Software Engineering and Project Management

(MVJ22CS51)

MODULE - 3

Software Testing*: A Strategic Approach to Software Testing, Strategic Issues, Test Strategies for Conventional Software, Test Strategies for Object -Oriented Software, Validation Testing, System Testing, The Art of Debugging.*

Agile Methodology & DevOps*: Before Agile – Waterfall, Agile Development.*

Self-Learning Section*:*

What is DevOps? DevOps Importance and Benefits, DevOps Principles and Practices, 7 C's of DevOps Lifecycle for Business Agility, DevOps and Continuous Testing, How to Choose Right DevOps Tools?, Challenges with DevOps Implementation.

SOFTWARE TESTING

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

A STRATEGIC APPROACH TO SOFTWARE TESTING

Generic characteristics of Software Testing:

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

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements.

A strategy should provide guidance for the practitioner and a set of milestones for the manager.

1. 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?"

Verification and validation includes a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing.

2. Organizing for Software Testing:

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed.

The developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture.

Only after the software architecture is complete does an independent test group become involved.

The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built.

Independent testing removes the conflict of interest that may otherwise be present. After all, ITG personnel are paid to find errors.

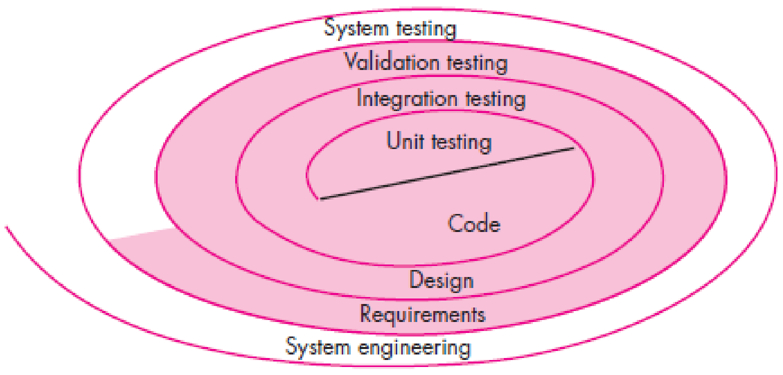
The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted.

While testing is conducted, the developer must be available to correct errors that are uncovered.

The ITG is part of the software development project team in the sense that it becomes involved during analysis and design and stays involved (planning and specifying test procedures) throughout a large project.

3. *Software Testing Strategy—The Big Picture:*
   A Software Testing Strategy for Conventional Software Architectures

FIGURE 17.1
Testing strategy



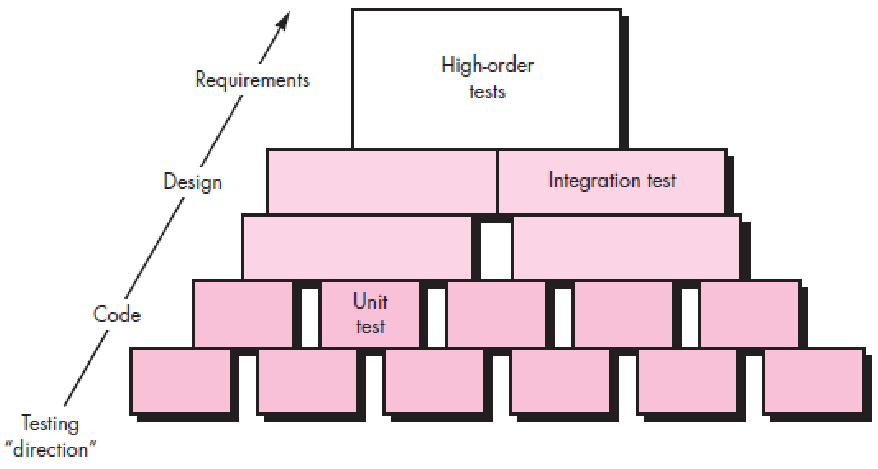
A strategy for software testing may also be viewed in the context of the spiral.

- 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 in a clockwise direction along streamlines that broaden the scope of testing with each turn.

A strategy for software testing may also be viewed from a procedural point of view.

Testing within the context of software engineering is actually a series of four steps that are implemented sequentially.

FIGURE 17.2
Software testing steps



1. Tests focus on 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.
2. 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. 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.
3. 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 informational, functional, behavioral, and performance requirements.
4. 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.

4. *A Software Testing Strategy for Object-Oriented Architectures:*

Testing must include error discovery techniques (e.g. formal technical reviews) that are applied to analysis and design models.

The completeness and consistency of object-oriented representation must be assessed as they are built.

Here, we begin with testing in the small and work outward toward testing in the large.

However, our focus when testing in the small changes from an individual module (conventional view) to a class that encompasses attributes and operations and implies communication and collaboration.

As classes are integrated into an object-oriented architecture, a series of regression tests are run to uncover errors due to communication and collaboration between classes.

Finally the system as a whole is tested to ensure that errors in requirements are uncovered.

5. *Criteria for Completion of Testing:*

A question arises every time: When are we done testing—how do we know that we've tested enough? When testing is carried out for users from targeted population (who will use the project).

The *cleanroom software engineering* approach i.e. statistical use techniques that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population.

When there are no errors, then completion of testing has occurred.

STRATEGIC ISSUES

A systematic strategy for software testing will succeed when software testers:

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes "rapid cycle testing." The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.
- Build "robust" software that is designed to test itself.
- Use effective technical reviews as a filter prior to testing.
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

There are many strategies that can be used to test software.

A testing strategy that is chosen by most software teams falls between the two extremes.

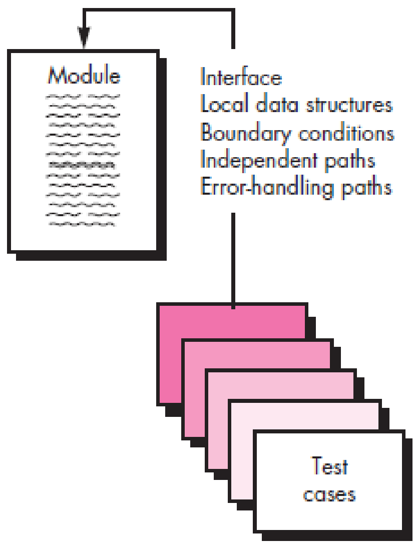
1. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units.
2. Culminating with tests that exercise the constructed system.

1. ***Unit Testing***

Unit testing focuses verification effort on the smallest unit of software design—the software component or module.

Unit-test considerations:

FIGURE 17.3
Unit test



The above figure is described as:

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

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.

Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. This test is known as *Bounding Testing.* Software often fails at its boundaries.

A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. This approach is known as 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

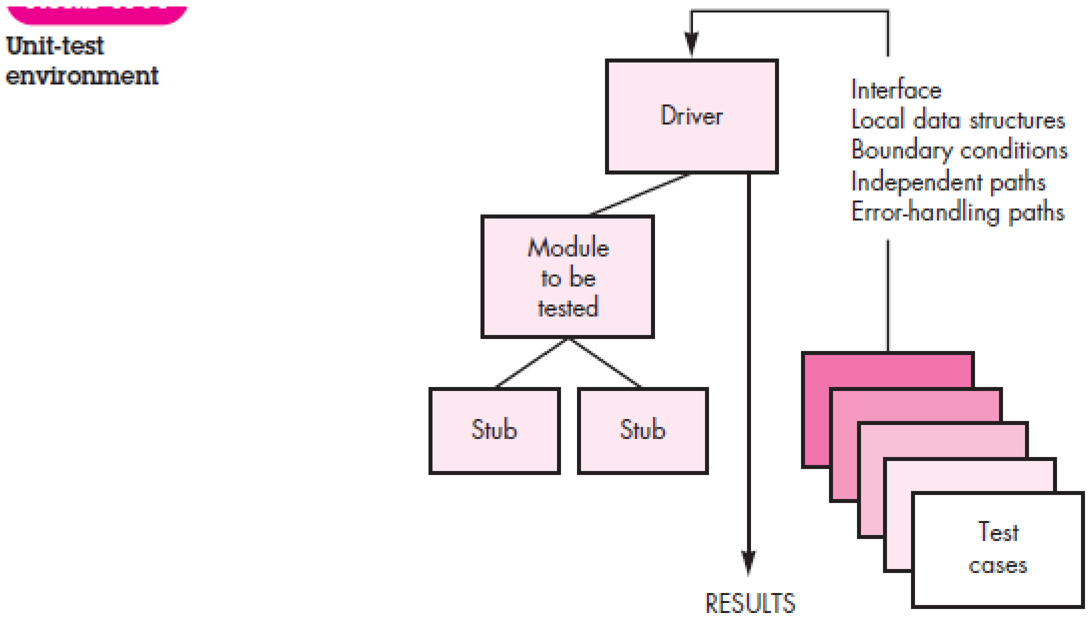
(4) exception-condition processing is incorrect, or

(5) error description does not provide enough information to assist in the location of the cause of the error.

Unit-test procedures:

The design of unit tests can occur before coding begins or after source code has been generated.

A component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.



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 written (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).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

2. ***Integration Testing***

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.

There is often a tendency to attempt non-incremental integration; that is, to construct the program using a “big bang” approach. All components are combined in advance. The entire program is tested as a whole.

Two incremental integration strategies are:

(1) Top-down integration
(2) Bottom-up integration

(1) *Top-down integration:*

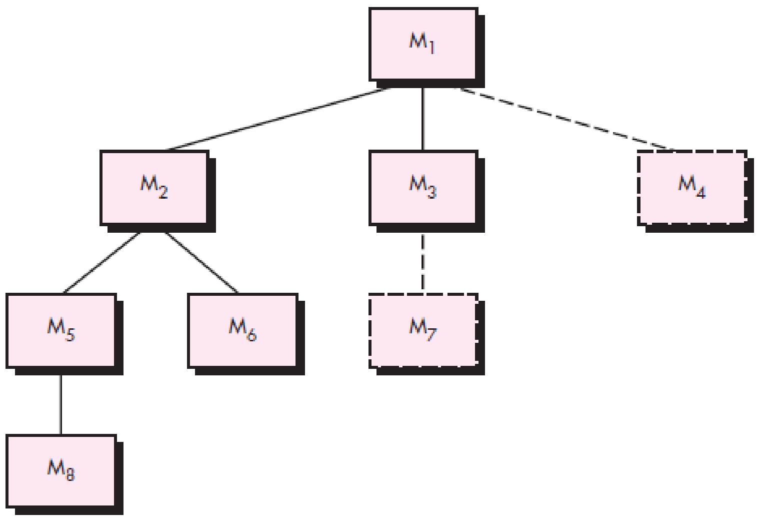
Top-down integration testing is an incremental approach to construction of the software architecture.

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.

FIGURE 17.5

Top-down integration



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 M_1, M_2 , M_5 would be integrated first. Next, M_8 or (if necessary for proper functioning of M_2) M_6 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.

For example, components M_2, M_3, and M_4 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.
1. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
2. Tests are conducted as each component is integrated.
3. On completion of each set of tests, another stub is replaced with the real component.

4. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

(2) *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.

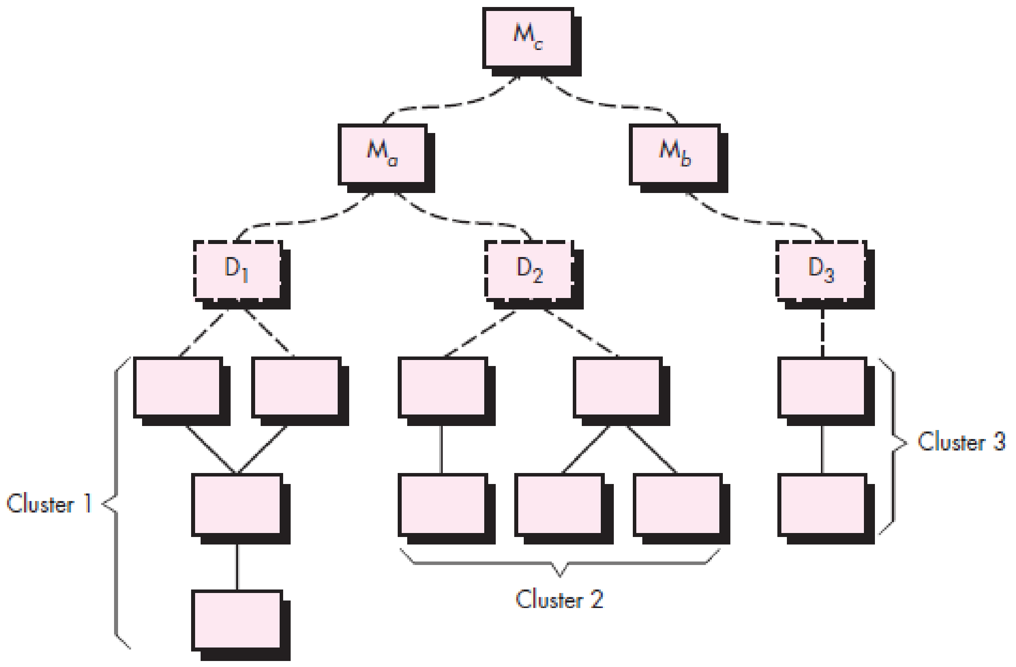
2. A driver (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

FIGURE 17.6

Bottom-up integration



Integration follows the pattern as in above 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 Ma. Drivers D_1 and D_2 are removed and the clusters are interfaced directly to M_a. Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b. 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. 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.

3. ***Regression Testing:***

Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.

Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing 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.

4. ***Smoke testing:***

Smoke testing is an integration testing approach that is commonly used when product software is developed.

Smoke-testing approach has 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 “showstopper” 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.

McConnell describes the smoke test in the following manner:

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

Smoke testing provides a number of benefits when it is applied on complex, time-critical software projects:

- Integration risk is minimized.
- The quality of the end product is improved.
- Error diagnosis and correction are simplified.
- Progress is easier to assess.

5. ***Strategic options:***

The major disadvantage of the top-down approach is the need for stubs and the attendant testing difficulties that can be associated with them. Problems associated with stubs may be offset by the advantage of testing major control functions early.

The major disadvantage of bottom-up integration is that "the program as an entity does not exist until the last module is added". This drawback is tempered by easier test case design and a lack of stubs.

A combined approach (sometimes called *sandwich testing*) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.

As integration testing is conducted, the tester should identify critical modules. A critical module has one or more of the following characteristics:

(1) addresses several software requirements

(2) has a high level of control (resides relatively high in the program structure)

(3) is complex or error prone or

(4) has definite performance requirements.

Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.

6. ***Integration test work products:***

An overall plan for integration of the software and a description of specific tests is documented in a *Test Specification*.

This work product incorporates a test plan and a test procedure and becomes part of the software configuration.

Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software.

For example, integration testing for the SafeHome security system the following can be test phases:

- User interaction (command input and output, display representation, error processing and representation)
- Sensor processing (acquisition of sensor output, determination of sensor conditions, actions required as a consequence of conditions)
- Communications functions (ability to communicate with central monitoring station)
- Alarm processing (tests of software actions that occur when an alarm is encountered)

The following criteria and corresponding tests are applied for all test phases:

Interface integrity - Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.

Functional validity - Tests designed to uncover functional errors are conducted.

Information content - Tests designed to uncover errors associated with local or global data structures are conducted.

Performance - Tests designed to verify performance bounds established during software design are conducted.

<u>TEST STRATEGIES FOR OBJECT-ORIENTED SOFTWARE</u>

1. *<u>Unit Testing in the OO Context</u>*

An encapsulated class is usually the focus of unit testing.

Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data.

However, operations (methods) within the class are the smallest testable units. Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change.

2. *<u>Integration Testing in the OO Context</u>*

There are two different strategies for integration testing of OO systems:

1. *Thread-based testing* - integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually.
2. Regression testing is applied to ensure that no side effects occur. The second integration approach, use-based testing, begins the construction of the system by testing those classes (called independent classes) that use very few (if any) server classes.
3. After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test **cases that attempt to uncover errors in the collaborations.**

<u>VALIDATION TESTING</u>

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 the validation or system level, the distinction between conventional software, object-oriented software, disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

Validation Testing approaches are:

1. *<u>Validation-Test Criteria</u>*

After each validation test case has been conducted, one of two possible conditions exists:

(1) The function or performance characteristic conforms to specification and is accepted or

(2) a deviation from specification is uncovered and a deficiency list is created.

2. *<u>Configuration Review</u>*

An important element of the validation process is a configuration review.

The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities.

The configuration review, sometimes called an audit.

3. *Alpha and Beta Testing*

Alpha and Beta testing to uncover errors that only the end user seems able to find.

The alpha test is conducted at the developer's site by a representative group of end users.

The beta test is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present.

A variation on beta testing, called customer acceptance testing, is sometimes performed when custom software is delivered to a customer under contract.

SYSTEM TESTING

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.

Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

1. *Recovery Testing*

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), einitialization, 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.

2. *Security Testing*

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

3. *Stress Testing*

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

4. *Performance Testing*

Performance testing is designed to test the run-time performance of software within the context of an integrated system.

5. *Deployment Testing*

Deployment testing, is also known as 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.

THE ART OF DEBUGGING

Debugging occurs as a consequence of successful testing. When a test case uncovers an error, debugging is the process that results in the removal of the error.

1. *The Debugging Process*

Debugging is not testing but often occurs as a consequence of testing.

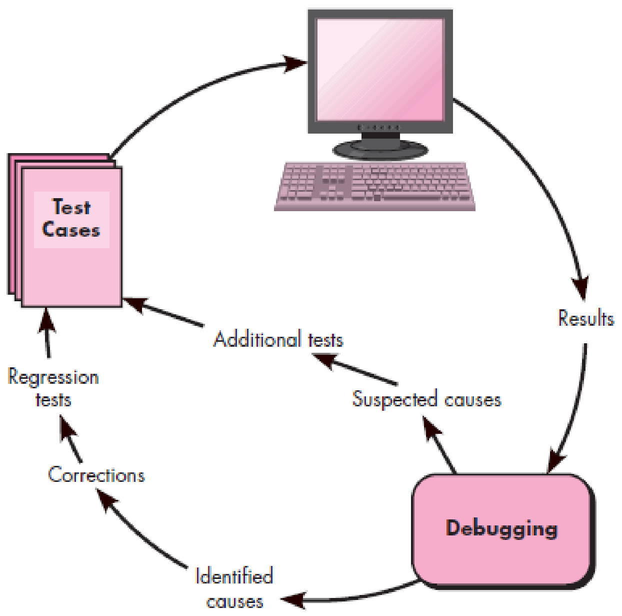
The debugging process will usually have one of two outcomes:

1. the cause will be found and corrected or
2. the cause will not be found.

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 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.

FIGURE 17.7 The debugging process



2. *Psychological Considerations*

Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it and others aren't.

3. *Debugging Strategies*

Regardless of the approach that is taken, 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.

Three debugging strategies have been proposed:

(1) brute force,

(2) backtracking, and

(3) cause elimination.

Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

(1) 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.

The brute force category of debugging is probably the most common and least efficient method for isolating the cause of a software error. You apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements

(2) Automated debugging

Each of these debugging approaches can be supplemented with debugging tools that can provide you with semi-automated support as debugging strategies are attempted.

(3) The people factor.

Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people! A fresh viewpoint, unclouded by hours of frustration, can do wonders.

4. *Correcting the Error*

Van Vleck suggests three simple questions that you should ask before making the "correction" that removes the cause of a bug:

(1) Is the cause of the bug reproduced in another part.
(2) What "next bug" might be introduced by the fix I'm about to make?
(3) What could we have done to prevent this bug in the first place?

AGILE METHODOLOGY

BEFORE AGILE – WATERFALL, AGILE DEVELOPMENT

BEFORE AGILE – WATERFALL

Waterfall methodology was the standard for software development.

Using the waterfall method required a ton of documentation up front, before any coding started. Usually, the process started with a business analyst writing a business requirements document that captured what the business needed from the application. These documents were long and detailed, containing everything from overall strategy to comprehensive functional specifications and visual user interface designs.

Technologists used the business requirements document to develop a technical requirements document. This document defined the application's architecture, data structures, object-oriented functional designs, user interfaces, and other non-functional requirements.

Once the business and technical requirements documents were complete, developers would kick off coding, then integration, and finally testing. All of this had to be done before an application was deemed production-ready. The whole process could easily take a couple of years.

AGILE DEVELOPMENT

Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity.

4 Values of Agile Software Development

Here are 4 core values of Agile Software Development:

(1) People and interactions above procedures and equipment.
(2) Working software is preferred over thorough documentation.
(3) Collaboration with the customer during contract negotiations
(4) Following a strategy over reacting to change.

12 Principles of Agile Software Development

The Agile methodology is guided by a set of 12 core principles that promote flexibility, collaboration, and efficiency in software development.

Below is a breakdown of the key Agile principles:

1. Customer Satisfaction through Early and Continuous Delivery

Agile prioritizes delivering valuable software to customers early and consistently, ensuring that the final product meets their needs right from the start.

2. Embrace Changing Requirements

Agile teams welcome changes in requirements, even late in the development process, as it allows them to better align the software with evolving business needs.

3. Frequent Delivery of Working Software

Agile focuses on delivering working software often and in small timeframes. This helps the team get regular feedback and make improvements quickly, ensuring the software stays aligned with user needs.

4. Collaboration Between Business and Development Teams

Agile promotes close collaboration between business stakeholders and developers, ensuring that both teams are aligned in achieving common goals.

5. Empower Individuals with the Right Environment and Support

Successful Agile projects are built around motivated individuals. Teams are empowered by being provided with the right environment, resources, and trust to accomplish their tasks.

6. Prioritize Face-to-Face Communication

While digital communication is useful, Agile encourages direct, face-to-face interactions as the most effective way to convey information within a team.

7. Working Software as the Primary Measure of Progress

The primary measure of progress in Agile is the delivery of working software, rather than detailed documentation or lengthy processes.

8. Maintain a Sustainable Development Pace

Agile teams should be able to work at a constant, sustainable pace, allowing them to maintain productivity without burnout.

9. Focus on Technical Excellence and Good Design

Continuous attention to technical excellence and high-quality design enhances agility and enables faster, more efficient software development.

10. Simplicity is Essential

Agile values simplicity, emphasizing maximizing productivity by minimizing unnecessary work and focusing on essential tasks that bring the most value.

11. Encourage Self-Organizing Teams

Agile promotes self-organizing teams that can determine the best approach to design, build, and deliver software, fostering innovation and ownership.

12. Regular Reflection and Adaptation

Agile teams regularly reflect on their processes and performance, making adjustments as needed to improve their effectiveness and outcomes.

Agile Software Development Cycle

The Agile software development cycle is an iterative process that promotes flexibility and collaboration. It consists of the following key phases:

1. Concept/Planning

In this phase, teams gather requirements and define the project’s goals. It’s about identifying what needs to be built and prioritizing features based on customer needs.

2. Design

Teams create a simple design or framework for the software, outlining how the product will function. The design is kept flexible to allow for changes.

3. Development

During development, the team builds the product in small increments, typically called "sprints." Each sprint results in a working piece of the software.

4. Testing

After each sprint, the product is tested to identify bugs and ensure that the functionality works as intended. Testing is done continuously throughout the development cycle.

5. Deployment

Once a sprint is completed and tested, the working software is deployed to the customer or user for feedback. Continuous delivery is a key part of Agile.

6. Feedback/Review

Teams gather feedback from customers and stakeholders to understand what works, what doesn't, and how the product can be improved in the next iteration.

7. Iteration

Based on the feedback received, the development cycle repeats with adjustments. The process continues until the product meets the desired quality and customer expectations.

Key Agile Methodologies

Agile is a covers a number of techniques and procedures. The most common Agile Methodologies are as follows:

1. ***Scrum***

It focuses primarily on how to handle tasks in a team-based development setting, and it is an agile development methodology. Scrum basically evolved from activities that take place during a rugby round. Scrum promotes operating in small teams and thinks that the development team should be empowered (say- 7 to 9 members). Three roles make up Agile and Scrum, and their duties are described below:

- Scrum Lead: The scrum master is in charge of organizing the team, the sprint meeting, and removing roadblocks.
- Product creator: The product owner builds the product backlog, organizes it by priority, and is in charge of delivering features at each iteration.
- Agile Team: Team coordinates and oversees its own work to finish the sprint or cycle.

Product Backlog in Scrum

The number of requirements (user stories) that need to be finished for each release is tracked in this repository. The Product Owner should keep track of it, prioritize it, and share it with the scrum team. The team may also ask for the addition, modification, or elimination of a new requirement.

Scrum Practices

Scrum methodologies' workflow:

- A sprint is one iteration of a scrum.
- The list of information needed to produce the final product is called the product backlog.
- The most important user stories from the Product backlog are chosen and added to the Sprint backlog.
- Working together on the specified sprint backlog.
- Team verifies everyday work for accuracy.
- The team provides product functionality at the end of the sprint.

2. ***Extreme Programming (XP)***

The Agile framework for software development processes most closely resembles XP. It strives to build high-quality software while also simplifying the entire process for the development team. XP places a high importance on feedback, communication, simplicity, bravery, and respect.

It works best when,

- The criteria are always shifting.
- Team deadlines are constrained.
- Stakeholders desire to lower risk while meeting timelines.
- Unit and functional testing can be automated by teams.

3. ***Adaptive Software Development (ASD)***

In the first decade of the 1990s, Sam Bayer and Jim Highsmith built adaptive software development (ASD). It comprises the notions of continual adaptation, or how to adopt change rather than avoid it. Learn, collaborate, and speculate is the term of the dynamic growth process that is utilized in ASD. Because the business setting is continuously changing, this process is concentrated on close customer and development engagement and on-going learning.

ASD offers a non-linear iterative life cycle, in contrast to most software development methodologies, which use a static life cycle, i.e., Plan-Design-Build. Each process can iterate and be altered while another process is being carried out. It suggests rapid application development, which places an emphasis on speed of development to produce a high-quality, low-maintenance product that involves the user as much as possible. The following are the primary traits of ASD:

Speculate:

The major objectives and aims of the project must be established during this phase of its beginning by comprehending the constraints (risk areas) within which it must work. Maintaining coordination between teams during this phase ensures that what is learned by one team is communicated to the others and does not need to be acquired again by other groups from scratch. This phase is where the majority of the development is concentrated.

Learn:

The last stage involves several collaboration cycles, and the goal is to record all of the lessons learnt, both good and bad. The prosperity of the project relies on this stage.

4. ***Dynamic Software Development Method (DSDM)***

A group of specialists and vendors in the sector of software development built the Dynamic Software Development Method in 1994. Software programs with budgets and constrained schedules are the primary priority of DSDM. It accentuates regular product process delivery, and growth is iterative and gradual.

With the Dynamic Software Development Method (DSDM), a roadmap of continuous and early deliveries can be created for the project. This allows for the implementation of an incremental solution, adaptation in response to feedback received along the way, and verification that the anticipated benefits are being realized.

The DSDM is an agile model that may unquestionably assist companies used to operating on projects to alter their mindset and method of operation in order to increase their ability to create value and shorten time to market.

5. ***Feature Driven Development (FDD)***

The key component of this methodology is "designing & creating" features. FDD, in contrast to other Agile development techniques in software engineering, outlines very precise and condensed work phases that must be completed separately for each feature. Domain walkthrough, design review, promotion to build, code review, and design are all included. FDD creates products with the target market in mind.

- Visibility of progress and results
- Regular Builds
- Configuration Management
- Inspections
- Feature Teams
- Component/ Class Ownership
- Development by feature
- Domain object Modeling

6. ***Kanban***

Without adding to the stress of the software development lifecycle, Kanban is a highly visual workflow management technique that enables teams to actively supervise product creation, with a focus on continuous delivery (SDLC). It has gained popularity among groups that use Lean software development techniques.

The three fundamental tenets of Kanban are to visualize the workflow, reduce the amount of work that is in process, and enhance the flow of work. The Kanban technique is intended to aid teams in collaborating more effectively, much like Scrum is. It promotes an atmosphere of active and continuing learning and growth by encouraging continuous collaboration and attempting to establish the ideal process.

7. ***Behavior Driven Development (BDD)***

A behavior-focused agile system development methodology is called behavior driven development (BDD). It was created by Dan North in 2003 as an extension of the TDD methodology. Dan North tried to include non-technical people when creating the system’s technological functionality. Inadvertently leaving out business principles that are already part of the functionality when developing software can occasionally lead to repeated and even serious defects.

BDD uses universal language concepts to facilitate communication inside a software project between persons with and without technical expertise. The BDD development process is built on the writing of test cases and features. These provide the guidelines and requirements for proper system operation. It defines what is necessary for the functionality to start, what will happen next, and what the results will be after it has been completed. Teams who use BDD are better able to communicate needs clearly, find bugs quickly, and create long-lasting software.

Benefits of Agile Software Development

Agile Software Development methodologies are increasingly popular among leaders and developers for several compelling reasons:

- *Enhanced Collaboration*: Agile fosters strong collaboration between development teams and clients, resulting in greater clarity and satisfaction.
- *Higher Product Quality:* Regular testing allows for early issue detection, leading to higher-quality, well-tested products.
- *Flexibility to Change:* Agile enables teams to quickly adapt to shifting client needs and market changes, ensuring timely delivery of relevant products.
- *Lower Risk and Faster ROI:* Frequent reviews reduce risks, allowing teams to pivot quickly if issues arise. Agile’s focus on client feedback ensures decisions enhance product value.
- *Continuous Improvement*: Agile emphasizes iterative development and regular retrospectives, allowing teams to learn from each cycle and improve processes continuously.
- *Customer-Centric Focus*: By involving customers throughout the development process, Agile ensures that the final product aligns closely with user needs and expectations, leading to increased customer satisfaction

Limitations of Agile Software Development

While Agile offers many benefits, it also presents some challenges that teams need to consider.

- *Difficulty in Estimation*: Estimating effort for large projects can be challenging in Agile, leading to uncertainties in timelines and resource allocation.
- *Reduced Documentation:* The focus on working software may result in less documentation, creating information gaps that can lead to misunderstandings.
- *Customer Dependence:* Agile’s need for customer feedback means that unclear input can misguide the project, leading to products that may not effectively meet user needs.
- *Unpredictability:* Regular changes and constant feedback can make it hard to predict how long a project will take or how much it will cost, which can lead to adding more work than planned.
- *Burnout Risk:* The fast-paced nature of Agile can lead to team burnout if adequate breaks and support are not provided, impacting morale and productivity.

****

Agile vs. Waterfall

Agile is an iterative approach to software development, emphasizing flexibility and collaboration among cross-functional teams. It focuses on delivering small, incremental releases, adapting to changes throughout the development process.

In contrast, Waterfall is a linear and sequential approach, where each phase must be completed before moving to the next. Changes are difficult to incorporate once a phase is completed. Agile promotes adaptability and customer feedback, while Waterfall provides a structured plan but may struggle with accommodating changes late in the development cycle.

When it comes to software development methodologies, Agile and Waterfall are two prominent approaches that offer distinct advantages and challenges. Understanding the key characteristics, pros, and cons of each methodology is crucial for choosing the most suitable approach for your project.

Is Agile more expensive than Waterfall?

Agile may seem more expensive initially due to its ongoing feedback loops, but it often reduces expenses over time by addressing issues early and delivering value incrementally. Waterfall, with its fixed upfront costs, can lead to higher expenses if changes are needed later. The cost effectiveness depends on the project's requirements and context.

The Agile Methodology

Agile methodology is characterized by its iterative and flexible approach to software development. It emphasizes collaboration among cross-functional teams and prioritizes adaptability to changing requirements throughout the development process.

The Pros:

- Flexibility: Agile allows teams to adapt to changing requirements and priorities, ensuring the final product meets the evolving needs of stakeholders.
- Faster Delivery: By breaking down the project into smaller increments, Agile enables faster delivery of working software, leading to quicker feedback and validation.
- Collaboration: Agile fosters collaboration among team members, promoting better communication and transparency throughout the development cycle.
- Continuous Improvement: The iterative nature of Agile encourages continuous improvement, as teams regularly assess their progress and make necessary adjustments based on feedback.

The Cons:

- Predictability: Agile may struggle with maintaining predictability in project timelines and budgets, as requirements evolve over time.
- Stakeholder Involvement: Agile requires active involvement from stakeholders and dedicated resources, which may not always be feasible.
- Scope Management: Agile is less suitable for projects with fixed scope and strict deadlines, as changes in requirements can impact project scope and timelines.

The Waterfall Methodology

Waterfall methodology follows a linear and sequential approach to software development, with distinct phases such as requirements gathering, design, implementation, testing, and maintenance.

The Pros:

- Clear Roadmap: Waterfall provides a clear roadmap for project execution, making it easier to plan and estimate resources and timelines upfront.
- Stability: Well-suited for projects with stable requirements and predefined objectives, Waterfall ensures thorough documentation and adherence to established processes.
- Predictability: Waterfall offers predictability in project timelines and budgets, as requirements are defined upfront and changes are minimized during development.

The Cons:

- Rigidity: The rigidity of Waterfall can be a drawback when faced with evolving requirements, making it challenging to accommodate changes late in the development cycle.
- Flexibility: Waterfall lacks the flexibility of Agile to adapt to changes quickly, potentially leading to delays or rework if issues arise.
- Limited Feedback: Due to its sequential nature, Waterfall may limit feedback opportunities until later stages of development, increasing the risk of delivering a product that doesn't meet stakeholder expectations.

Choosing Between Agile and Waterfall

When deciding between Agile and Waterfall, consider the specific needs and characteristics of your project and its needs:

When Agile is ideal:

- Ideal for dynamic environments or projects with evolving requirements.
- Suitable for startups, innovative products, and continuous improvement initiatives.
- Requires active stakeholder involvement and dedicated resources.

When Waterfall is ideal:

- Preferable for projects with well-defined requirements and strict budgets or deadlines.
- Commonly used in government contracts, large-scale infrastructure projects, and industries with regulatory compliance needs.
- Offers predictability in project timelines and budgets, but may struggle with accommodating changes late in the development cycle.

DevOps

Self-Learning Section:

What is DevOps?

DevOps is a software engineering methodology that combines development (Dev) and operations (Ops) to integrate the work of these teams. It's a culture that emphasizes collaboration, automation, and continuous improvement.

DevOps Importance and Benefits

DevOps help organizations deliver products faster, improve quality, and reduce costs:

Here are some of the benefits of DevOps:

- Faster delivery:
  DevOps can help organizations release new features and fix bugs more quickly, which can help them respond to customer needs.
- Improved quality:
  DevOps practices like continuous integration and continuous delivery (CI/CD) can help ensure that software is developed quickly while maintaining high quality standards.
- Better collaboration:
  DevOps can help break down silos between development and operations teams, which can lead to better communication and collaboration.
- Faster issue resolution:
  DevOps can help organizations resolve issues faster and reduce complexity.
- Greater scalability:
  DevOps can help organizations manage complex or changing systems efficiently and with reduced risk.
- Improved security:
  DevOps can help organizations move quickly while retaining control and preserving compliance.

DevOps Principles and Practices

Some principles and practices of DevOps:

- Continuous integration:
  A key practice that involves regularly building and validating a project. The development team updates code changes in the repository on a regular basis.
- Infrastructure as Code (IaC):
  A key practice that uses coding techniques to deliver infrastructure and the software that runs on it.
- Continuous delivery:
  A major DevOps practice that involves continuously building, testing, and delivering code. This can lead to benefits in speed, stability, and flexibility.
- Continuous monitoring:
  An operational phase that involves monitoring the performance of a software application to improve its overall efficiency.
- Continuous deployment:

A pillar of DevOps that allows tech teams to adjust to changing market demands faster.
- Automation:
  A key principle of DevOps that involves automating everything from code generation to monitoring the system in production.
- Continuous testing:
  An important practice that involves obtaining immediate feedback on the business risk associated with the release phase of the software delivery lifecycle.
- Collaboration:
  A core principle of DevOps that involves bringing together the development and operations teams to work as a single unit.

7 C's of DevOps

The 7 C's of DevOps are a holistic approach to the DevOps lifecycle that guide practices in each phase:

1. Continuous development
2. Continuous integration
3. Continuous testing
4. Continuous deployment
5. Continuous feedback
6. Continuous monitoring
7. Continuous operations

Lifecycle for Business Agility

The lifecycle for business agility can include several stages, including:

- Ideation: The project owner works with stakeholders, the business team, developers, and future users
- Development: The team builds the first iteration of the software
- Testing: The team tests the first iteration of the software
- Deployment: The team deploys the software to the cloud or an on-premise server
- Operations: The team tests the software in production and end users use it
- Continuous feedback: The team assesses customer behavior and input to identify areas for improvement
- Continuous monitoring: The team uses tools to monitor software functionality and identify anomalies
- Automated dashboards: The team uses dashboards to track progress, spot problems, and make informed decisions

DevOps and Continuous Testing

Continuous Testing refers to running automated tests every time code changes are made, providing fast feedback as part of the software delivery pipeline. It was introduced to help developers quickly identify, notify, and fix issues. The goal is to test more frequently, starting with individual components and later testing the entire codebase.

Continuous Testing plays a significant role in DevOps:

1. Software Development Evolution:

Traditionally, software development followed a rigid process with defined timelines for development and QA phases. Code would be passed between teams, making updates and deployments slow and cumbersome.

2. Agile Transformation:

The shift to Agile methodologies has made real-time changes and feature updates more convenient.

The integration of Continuous Testing and CI/CD pipelines accelerates the movement of code through the development, testing, and deployment stages.

3. Continuous Testing in DevOps:

In Agile DevOps, Continuous Testing ensures code quality by running automated unit tests continuously throughout the development cycle.

For example, a QA engineer checks automated unit tests on Jenkins servers. If tests pass, the build moves to the QA servers for further testing (e.g., load and functional testing). If tests fail, the build is rejected, and the developer is notified.

4. Risk Mitigation:

Continuous Testing allows for the early detection of critical bugs, minimizing the risk of costly fixes later in the process.

By catching issues early, teams save time and resources in the later stages of development.

5. Test Automation:

Continuous Testing encourages the automation of tests at each stage of the development cycle, ensuring software quality and code validity.

6. Readiness for Deployment:

Continuous Testing helps teams evaluate the readiness of the software for deployment, ensuring it’s production-ready by the time it passes through the delivery pipeline.

How to Choose Right DevOps Tools?

When choosing DevOps tools, you can consider:

- Needs: What your development process needs, and its strengths and weaknesses
- Integration: How well the tools integrate with other tools, and if they are provided by multiple providers
- Automation: Whether the tools can automate different software development processes
- Ease of use: How easy the tools are to use and manage, and if they have a central dashboard
- Monitoring: Whether the tools have monitoring and analytics features
- Collaboration: Whether the tools have features for real-time collaboration
- Bug detection: Whether the tools have capabilities for detecting and fixing bugs

Here are some popular DevOps tools:

Jenkins
Helps companies build build pipelines quickly, which can reduce risk in the software development lifecycle

Docker
Makes containerization easier, where each container includes everything needed to create a product

Puppet
Helps manage the DevOps infrastructure lifecycle, including provisioning, configuration, management, and compliance

Chef
A popular tool for experienced DevOps teams that want to automate their deployment and development infrastructure

Prometheus
A popular monitoring tool that provides a comprehensive solution for detecting and resolving system issues

Nagios
Helps DevOps teams monitor the health and status of their IT infrastructure, and provides alerting capabilities

Jira
A popular collaboration tool that helps with planning complex software projects

Splunk
A top load management and analysis solution that is in high demand.

Challenges with DevOps Implementation

Here are some challenges that can arise when implementing DevOps:

1. Resistance to change: People may be reluctant to change, even at work. This can include not being interested in new tools or processes, or not wanting to learn new skills.
2. Managing multiple environments: Managing environments for development, testing, and production can be difficult, especially if there's no plan in place.
3. Security concerns: When using AI and ML in DevOps, security is a major concern. This is because AI and ML rely on large amounts of data shared through connected systems.
4. Environmental issues: When code moves between teams, there can be waste because each environment is configured differently.
5. Moving away from legacy systems: Migrating from legacy systems can be costly, complex, and disruptive. It can also lead to data loss, user resistance, and security risks.
6. Lack of knowledge: Smaller teams or organizations may not have the knowledge to include databases in the DevOps process.

7. Distraction from tools: There are many tools available, and each one markets itself as the solution to every issue.
8. Inconsistent ecosystems: DevOps can face challenges with inconsistent ecosystems.
9. Manual processes: DevOps can face challenges with manual processes.
10. Inadequate access control: DevOps can face challenges with inadequate access control.
11. Skill gaps: DevOps can face challenges with skill gaps.
12. Lack of compatibility between multiple tools: DevOps can face challenges with lack of compatibility between multiple tools.