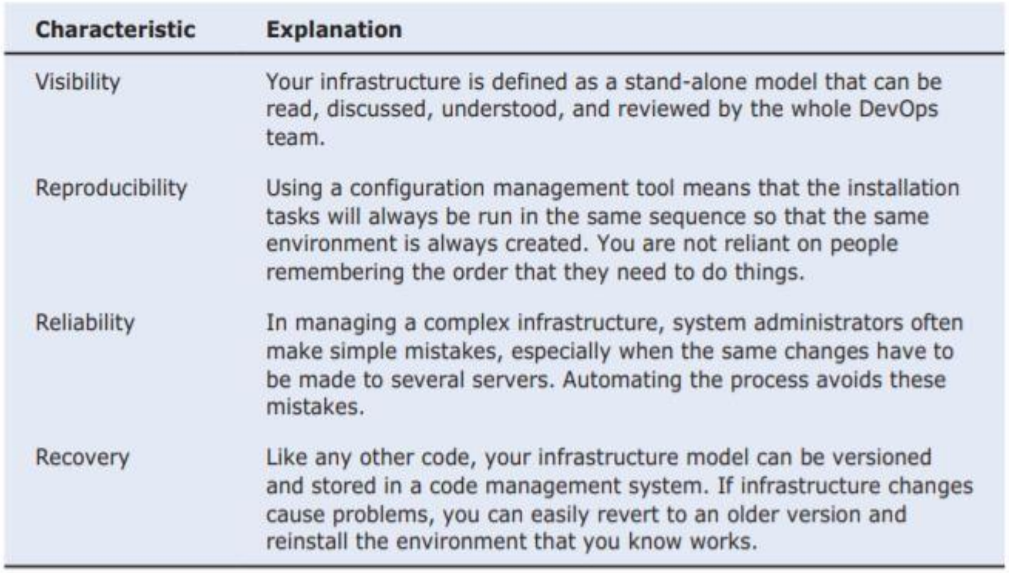
Figure 13 Infrastructure as code

Defining our infrastructure as code and using a configuration management system solve two key problems of continuous deployment:

1. our testing environment must be exactly the same as your deployment environment. If you change the deployment environment, you have to mirror those changes in your testing environment.

2. When we change a service, we have to be able to roll that change out to all of our servers quickly and reliably. If there is a bug in our changed code that affects the system's reliability, we have to be able to seamlessly roll back to the older system.

The business benefits of defining our infrastructure as code are lower costs of system management and lower risks of unexpected problems arising when infrastructure changes are implemented. These benefits stem from four fundamental characteristics of infrastructure as code, shown in Table 7



Characteristic	Explanation
Visibility	Your infrastructure is defined as a stand-alone model that can be read, discussed, understood, and reviewed by the whole DevOps team.
Reproducibility	Using a configuration management tool means that the installation tasks will always be run in the same sequence so that the same environment is always created. You are not reliant on people remembering the order that they need to do things.
Reliability	In managing a complex infrastructure, system administrators often make simple mistakes, especially when the same changes have to be made to several servers. Automating the process avoids these mistakes.
Recovery	Like any other code, your infrastructure model can be versioned and stored in a code management system. If infrastructure changes cause problems, you can easily revert to an older version and reinstall the environment that you know works.

Table 10.7 Characteristics of infrastructure as code

3 DevOps measurement

After you have adopted DevOps, you should try to continuously improve your DevOps process to achieve faster deployment of better-quality software. This means you need to have a measurement program in place in which you collect and analyze product and process data. By making measurements over time, you can judge whether or not you have an effective and improving process

Measurements about software development and use fall into four categories:

1. **Process measurements** :collect and analyze data about your development, testing, and deployment processes.

2. **Service measurements** ;collect and analyze data about the software's performance, reliability, and acceptability to customers.

3. **Usage measurements** : collect and analyze data about how customers use your product.

4. **Business success measurements** : collect and analyze data about how your product contributes to the overall success of the business.