# UNIT IV

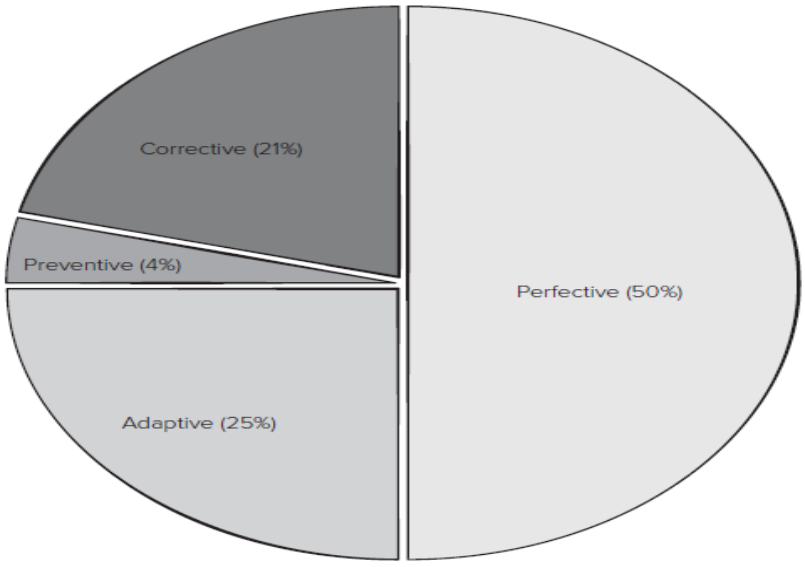
## Software Maintenance: -

Maintenance is defined as the activities needed to keep software operational after it has been accepted and delivered (released) in the end-user environment. Maintenance will continue for the life of the software product.

Maintenance Type –

1. Corrective maintenance is the reactive modification of software to repair problems discovered after the software has been delivered to a customer's end user.
2. Adaptive maintenance is reactive modification of software after delivery to keep the software usable in a changing end-user environment.
3. Perfective maintenance is the proactive modification of the software after delivery to provide new user features, better program code structure, or improved documentation.
4. Preventive maintenance is the proactive modification of the software after delivery to detect and correct product faults before they are discovered by users in the field

Distribution of Maintenance Effort




## 1. Corrective maintenance

It is one of the primary types of maintenance approaches and is often employed as a reactive measure to restore functionality to a system.

**Aspects of the Corrective Approach –**

1. Problem Identification: It involves identification of a problem that has occurred through bug reports, user reports, error messages, or regular system checks.

2. Diagnosis and Analysis: Once a problem is identified, a diagnostic process follows to determine the root cause. It involves system analysis, debugging, or using diagnostic tools.

Root cause analysis to understand why the fault occurred and ensures that the corrective action addresses the actual problem rather than just the symptoms.

3. Correction or Repair: The corrective action is taken based on the diagnosis. This involve fixing bugs or patching vulnerabilities.

4. Testing and Verification: After corrective actions are implemented, test the system to ensure that the problem is resolved. This could involve running the system under normal and stressed conditions.

5. Documentation:

Documenting the problem, the corrective steps taken, and the results of the testing is important for future references.

This helps build a knowledge base that can be useful for troubleshooting similar issues in the future and aids in continuous improvement.

### Tools for fixing bugs –

**1. Debugger**

Visual Studio Debugger: An integrated debugger within Visual Studio that supports .NET languages, C++, and more.

PyCharm Debugger: A Python-specific debugger in the PyCharm IDE, offering features like breakpoints, variable inspection, and conditional debugging.

**2. Automated Testing Tools**

pytest: A popular testing framework for Python, supporting unit tests and complex functional tests.

JUnit: A widely used testing framework for Java applications.

Selenium: Primarily for automating web application testing.

**3. Version Control Systems with Issue Tracking**

Git + GitHub Issues: Git is a version control system, while GitHub provides issue tracking and collaboration features.

GitLab: A Git-based platform with built-in issue tracking, CI/CD, and project management features. In this way there many more tools are available for fixing bugs.

## 2. Adaptive maintenance

Adaptive maintenance is a type of software maintenance focused on modifying a software system to accommodate changes in its environment. These changes can include updates in operating systems, hardware, databases, network protocols, or third-party services, which necessitate adjustments to keep

the software functioning as intended. It ensures that the software continues to work properly when the external environment evolves, without changing its core functionality or performance.

**Characteristics of Adaptive Maintenance:**

I. Environment-Driven –

Adaptive maintenance responds to changes in the external environment of the software. These changes may arise from-

i. Hardware upgrades (e.g., moving from one server type to another).

ii. Changes in operating systems or new OS versions.

iii. Database management system updates.

iv. Third-party library updates or discontinuation.

v. Changes in networking infrastructure (e.g., IP address changes, new protocols).

II. Compatibility –

The primary goal is to ensure compatibility between the software and the new environment. It doesn't add new features but adapts the software so it can continue functioning within the altered ecosystem.

III. Proactive and Reactive –

Proactive: Sometimes organizations can anticipate changes (e.g., when an OS end-of-life is announced), allowing them to plan ahead for the maintenance.

Reactive: In other cases, adaptive maintenance may be required urgently when unforeseen changes cause the system to malfunction or become incompatible.

## 3. Perfective maintenance

Perfective maintenance is a type of software maintenance focused on improving or enhancing a system's performance, functionality, or usability based on user feedback or evolving business requirements. It addresses aspects of the system that are functioning correctly but can be optimized or refined for better user experience or efficiency. Unlike corrective maintenance, which deals with fixing errors, perfective maintenance is about making the software "better" in response to new needs or opportunities.

### Characteristics of Perfective Maintenance:

1. User-Centric Improvements –

Perfective maintenance often arises from feedback from end-users who request enhancements to improve their experience or make the software more aligned with their needs. These enhancements can include adding new features, improving the user interface, or increasing system efficiency.

2. Proactive Refinements –

While some perfective maintenance is reactive to user feedback, much of it is proactive. It involves refining the system to keep it up-to-date with evolving industry standards, business goals, or technology trends.

3. Functional and Performance Enhancements –

Functional Enhancements: Adding new features, adjusting workflows, or enhancing existing features to meet new business requirements.

Performance Enhancements: Improving the system's speed, resource utilization, scalability, or security to provide better overall performance without changing the core functionality.

4. Focus on Longevity –

Perfective maintenance is about ensuring that a software system remains relevant, usable, and efficient over time.

## Version Control Systems: -

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. A version control system implements or is directly integrated with four major capabilities:

(1) A project database (repository) that stores all relevant configuration objects.

(2) A version management capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions).

(3) A make facility that enables you to collect all relevant configuration objects and construct a specific version of the software.

(4) An issue tracking (also called bug tracking) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

A number of version control systems establish a change set—a collection of all changes (to some baseline configuration) that are required to create a specific version of the software.

Dart notes that a change set "captures all changes to all files in the configuration along with the reason for changes and details of who made the changes and when."

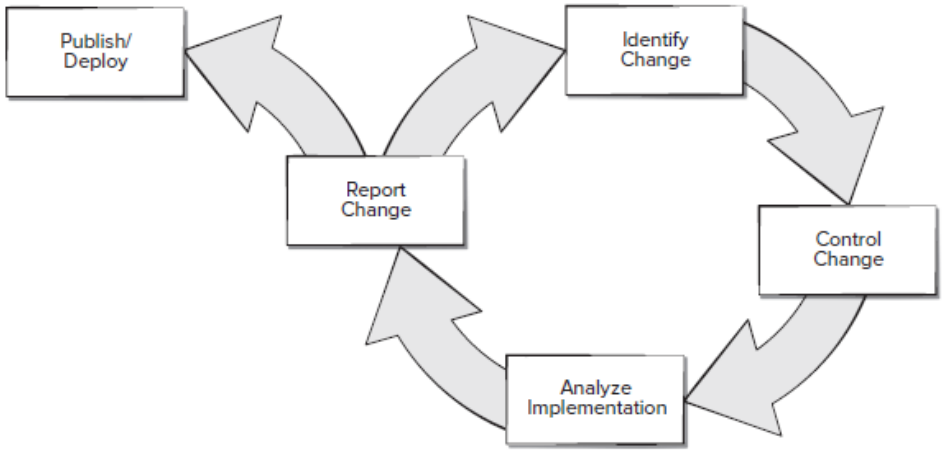
## Software configuration management (SCM): -

Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process.

### Typical SCM work flow:

**FIGURE 22.1**

**Software configuration management work flow**




Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change, (3) ensure that change is being properly implemented, and (4) report changes to others who may have an interest.

Software configuration management is a set of tracking and control activities that are initiated when a software engineering project begins and terminates only when the software is taken out of operation.

The first law of system engineering states: “No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle.”

There are four fundamental sources of change:

- New business or market conditions dictate changes in product requirements or business rules.
- New stakeholder needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
- Reorganization or business growth or downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

## Elements of a Configuration Management System:

**Component elements –** A set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.

**Process elements –** A collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software.

**Construction elements –** A set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.

**Human elements –** A set of tools and process features (encompassing other CM elements) used by the software team to implement effective SCM.

These elements are not mutually exclusive. For example, component elements work in conjunction with construction elements as the software process evolves.

## Software Configuration Items:

A software configuration item as information that is created as part of the software engineering process. An SCI could be considered to be a single section of a large specification or one test case in a large suite of tests. An SCI is all or part of a work product (e.g., a document, an entire suite of test cases, a named program component, or a software tool).

## Configuration Object

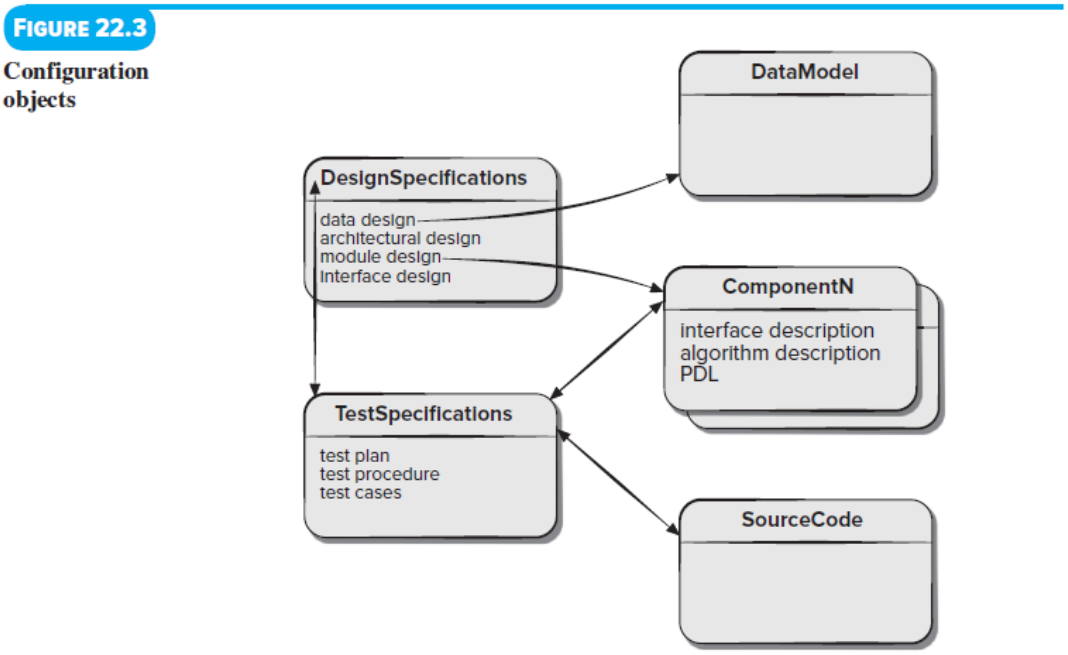



FIGURE 22.3 Configuration objects

An SCIs are organized to form configuration objects that may be cataloged in the project database with a single name. A configuration object has a name, attributes, and is "connected" to other objects by relationships. Referring to above figure, the configuration objects, DesignSpecification, DataModel, ComponentN, SourceCode, and TestSpecification are each defined separately. However, each of the object is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is, DataModel and ComponentN are part of the object DesignSpecification. A double-headed straight arrow indicates an interrelationship. If a change were made to the SourceCode object, the interrelationships enable you to determine what other objects (and SCIs) might be affected.

## The SCM Repository:

The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner. It provides the obvious functions of a modern database management system by ensuring data integrity, sharing, and integration.

The repository is defined in terms of a meta-model. The meta-model determines how information is stored in the repository, how data can be accessed by tools and viewed by software engineers, how well data security and integrity can be maintained, and how easily the existing model can be extended to accommodate new needs.

## Software refactoring: -

Software refactoring (also known as restructuring) modifies source code and/or data in an effort to make it amenable (easily controlled) to future changes. In general, refactoring does not modify the overall program architecture. It tends to focus on the design details of individual modules and on local data structures defined within modules.

Software refactoring is the process of restructuring existing code without changing its external behavior. The main goal of refactoring is to improve code quality, readability, maintainability, and performance.

### Data Refactoring –

Data definitions to achieve consistency among data item names or physical record formats within an existing data structure or file format.

Another form of redesign, called data name rationalization, ensures that all data naming conventions conform to local standards

### Code Refactoring –

Code refactoring is performed to yield a design that produces the same function but with higher quality than the original program.

## Common Refactoring and code quality improvement Techniques –

### 1. Extract Method:

Extract a portion of code from a large method and place it in a separate method with a descriptive name.

It improves readability and promotes code reuse.

### 2. Rename Variables and Methods:

Change names of variables, methods, or classes to more meaningful ones.

It improves readability and conveys purpose or function.

### 3. Consolidate Duplicate Code:

Identify and combine duplicate code blocks into a single method.

It reduces redundancy and minimizes maintenance effort.

Replace Magic Numbers with Constants:

Replace hard-coded values with named constants.

It increases code clarity and makes it easier to update values.

Improved Maintainability:

When duplicate code is consolidated, it exists in only one place, making it easier to maintain. Any future changes only need to be made once, reducing the risk of missing updates in one of the duplicated sections.

This reduces the possibility of bugs being introduced when you update one block of code but forget to update other duplicate blocks.

### 4. Introduce Parameter Object:

When multiple parameters are passed frequently together, encapsulate them in an object.

It reduces the number of parameters and increases flexibility for passing data.

### 5. Encapsulate Field:

Make class variables private and provide getter/setter methods for controlled access.

It increases encapsulation and prevents unauthorized modification of data.

### 6. Code Reviews:

Regular peer reviews allow for catching errors, improving code readability, and sharing knowledge.

Reviews help identify areas for refactoring and enforce consistent code style.

### 7. Automated Testing:

Writing unit tests ensures that code changes don't break existing functionality.

Code refactoring is safer with a comprehensive test suite in place.

### 8. Continuous Integration (CI):

Use CI tools (e.g., Jenkins, GitHub Actions) to automate the process of running tests, checking code quality, and building the software as code is integrated into the main branch.

This practice ensures early detection of defects and keeps the codebase in a deployable state.

### 9. Code Formatting and Style Guides:

Adhering to a consistent style, such as PEP 8 for Python, improves readability.

### 10. Modularization:

Break large monolithic code into smaller, reusable, and focused modules.

Increases readability and allows for independent testing and development.

11. Use Refactoring Tools:

Many modern IDEs (like PyCharm, Visual Studio Code) offer built-in refactoring tools, including method extraction, renaming, and automatic code formatting.

These tools simplify the refactoring process and help avoid manual errors.

12. Monitor Code Metrics:

Track metrics such as cyclomatic complexity, code coverage, and code duplication to assess code quality.

13. Documentation:

Write descriptive comments, docstrings, and external documentation for complex logic, APIs, and modules.

Good documentation improves maintainability and aids other developers in understanding the codebase.

**Examples:**

1. Encapsulate Field

**Before –**

```
class Employee:
        def __init__(self, name, salary):
                self.name = name
                self.salary = salary
```

**After –**

```
class Employee:
   def __init__(self, name, salary):
      self._name = name  # Encapsulated with an underscore prefix
      self._salary = salary  # Encapsulated with an underscore prefix


   # Getter for name
   @property
   def name(self):
      return self._name


   # Setter for name
```

```
    @name.setter
    def name(self, value):
        self._name = value

    # Getter for salary
    @property
    def salary(self):
        return self._salary

    # Setter for salary
    @salary.setter
    def salary(self, value):
        self._salary = value
```

2. Introduce Parameter Object

**Before –**

```
def student_details(name, rollno, branch, semester, department, contact_no ):
```

**After –**

```
class Student_Details:
    def __init__(self, name, rollno, branch, semester, department, contact_no):
        self.name = name
        self.rollno = rollno
        self.branch = branch
        self.semester = semester
        self.department = department
        self.contact_no = contact_no
def student_details(student):
    print(f"Name: {student.name}")
    print(f"Roll No: {student.rollno}")
```

```
    print(f"Branch: {student.branch}")
    print(f"Semester: {student.semester}")
    print(f"Department: {student.department}")
    print(f"Contact No: {student.contact_no}")

student = StudentDetails("Alice", 101, "Computer Science", 5, "Engineering", "123-456-7890")
student_details(student)
```

3. Replace Magic Numbers with Constants

**Before –**

```
def calculate_circumference(radius):
        return 2 * 3.14 * radius
```

**After –**

```
PI = 3.14 # Defined constant for the magic number
def calculate_circumference(radius):
        return 2 * PI * radius
```

4. Rename Variables and Methods

**Before –**

```
def calc_area(x, y):
        return x * y
```

**After –**

```
def calculate_area(width, height):
    return width * height
```

## Legacy system modernization and migration strategies: -

Legacy system modernization and migration refer to the processes involved in updating, re-architecting, or transforming older, outdated systems to improve their functionality, performance, scalability, and maintainability. Legacy systems are often critical to business operations, but they can become increasingly difficult and costly to maintain over time due to outdated technology, lack of vendor support, and limitations in adapting to modern requirements.

Legacy system modernization and migration are critical processes for organizations looking to improve efficiency, reduce technical debt, and adapt to modern technologies. There are several strategies to modernize and migrate legacy systems depending on the business needs, system complexity, and budget.

## Strategies and approaches for modernizing and migrating legacy systems:

### 1. Encapsulation

Keep the legacy system operational but provide new functionality by encapsulating it with modern APIs or web services.

Wrap the existing system with an API layer, allowing other systems to interact with it using modern protocols (e.g., REST, SOAP).

Use this strategy when the legacy system still meets the core requirements but lacks flexibility, interoperability, or integration capabilities.

Reusing legacy system components by exposing their functions as services through APIs, wrappers, or integration layers.

When to use: When the core functionality of the legacy system is still valuable, but the system needs to interact with modern applications or services.

### 2. Rehosting (Lift and Shift)

Move the legacy system to a new environment (often to the cloud) without making significant changes to the underlying architecture.

Use tools to migrate the system to cloud infrastructure or new servers with minimal modification.

Use this strategy when the current hardware is outdated or costly, but the software still meets business needs.

Moving the existing application to a new infrastructure environment (e.g., from on-premises to the cloud) with minimal or no changes to the application's code.

When to use: When an organization needs quick migration to a new platform, especially when moving to the cloud for better scalability or lower operational costs.

### 3. Replatforming

Migrate the legacy system to a modern platform with minor changes to optimize for the new environment.

Move the system to a modern operating system, database, or application server while keeping most of the application logic intact.

Use this strategy when the legacy system needs to run on newer technology without a full re-architecture.

Moving the legacy system to a modern platform with minimal code changes, while optimizing for some cloud-native or modern features (e.g., using managed databases or cloud storage).

When to use: When a balance between minimal disruption and the need for system optimization is required.

### 4. Refactoring (Rearchitecting)

Rework and improve the existing codebase to make it more maintainable and scalable without changing its functionality.

Update the system's architecture (e.g., moving to microservices or serverless architecture) and rewrite parts of the code to eliminate dependencies on outdated technologies.

Use this strategy when the system needs substantial improvements in performance, scalability, or maintainability.

Improving the internal structure of the system code without changing its external behavior. This involves restructuring code to improve maintainability, performance, and scalability.

When to use: When the legacy system's architecture and code are too complex, or when technical debt is high but the core functionality is still valuable.

### 5. Rebuilding (Redesign and Rebuild)

Rewrite the system from scratch, using modern technology and best practices.

Analyze the existing system's requirements, design a new system architecture, and build it from the ground up.

Use this strategy when the current system no longer meets business needs, is difficult to maintain, or relies heavily on obsolete technology.

Rebuilding the application from scratch using modern technologies and approaches, often using cloud-native development frameworks. A development framework is a set of tools, libraries, and conventions that help developers to create applications more quickly and efficiently.

When to use: When the legacy system is outdated, difficult to maintain, and unable to meet business needs.

### 6. Replacement

Replace the legacy system with a new, off-the-shelf software solution or custom-built application.

Migrate data from the legacy system to the new one and integrate the new solution into the existing IT landscape.

Use this strategy when a commercial product meets the business needs and avoids the complexities of building a custom solution.

Replacing the legacy system entirely with a new off-the-shelf or custom-built solution. Off-the-shelf solutions are ready-made products that are available to purchase and use immediately. They are intended to serve a broad audience with common needs. Custom-built solutions are designed to meet the specific needs of a company or user

When to use: When the existing system is obsolete and cannot meet current business needs, and the cost of rebuilding or refactoring is too high.

### 7. Retiring

Phasing out the legacy system and not replacing it.

When to use: When the system is no longer needed or the functionality is duplicated in other systems.

### 8. Hybrid Approaches

Combining several modernization strategies, such as rehosting part of a system while rewriting critical components.

When to use: When an organization needs to modernize in phases or when different parts of a system have different modernization requirements.