Module 3 Implementation and Testing

Design Patterns

The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem.

Pattern Elements:

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways.
4. A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether a pattern can be used in a particular situation.

Design Problems:

To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied:

1. Tell several objects that the state of some other object has changed.
2. Tidy up the interfaces to a number of related objects that have often been developed incrementally.
3. Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented.
4. Allow for the possibility of extending the functionality of an existing class at runtime.

Implementation Issues

Some aspects of implementation that are particularly important to software engineering and that are often not covered in programming texts. These are:

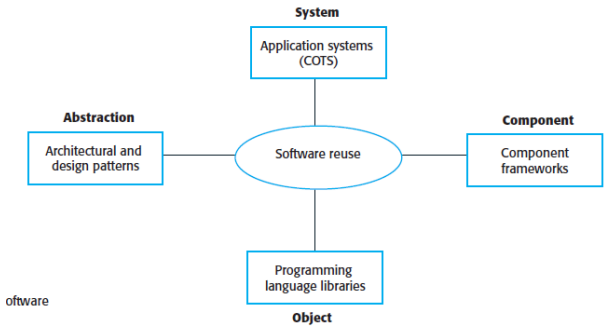
1. *Reuse* Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
2. *Configuration management* During the development process, many different versions of each software component are created. If you don't keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.
3. *Host-target development* Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system). The host and target systems are sometimes of the same type, but often they are completely different.

1. Reuse:

Software reuse is possible at several different levels, as shown in Figure 7.13:

1. *The abstraction level:* At this level, you don't reuse software directly but rather use knowledge of successful abstractions in the design of your software.
2. *The object level:* At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you must find appropriate libraries and discover if the objects and methods offer the functionality that you need.
3. *The component level:* Components are collections of objects and object classes that operate together to provide related functions and services. You often must adapt and extend the component by adding some code of your own.

4. *The system level:* At this level, you reuse entire application systems. This function usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface.



oftware

2. *Configuration Management:*

Configuration management is the name given to the general process of managing a changing software system. The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system

There are four fundamental configuration management activities:

1. *Version management,* where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers. They stop one developer from overwriting code that has been submitted to the system by someone else.
2. *System integration,* where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
3. *Problem tracking,* where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.
4. *Release management,* where new versions of a software system are released to customers. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

3. *Host – target development*

Software is developed on one computer (the host) but runs on a separate machine (the target). More generally, we can talk about a development platform (host) and an execution platform (target). A platform is more than just hardware. It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
IDE (Integrated Development Environment):

- Software development tools are now usually installed within an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development within some common framework and user interface.
- Generally, IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially or may be an instantiation of a general-purpose IDE, with specific language-support tools.
- A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed and integration mechanisms that allow tools to work together. The best-known general-purpose IDE is the Eclipse

OPEN-SOURCE DEVELOPMENT

- Open-source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process

- Open-source software extended this idea by using the Internet to recruit a much larger population of volunteer developers.

- Many of them are also users of the code. In principle at least, any contributor to an open-source project may report and fix bugs and propose new features and functionality.

- However, in practice, successful open-source systems still rely on a core group of developers who control changes to the software.

- Open-source software is the backbone of the Internet and software engineering.

- The Linux operating system is the most widely used server system, as is the open-source Apache web server. Other important and universally used open- source products are Java, the Eclipse IDE, and the MySQL database management system

- For a company involved in software development, there are two open-source issues that have to be considered:
  1) Should the product that is being developed make use of open-source components?
  2) Should an open-source approach be used for its own software development?

Open-Source Licencing

- Although a fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
- Legally, the developer of the code (either a company or an individual) owns the code. They can place restrictions on how it is used by including legally binding conditions in an open-source software .
- Some open-source developers believe that if an open-source component is used to develop a new system, then that system should also be open source.
- Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed-source systems.

- Most open-source licenses are variants of one of three General models:

1. The GNU General Public License (GPL). This is a so-called reciprocal license that simplistically means that if you use open-source software that is licensed under the GPL license, then you must make that software open source.
2. The GNU Lesser General Public License (LGPL). This is a variant of the GPL license where you can write components that link to open-source code without having to publish the source of these components. However, if you change the licensed component, then you must publish this as open source.
3. The Berkley Standard Distribution (BSD) License. This is a nonreciprocal license, which means you are not obliged to re-publish any changes or modifications made to open-source code. You can include the code in proprietary systems that are sold. If you use open-source components, you must acknowledge the original creator of the code. The MIT license is a variant of the BSD license with similar conditions.

Licence Management:

1. Establish a system for maintaining information about open-source components that are downloaded and used.
2. Be aware of the different types of licenses and understand how a component is licensed before it is used.
3. Be aware of evolution pathways for components. You need to know a bit about the open-source project where components are developed to understand how they might change in future.
4. Educate people about open source.
5. Have auditing systems in place.
6. Participate in the open-source community.

REVIEW TECHNIQUES

Cost of Impact of Software Defects

- Within the context of software process the terms *defect* and *fault* are synonymous. Both imply a quality problem that is discovered *after* the software has been released to end users.

- The primary objective of technical reviews is to find errors during the process so that they do not become defects after release of the software. The obvious benefit of technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.

INFORMAL REVIEW

Informal reviews include a simple desk check of a software engineering work product with a colleague, a casual meeting (involving more than two people) for the purpose of reviewing a work product, or the review-oriented aspects of pair programming

A simple *desk check* or a *casual meeting* conducted with a colleague is a review. However, because there is no advance planning or preparation, no agenda or meeting structure, and no follow-up on the errors that are uncovered, the effectiveness of such reviews is considerably lower than more formal approaches. But a simple desk check can and does uncover errors that might otherwise propagate further into the software process.

One way to improve the efficacy of a desk check review is to develop a set of simple review checklists for each major work product produced by the software team. The questions posed within the checklist are generic, but they will serve to guide the reviewers as they check the work product

Pair Programming: Pair programming can be characterized as a continuous desk check. Rather than scheduling a review at some point in time, pair programming encourages continuous review as a work product (design or code) is created. The benefit is immediate discovery of errors and better work product quality as a consequence.

FORMAL TECHNICAL REVIEW

A *formal technical review* (FTR) is a software quality control activity performed by software engineers (and others).
The objectives of an FTR are:
(1) to uncover errors in function, logic, or implementation for any representation of the software;
(2) to verify that the software under review meets its requirements;
(3) to ensure that the software has been represented according to predefined standards;
(4) to achieve software that is developed in a uniform manner; and

(5) to make projects more manageable.

- In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation.
- The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

Review Meetings:

Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.
- The focus of the FTR is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component)

Review Reporting and Record Keeping

During the FTR, a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting, and a *review issues list* is produced. In addition, a *formal technical review summary report* is completed. A review summary report answers three questions:

1. What was reviewed?

2. Who reviewed it?

3. What were the findings and conclusions?

The review summary report is a single-page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties.

The review issues list serves two purposes: (1) to identify problem areas within the product and (2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report.

Review Guidelines

Guidelines for conducting formal technical reviews must be established in advance, distributed to all reviewers, agreed upon, and then followed. A review that is uncontrolled can often be worse than no review at all. The following represents a minimum set of guidelines for formal technical reviews:

- Review the product, not the producer.
- Set an agenda and maintain it.
- Limit debate and rebuttal
- Enunciate problem areas, but don't attempt to solve every problem noted.
- Take written notes
- Limit the number of participants and insist upon advance preparation
- Develop a checklist for each product that is likely to be reviewed
- Allocate resources and schedule time for FTRs.
- Conduct meaningful training for all reviewers
- Review your early reviews.

POST MORTEM EVALUATIONS

- Baaz and his colleagues suggest the use of a *post-mortem evaluation* (PME) as a mechanism to determine what went right and what went wrong when software engineering process and practice are applied in a specific project.
- PME examines the entire software project, focusing on both “excellences (that is, achievements and positive experiences) and *challenges* (problems and negative experiences)”

- Often conducted in a workshop format, a PME is attended by members of the software team and stakeholders.
- The intent is to identify excellences and challenges and to extract lessons learned from both. The objective is to suggest improvements to both process and practice going forward.

SOFTWARE TESTING STRATEGIES

- Testing is a set of activities that can be planned in advance and conducted systematically.

- Software is tested to uncover errors that were made inadvertently as it was designed and constructed.

Test Strategies for Conventional Software

1. *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 as a guide, important control paths are tested to uncover errors within the boundary of the module.
- The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing.
- 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 for multiple components.

Unit Testing consideration:

- The module interface is tested to ensure that information properly flows into and out of the program unit under test.
- 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

Unit Testing Procedures:

- Unit testing is normally considered as an adjunct to the 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.

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.

- **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.

- 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 actual components.
  3. Tests are conducted as each component is integrated.
  4. On completion of each set of tests, another stub is replaced with the real component.
  5. Regression testing may be conducted to ensure that new errors have not been introduced.

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 subfunction.
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.

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

3. *Validation Testing*

- 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 different software categories disappears.
- Testing focuses on user-visible actions and user-recognizable output from the system.

- 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 defines 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.

- 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.

- 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

4. <u>System Testing</u>

a. 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), 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.

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

c. Stress Testing: *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.

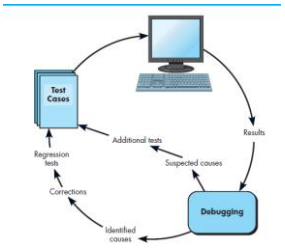
d. Performance Testing: Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted.

e. Deployment Testing: *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.

5. <u>Debugging:</u>

- *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.

- 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.



Strategies:

- The *brute force* category of debugging is probably the most common and least effi cient 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

- *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

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, you 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.

PATH TESTING

Path Testing is a method that is used to design the test cases. In path testing method, the control flow graph of a program is designed to find a set of linearly independent paths of execution. In this method Cyclomatic Complexity is used to determine the number of linearly independent paths and then test cases are generated for each path.

It give complete branch coverage but achieves that without covering all possible paths of the control flow graph. McCabe's Cyclomatic Complexity is used in path testing. It is a structural testing method that uses the source code of a program to find every possible executable path.

Path Testing Process:

1. Control Flow Graph: Draw the corresponding control flow graph of the program in which all the executable paths are to be discovered.
2. Cyclomatic Complexity: After the generation of the control flow graph, calculate the cyclomatic complexity of the program using the following formula. McCabe's Cyclomatic Complexity = E - N + 2P
   Where, E = Number of edges in control flow graph N = Number of vertices in control floe graph P = Program factor
3. Make Set: Make a set of all the path according to the control floe graph and calculated. The cardinality of set is equal to the calculated cyclomatic complexity.
4. Create Test Cases: Create test case for each path of the set obtained in above step.

Path Testing Techniques:

1. Control Flow Graph: The program is converted into control flow graph by representing the code into nodes and edges.
2. Decision to Decision path: The control flow graph can be broken into various Decision to Decision paths and then collapsed into individual nodes.

3. Independent paths: Independent path is a path through a Decision-to-Decision path graph which cannot be reproduced from other paths by other methods.

CONTROL STRUCTURE TESTING

Control structure testing is used to increase the coverage area by testing various control structures present in the program. The different types of testing performed under control structure testing are as follows-

1. Condition Testing 2. Data Flow Testing 3. Loop Testing

1. **Condition Testing:** Condition testing is a test cased design method, which ensures that the logical condition and decision statements are free from errors. The errors present in logical conditions can be incorrect Boolean operators, missing parenthesis in a booleans expression, error in relational operators, arithmetic expressions, and so on.
2. **Data Flow Testing:** The data flow test method chooses the test path of a program based on the locations of the definitions and uses all the variables in the program.
3. **Loop Testing:** Loop testing is actually a white box testing technique. It specifically focuses on the validity of loop construction. Three types of loops: simple, structured and unstructured.

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 is not an alternative to white-box techniques.
- Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.
- 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.
- Unlike white-box testing, which is performed early in the testing process, black box testing tends to be applied during later stages of testing .
- Because black-box testing purposely disregards control structure, attention is focused on the information domain.
- The first step in black-box testing is to understand the objects 5 that are modelled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to defi ne a series of tests that verify “all objects. 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.

- *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. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed.

- 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. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well. Boundary value analysis leads to a selection of test cases that exercise bounding values.

- *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.

TEST DOCUMENTATION

- Test documentation is documentation of artifacts created before or during the testing of software. It helps the testing team to estimate testing effort needed, test coverage, resource tracking, execution progress, etc. It is a complete suite of documents that allows you to describe and document test planning, test design, test execution, test results that are drawn from the testing activity.

Types of Testing Documents	Description
Test policy	It is a high-level document which describes principles, methods and all the important testing goals of the organization.
Test strategy	A high-level document which identifies the Test Levels (types) to be executed for the project.
Test plan	A test plan is a complete planning document which contains the scope, approach, resources, schedule, etc. of testing activities.
Requirements Traceability Matrix	This is a document which connects the requirements to the test cases.
Test Scenario	Test scenario is an item or event of a software system which could be verified by one or more Test cases.
Test case	It is a group of input values, execution preconditions, expected execution postconditions and results. It is developed for a Test Scenario.
Test Data	Test Data is a data which exists before a test is executed. It used to execute the test case.
Defect Report	Defect report is a documented report of any flaw in a Software System which fails to perform its expected function.
Test summary report	Test summary report is a high-level document which summarizes testing activities conducted as well as the test result.

- The main reason behind creating test documentation is to either reduce or remove any uncertainties about the testing activities. Helps you to remove ambiguity which often arises when it comes to the allocation of tasks
- Documentation not only offers a systematic approach to software testing, but it also acts as training material to freshers in the software testing process
- It is also a good marketing & sales strategy to showcase Test Documentation to exhibit a mature testing process
- Test documentation helps you to offer a quality product to the client within specific time limits.
- In Software Engineering, Test Documentation also helps to configure or set-up the program through the configuration document and operator manuals.
- Test documentation helps you to improve transparency with the client

TEST AUTOMATION

In software testing, test automation is the use of software separate from the software being tested to control the execution of tests and the comparison of actual outcomes with predicted outcomes.[1] Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or perform additional testing that would be difficult to do manually. Test automation is critical for continuous delivery and continuous testing.

There are many approaches to test automation, however, below are the general approaches used widely:

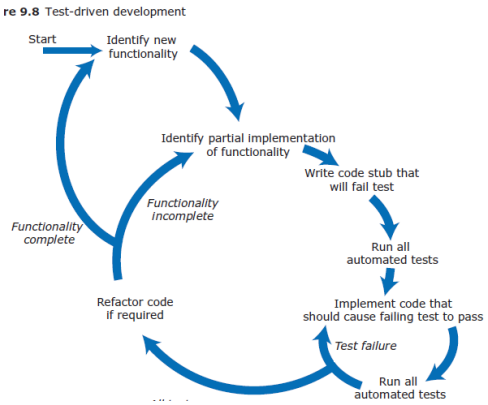
1. **Graphical user interface testing**. A testing framework that generates user interface events such as keystrokes and mouse clicks, and observes the changes that result in the user interface, to validate that the observable behaviour of the program is correct.
2. **API driven testing**. A testing framework that uses a programming interface to the application to validate the behaviour under test. Typically, API driven testing bypasses application user interface altogether. It can also be testing public (usually) interfaces to classes, modules or libraries are tested with a variety of input arguments to validate that the results that are returned are correct.

Levels: A strategy to decide the amount of tests to automate is the test automation pyramid. This strategy suggests to write three types of tests with different granularity. The higher the level, less is the amount of tests to write

- As a solid foundation, Unit testing provides robustness to the software products. Testing individual parts of the code makes it easy to write and run the tests.
- The service layer refers to testing the services of an application separately from its user interface, these services are anything that the application does in response to some input or set of inputs.
- At the top level we have UI testing which has fewer tests due to the different attributes that make it more complex to run, for example the fragility of the tests, where a small change in the user interface can break a lot of tests and adds maintenance effort.

TEST DRIVEN DEVELOPMENT

- Test-driven development (TDD) is an approach to program development that is based on the general idea that you should write an executable test or tests for code that you are writing before you write the code.



- Test-driven development relies on automated testing. Every time you add some functionality, you develop a new test and add it to the test suite. All of the tests in the test suite must pass before you move on to developing the next increment.

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.

The benefits of test-driven development are:

1. It is a systematic approach to testing in which tests are clearly linked to sections of the program code. This means you can be confident that your tests cover all of the code that has been developed and that there are no untested code sections in the delivered code.
2. The tests act as a written specification for the program code. In principle 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.

Test-driven development works best for the development of individual program units; it is more difficult to apply to system testing. Even the strongest advocates of TDD accept that it is challenging to use this approach when you are developing and testing systems with graphical user interfaces.

SECURITY TESTING

- Security testing has comparable goals. It aims to find vulnerabilities that an attacker may exploit and to provide convincing evidence that the system is sufficiently secure. The tests should demonstrate that the system can resist attacks on its availability, attacks that try to inject malware, and attacks that try to corrupt or steal users' data and identity.

- Comprehensive security testing requires specialist knowledge of software vulnerabilities and approaches to testing that can find these vulnerabilities.

- One practical way to organize security testing is to adopt a risk-based approach, where you identify the common risks and then develop tests to demonstrate that the system protects itself from these risks. You may also use automated tools that scan your system to check for known vulnerabilities, such as unused HTTP ports being left open.

- In a risk-based approach, you start by identifying the main security risks to your product. To identify these risks, you use knowledge of possible attacks, known vulnerabilities, and security problems.

- Based on the risks that have been identified, you then design tests and checks to see if the system is vulnerable. It may be possible to construct automated tests for some of these checks, but others inevitably involve manual checking of the system's behavior and its files.

- Once you have identified security risks, you then analyse them to assess how they might arise. You can then develop tests to check some of these possibilities.

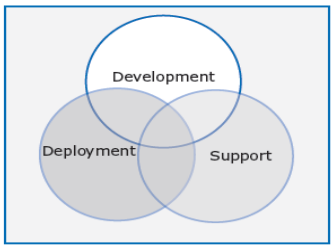
DEV – OPS AND CODE MANAGEMENT

DevOps (development + operations) integrates development, deployment, and support, with a single team responsible for all of these activities (Figure 10.2). 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. Agile enthusiasts started looking for a way around this problem.
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. Amazon's claim that this led to significant improvements in reliability was widely publicized.
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.

Table 10.1 DevOps principles

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.



Multi-skilled DevOps team

Table 10.2 Benefits of DevOps

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.

- DevOps aims to change this by creating a single team that is responsible for both development and operations. Developers also take responsibility for installing and maintaining their software.

- Creating a DevOps team means bringing together a number of different skill sets, which may include software engineering, UX design, security engineering, infrastructure engineering, and customer interaction.

- A successful DevOps team has a culture of mutual respect and sharing. Everyone on the team should be involved in Scrums and other team meetings. Team members should be encouraged to share their expertise with others and to learn new skills. Developers should support the software services that they have developed.

CODE MANAGEMENT

- Code management1 is a set of software-supported practices used to manage an evolving codebase.
- You 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

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.

Features of Source Code Management

Table 10.4 Features of source code management systems

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.

DEV – OPS AUTOMATION

Table 10.5 Aspects of DevOps automation

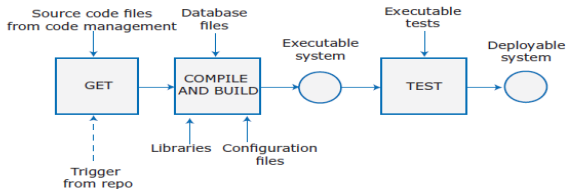
Aspect	Description
Continuous integration	Each time a developer commits a change to the project's master branch, an executable version of the system is built and tested.
Continuous delivery	A simulation of the product's operating environment is created and the executable software version is tested.
Continuous 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.

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.

- 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 10.9 Continuous 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.

- The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system.

- If you 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.

- 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.

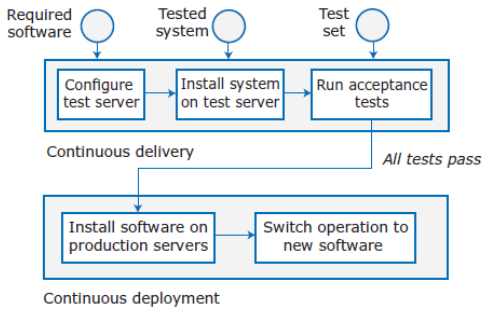
Continuous Delivery and deployment

- 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.

- Continuous delivery means that, after making changes to a system, you 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.



- Continuous deployment is obviously only practical for cloud-based systems. If your product is sold through an app store or downloaded from your website, continuous integration and delivery make sense

Table 10.6 Benefits of continuous deployment

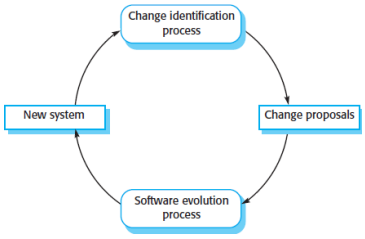
Benefit	Explanation
Reduced costs	If you use continuous deployment, you have no option but to invest in a completely automated deployment pipeline. Manual deployment is a time-consuming and error-prone process. Setting up an automated system is expensive and takes time, but you can recover these costs 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.

SOFTWARE EVOLUTION

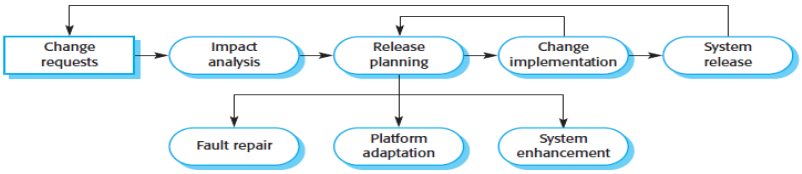
- The most appropriate evolution process for a software system depends on the type of software being maintained, the software development processes used in an organization, and the skills of the people involved

- Formal or informal system change proposals are the driver for system evolution in all organizations.

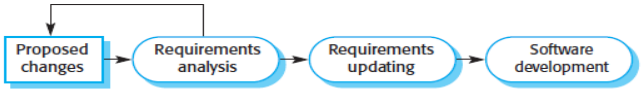
- The processes of change identification and system evolution are cyclical and continue throughout the lifetime of a system



Software evolution process:



Change implementation:



- In situations where development and evolution are integrated, change implementation is simply an iteration of the development process

- a critical difference between development and evolution is that the first stage of change implementation requires program understanding.

- During the program understanding phase, new developers have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program. They need this understanding to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

Agile Methods and evolution

- Agile techniques such as test-driven development and automated regression testing are useful when system changes are made.
- System changes may be expressed as user stories, and customer involvement can help prioritize changes that are required in an operational system.
- The Scrum approach of focusing on a backlog of work to be done can help prioritize the most important system changes.
- In short, evolution simply involves continuing the agile development process.

SOFTWARE MAINTENANCE

- Software maintenance is the general process of changing a system after it has been delivered.
- The term is usually applied to custom software, where separate development groups are involved before and after delivery.
- The changes made to the software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or to accommodate new requirements.

Types of maintenance

1. *Fault repairs to fix bugs and vulnerabilities.* Coding errors are usually relatively cheap to correct; design errors are more expensive because they may involve rewriting several program components. Requirements errors are the most expensive to repair because extensive system redesign may be necessary.
2. *Environmental adaptation to adapt the software to new platforms and environments.* This type of maintenance is required when some aspect of a system's environment, such as the hardware, the platform operating system, or other support software, changes. Application systems may have to be modified to cope with these environmental changes.
3. *Functionality addition to add new features and to support new requirements.* This type of maintenance is necessary when system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance

Maintenance Prediction: Maintenance prediction is concerned with trying to assess the changes that may be required in a software system and with identifying those parts of the system that are likely to be the most expensive to change.

Change Prediction:

- Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment. Some systems have a very complex relationship with their external environment, and changes to that environment inevitably result in changes to the system. To evaluate the relationships between a system and its environment, you should look at:
  1. *The number and complexity of system interfaces*
  2. *The number of inherently volatile system requirements*
  3. *The business processes in which the system is used*

Software Reengineering

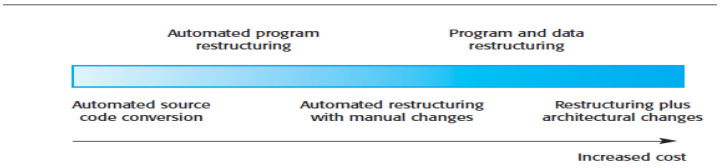
- Reengineering may involve redocumenting the system, refactoring the system architecture, translating programs to a modern programming language, or modifying and updating the structure and values of

the system's data. The functionality of the software is not changed, and, normally, you should try to avoid making major changes to the system architecture.

- Reengineering has two important advantages over replacement:
  1. *Reduced risk* There is a high risk in redeveloping business-critical software. Errors may be made in the system specification or there may be development problems
  2. *Reduced cost* The cost of reengineering may be significantly less than the cost of developing new software

- The activities in this reengineering process are:
  1. *Source code translation: convert code to a new language*
  2. *Reverse engineering: analyse the program to understand it*
  3. *Program structure improvement: restructure automatically for understandability.*
  4. *Program modularization: reorganize program structure.*
  5. *Data reengineering: clean up and restructure system data.*

Reengineering approaches:



Cost Factors:

1. Quality of software
2. Tool support available.
3. Extent of data conversion
4. Availability of expert staff.

Refactoring

- Refactoring is the process of making improvements to a program to slow down degradation through change.
- It means modifying a program to improve its structure, reduce its complexity, or make it easier to understand.
- Refactoring is sometimes considered to be limited to object-oriented development, but the principles can in fact be applied to any development approach.
- When you refactor a program, you should not add functionality but rather should concentrate on program improvement.
- You can therefore think of refactoring as "preventative maintenance" that reduces the problems of future change.

- Reengineering takes place after a system has been maintained for some time, and maintenance costs are increasing. You use automated tools to process and reengineer a legacy system to create a new system that is more maintainable.
- Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.