SOFTWARE DESIGN & DEVELOPMENT METHODOLOGIES

UNIT 5: Large-Scale Software Development, System Software Perspective, and UNIX/Linux Programming Environment

5.1 Introduction

The earlier units introduced the foundations of software engineering, design, coding, planning, maintenance, configuration management, and testing. Unit 5 brings these ideas together and applies them to a wider and more practical context: **large-scale software development** and the **software development environment**.

In small programming exercises, one student may write one program, compile it, run it, and submit it. But professional software development is very different. A large-scale software project may involve many people, many modules, many users, several technologies, multiple releases, strict testing, documentation, maintenance, version control, deployment environments, and long-term support.

This unit also introduces the computer from a **system software perspective**. You must understand that software does not run in isolation. It runs on hardware, uses an operating system, depends on memory, storage, processors, compilers, interpreters, device drivers, networks, and system utilities. For this reason, a software developer should know not only programming syntax, but also the environment in which programs are created and executed.

A major practical part of this unit is UNIX/Linux. UNIX and Linux are widely used in software development, servers, networking, cloud systems, embedded systems, cybersecurity, and system programming. You should learn how to work in a UNIX/Linux environment, install a programming setup, and create simple Java programs.

PART I: SOFTWARE DEVELOPMENT METHODOLOGIES IN PRACTICE

5.2 Meaning of Software Development Methodology

A software development methodology is an organized approach used to plan, develop, test, deliver, and maintain software.

A methodology gives guidance on:

- how requirements are gathered,
- how design is performed,
- how coding is organized,
- how testing is done,
- how progress is tracked,
- how changes are handled,
- how teams communicate.

A methodology is not merely a diagram. It is a disciplined way of working.

5.3 Why Methodologies Are Needed

Software development involves uncertainty and complexity. Without a methodology, teams may face confusion, duplicated work, poor communication, missed requirements, late delivery, and low-quality software.

A methodology helps answer:

- What should we do first?
- Who is responsible?
- How do we handle changes?
- When do we test?
- How do we know progress?
- How do we deliver safely?

For small classroom programs, a formal methodology may seem unnecessary. But for real projects, especially large systems, methodology is essential.

5.4 Common Software Development Methodologies

The earlier units discussed several models. In Unit 5, it is important to understand how these methodologies may be used in real project development.

5.4.1 Waterfall Methodology

The Waterfall methodology follows a sequential flow:

Requirements ⟶ Design ⟶ Coding ⟶ Testing ⟶ Deployment ⟶ Maintenance

It is useful when requirements are stable and well understood.

Example:

A government department wants a reporting system based on fixed legal forms. If the requirements are unlikely to change, Waterfall may be suitable.

Its weakness is that it does not handle changing requirements easily.

5.4.2 Prototyping Methodology

Prototyping is useful when users cannot clearly describe their needs at the beginning. A prototype gives users something visible to evaluate.

Example:

A college wants a new student portal, but staff and students are unsure how screens should look. A prototype can help clarify requirements.

5.4.3 Spiral Methodology

The Spiral model is risk-driven. It is useful for large, complex, and high-risk projects.

Example:

A national identity system or banking platform may use risk analysis repeatedly because security, reliability, integration, and performance risks are serious.

5.4.4 Iterative and Incremental Methodology

In iterative development, the system is built step by step. Each iteration improves or adds functionality.

Example:

- Release 1: Login and profile
- Release 2: Course registration
- Release 3: Fee payment
- Release 4: Reports

This allows users to benefit from early versions while development continues.

5.4.5 Agile Methodology

Agile emphasizes short development cycles, customer feedback, working software, and responsiveness to change.

A common Agile rhythm is:

Backlog ⟶ Sprint Planning ⟶ Development ⟶ Testing ⟶ Review ⟶ Retrospective

Agile is useful when requirements change often or when user feedback is essential.

5.4.6 Object-Oriented Methodology

Object-oriented development organizes software around objects and classes.

Example:

- Class: Student
- Class: Course
- Class: Lecturer
- Class: Payment
- Class: Result

This method is useful for modelling real-world entities and improving reuse and maintainability.

PART II: LARGE-SCALE SOFTWARE PROJECTS

5.5 Meaning of a Large-Scale Software Project

A large-scale software project is a software project that is too big, complex, risky, or long-term to be handled informally by one person or a very small team.

Large-scale projects usually have:

- many users,
- many modules,
- multiple teams,
- large databases,
- long development schedules,
- multiple releases,
- strict security needs,
- high reliability expectations,
- complex testing,
- continuous maintenance.

5.6 Examples of Large-Scale Software Projects

Examples include:

- Banking systems,
- Hospital management systems,
- Airline reservation systems,
- National identity systems,
- University management systems,
- Tax collection systems,
- Telecom billing systems,
- E-commerce platforms,
- Learning management systems,
- Government service portals.

These systems are large not only because they contain many lines of code, but because they affect many users and must operate reliably.

5.7 Characteristics of Large-Scale Software Projects

5.7.1 Large number of stakeholders

A large system may involve customers, users, managers, administrators, technical staff, auditors, regulators, and support teams.

Example:

A university system may involve students, lecturers, finance staff, registrars, librarians, examination officers, and system administrators.

5.7.2 Complex requirements

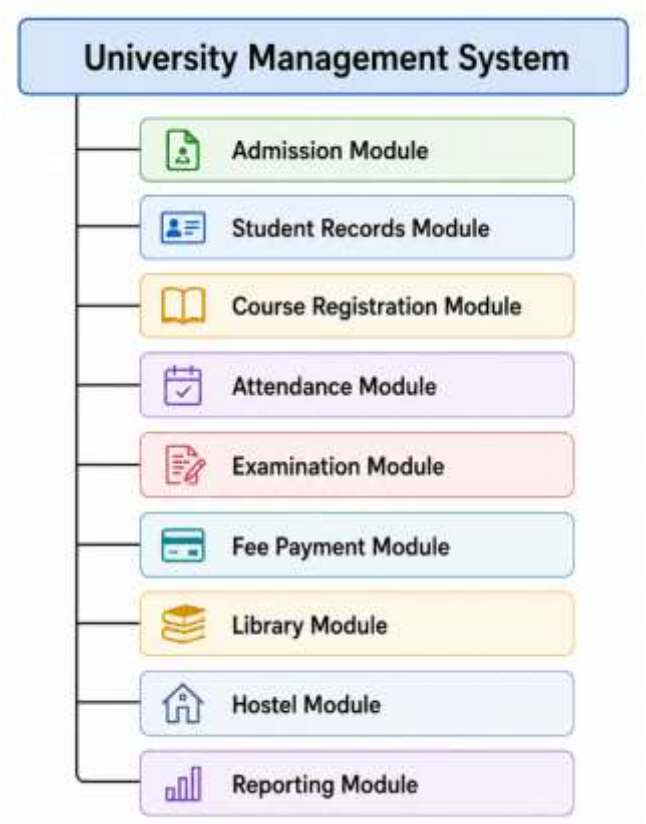
Requirements may be numerous and sometimes conflicting.

Finance staff may want strict controls. Students may want fast access. Administrators may want detailed reports. Security staff may want strong authentication.

5.7.3 Multiple modules

Large systems are divided into modules.

Example:



5.7.4 Team coordination

Many developers may work on different parts of the system. This requires communication, standards, version control, documentation, and integration planning.

5.7.5 High testing effort

Large projects require unit testing, integration testing, system testing, performance testing, security testing, regression testing, and user acceptance testing.

5.7.6 Long-term maintenance

Large systems usually remain in use for years. They must be updated as laws, policies, technologies, and user needs change.

5.8 Challenges in Large-Scale Software Projects

Large projects are difficult because of several challenges.

1. **Communication difficulty:** As the number of people increases, communication becomes harder. Misunderstandings become more likely.
2. **Requirement changes:** Users often change their minds after seeing early versions.
3. **Integration problems:** Modules developed by different teams may not work together correctly at first.
4. **Schedule pressure:** Large projects often have deadlines from management, customers, or public commitments.
5. **Quality control:** It is harder to maintain consistent quality across many teams and modules.
6. **Security and reliability:** Large systems often handle sensitive data and must be reliable.
7. **Maintenance complexity:** As the system grows, changes become more difficult because one change may affect many modules.

PART III: DEVELOPMENT ENVIRONMENT FOR LARGE-SCALE SOFTWARE PROJECTS

5.9 Meaning of Development Environment

A development environment is the collection of hardware, software, tools, processes, and configurations used to create, test, and maintain software.

It may include:

- Computers,
- Operating systems,
- Programming languages,
- Compilers,
- Interpreters,
- Code editors,
- Integrated development environments,
- Version control systems,
- Databases,

- Build tools,
- Testing tools,
- Documentation tools,
- Deployment servers.

5.10 Why Development Environment Matters

A poor development environment slows work and increases errors.

A good development environment supports:

- Coding,
- Compiling,
- Debugging,
- Testing,
- Collaboration,
- Version control,
- Deployment,
- Monitoring.

For large-scale projects, the environment must be organized and standardized. If each developer uses completely different tools and configurations, integration problems become common.

5.11 Components of a Large-Scale Development Environment

i. Source code editor or IDE

An editor or IDE allows developers to write and manage code.

Examples of IDE functions include:

- Syntax highlighting,
- Code completion,
- Debugging,
- Project navigation,
- Build support.

ii. Compiler or interpreter

A compiler translates source code into executable form. An interpreter executes code directly or through intermediate representation.

For Java, source code is compiled into bytecode using javac, then run on the Java Virtual Machine using java.

iii. Version control system

Version control tracks changes to source code and project files.

It helps teams answer:

- Who changed this?
- When was it changed?
- Why was it changed?
- Can we restore an earlier version?

iv. Build tools

Build tools automate compiling, packaging, and preparing software for release.

v. Testing tools

Testing tools help run unit tests, integration tests, regression tests, and performance tests.

vi. Database systems

Many large applications store data in databases.

vii. Documentation tools

Documentation tools help maintain user manuals, technical guides, API references, and installation instructions.

viii. Issue tracking systems

Issue tracking systems record bugs, feature requests, tasks, and change requests.

ix. Deployment environment

Deployment tools and servers help move software from development to production.

5.12 Using Methodologies to Develop Large Software Projects

A methodology guides the development environment and team activities.

For example, in an Agile project:

- Requirements are placed in a backlog,
- Work is selected during sprint planning,
- Developers code and test features,
- Code is reviewed,
- Automated tests are run,
- Working increments are demonstrated,
- Feedback updates the backlog.

In a Waterfall project:

- Requirements are documented first,
- Design is completed,
- Coding follows,
- Testing occurs after implementation,
- Maintenance begins after deployment.

In a Spiral project:

- Risks are identified early,
- Prototypes may be built,
- High-risk components are tested first,
- Planning occurs repeatedly.

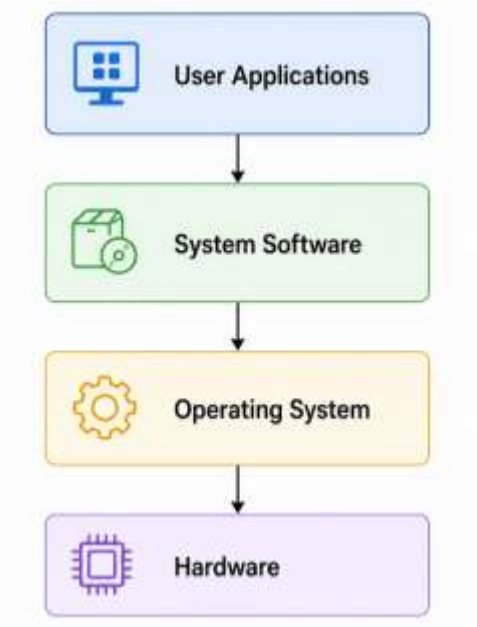
The main point is that large projects cannot rely on casual programming. They need a combination of methodology, tools, standards, and management.

PART IV: COMPUTER FROM SYSTEM SOFTWARE PERSPECTIVE

5.13 Computer as a System for Software Development

From a system software perspective, a computer is not just a machine that runs applications. It is a layered system that supports program execution.

A simplified view is:



Software developers should understand this structure because their programs depend on system resources.

5.14 System Software

System software is software that manages hardware resources and provides services for application software.

Examples include:

- operating systems,
- compilers,
- assemblers,
- interpreters,
- device drivers,
- linkers,
- loaders,
- file systems,
- utility programs.

Application software solves user problems. System software supports the running and development of application software.

5.15 Application Software versus System Software

Aspect	System Software	Application Software
Purpose	Manages computer resources	Solves user tasks
Examples	Operating system, compiler, driver	Word processor, browser, payroll system
User interaction	Often indirect	Usually direct
Dependency	Runs close to hardware	Depends on system software

A student management system is application software. The operating system, Java compiler, and file system supporting it are system software.

PART V: COMPONENTS OF A COMPUTER FOR SOFTWARE SYSTEM DEVELOPMENT

5.16 Major Components of a Computer

Software development depends on several computer components.

i. Central Processing Unit

The CPU executes instructions. It performs arithmetic, logic, control, and instruction processing. When a Java program runs, the CPU ultimately executes machine-level instructions produced by the runtime environment.

ii. Main Memory

Main memory, or RAM, temporarily stores programs and data while they are being used.
If a program processes many records, it needs enough memory.

iii. Secondary Storage

Secondary storage stores data permanently.

Examples:

- Hard disk,
- Solid-state drive,

- USB storage,
- Optical media.

Source code files, compiled files, databases, logs, and documents are stored here.

iv. Input Devices

Input devices allow users to enter data.

Examples:

- Keyboard,
- Mouse,
- Scanner,
- Microphone,
- Touchscreen.

v. Output Devices

Output devices present results.

Examples:

- Monitor,
- Printer,
- Speaker.

vi. Network Interface

Network interfaces allow computers to communicate. Many large software systems depend on networks for database access, web access, APIs, cloud services, and client-server communication.

vii. Operating System

The operating system manages hardware and provides services to programs.

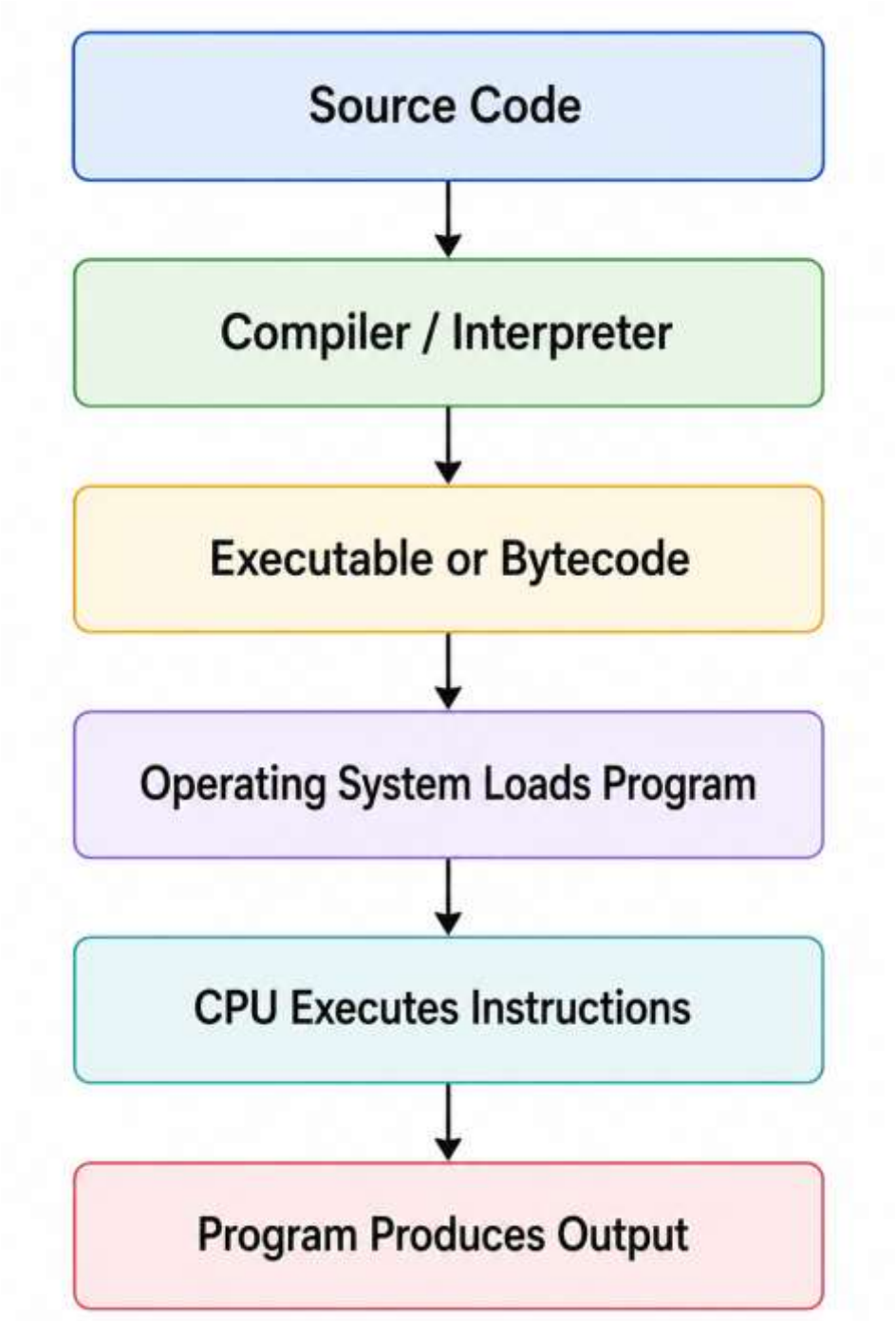
It handles:

- Process management,
- Memory management,
- File management,
- Device management,
- Security,

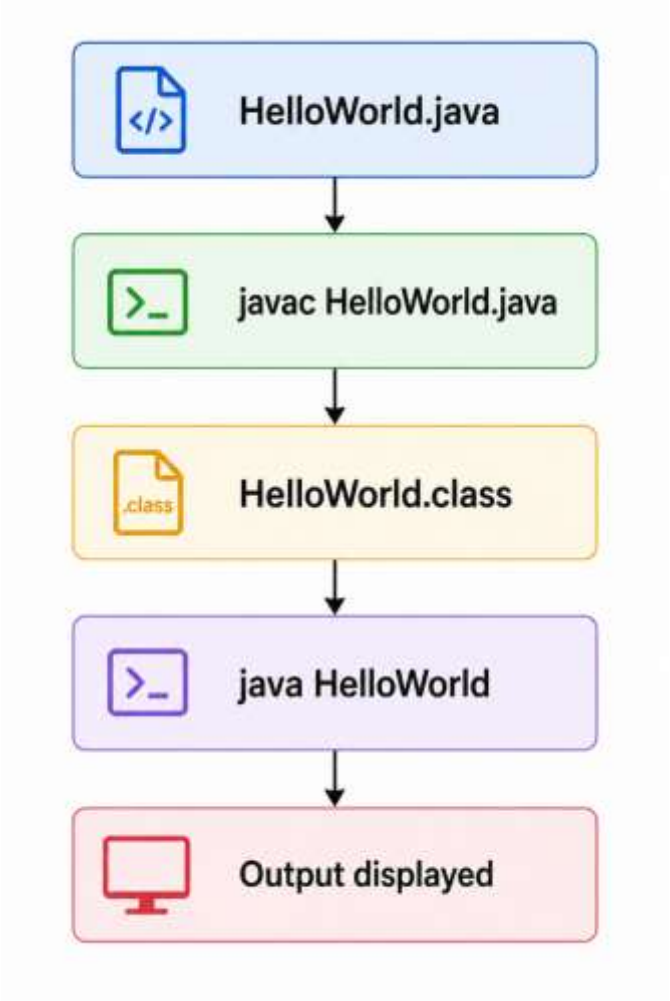
- Networking,
- User interface.

5.17 How a Program Runs

A simple explanation of program execution is:



For Java:



PART VI: WRITING PROGRAMS TO IMPLEMENT COMPUTER SYSTEM TASKS

5.18 What Are Computer System Tasks?

Computer system tasks are operations performed by programs using system resources.

Examples include:

- Reading input,
- Writing output,
- Creating files,
- Reading files,
- Processing data,
- Using memory,

- Communicating over a network,
- Handling errors,
- Interacting with the operating system.

5.19 Simple Example: Reading Input and Producing Output

A basic Java program can ask for a student name and display a greeting.

```
import java.util.Scanner;

public class StudentGreeting {

   public static void main(String[] args) {

      Scanner input = new Scanner(System.in);

      System.out.print("Enter student name: ");

      String name = input.nextLine();

      System.out.println("Welcome, " + name + "!");

   }

}
```

This program uses:

- Keyboard input,
- Memory to store the name,
- Console output,
- Java runtime environment.

5.20 Example: Simple Calculation

```
import java.util.Scanner;

public class FeeCalculator {

   public static void main(String[] args) {

      Scanner input = new Scanner(System.in);

      System.out.print("Enter tuition fee: ");

      double tuitionFee = input.nextDouble();

      System.out.print("Enter library fee: ");

      double libraryFee = input.nextDouble();

      double totalFee = tuitionFee + libraryFee;

      System.out.println("Total fee is: " + totalFee);

   }

}
```

This simple program introduces system interaction through input and output.

PART VII: UNIX/LINUX FOR SYSTEM SOFTWARE DEVELOPMENT

5.21 Meaning of UNIX/Linux

UNIX is a family of operating systems originally designed for multi-user, multitasking computing. **Linux** is a UNIX-like operating system widely used today in servers, cloud systems, embedded systems, development environments, cybersecurity, and education.

For teaching purposes, Linux is often used because it is widely available and supports many programming tools.

5.22 Why UNIX/Linux Is Important for Software Development

UNIX/Linux is important because it provides:

- Command-line tools,
- Strong file system organization,
- Programming language support,
- Scripting facilities,
- Networking tools,
- Process management,
- Permissions and security,
- Development utilities,
- Server environment compatibility.

Many real-world systems, especially servers and cloud applications, run on Linux.

5.23 Features of UNIX/Linux Useful to Developers

1. **Multi-user capability**

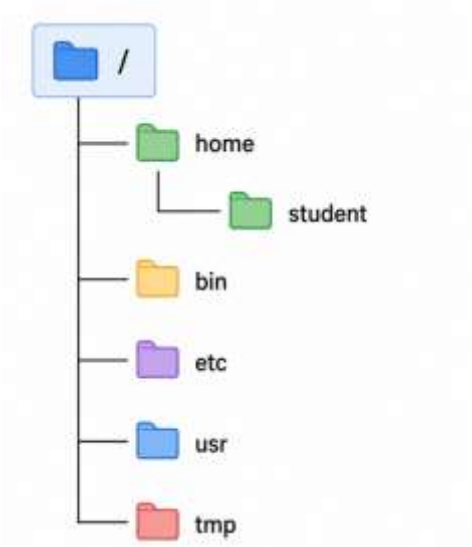
Many users can use the system with separate accounts and permissions.

2. **Multitasking**

The system can run many processes at the same time.

3. **Hierarchical file system**

Files and directories are organized in a tree structure.



4. **Command-line interface**

The terminal allows users to issue commands directly.

5. **Security and permissions**

Files have permissions controlling who can read, write, and execute them.

6. **Development tools**

Linux supports compilers, interpreters, editors, debuggers, version control, and scripting.

PART VIII: UNIX/LINUX DEVELOPMENT ENVIRONMENT

5.24 The Linux Terminal

The terminal is a text-based interface used to communicate with the operating system.

A command usually has this form:

```
command options arguments
```

Example:

```
ls -l Documents
```

This means list files in the Documents directory using long format.

5.25 Basic Linux Commands

1. **pwd:** Shows the current directory.

Example: pwd

2. **ls:** Lists files and directories.

Example: ls

3. **cd:** Changes directory.

Example: cd Documents

4. **mkdir:** Creates a directory.

Example: mkdir JavaPrograms

5. **touch:** Creates an empty file.

Example: touch notes.txt

6. **cp:** Copies files.

Example: cp file1.txt file2.txt

7. **mv:** Moves or renames files.

Example: mv oldname.txt newname.txt

8. **rm:** Removes files.

Example: rm oldfile.txt

NOTE: The deleted files from command line may not always go to a recycle bin.

9. **Cat:** Displays file contents.

Example: cat notes.txt

10. **nano or vim:** Used to edit text files.

Example: nano HelloWorld.java

5.26 File Permissions in Linux

Linux permissions control access to files. Permissions are commonly represented as:

- r = read
- w = write
- x = execute

They apply to:

- ✓ owner
- ✓ group
- ✓ others

Example output:

```
-rw-r--r-- 1 student student 120 HelloWorld.java
```

This means:

- Owner can read and write,
- Group can read,
- Others can read,
- No one has execute permission for this file.

5.27 Why Permissions Matter

Permissions protect files from unauthorized access or accidental modification. For software development, permissions are important when running scripts, protecting source files, managing server programs, and controlling system resources.

PART IX: INSTALLING UNIX/LINUX PROGRAMMING ENVIRONMENT

5.28 What Is a Programming Environment?

A programming environment includes all tools needed to write, compile, run, and debug programs.

For Java development on Linux, the environment usually includes:

- Linux operating system,
- terminal,
- text editor or IDE,
- Java Development Kit,
- compiler,
- runtime environment.

5.29 Java Development Kit

The Java Development Kit, or JDK, includes tools needed to develop Java programs.

Important tools include:

- javac: Java compiler
- java: Java program launcher

5.30 Checking Java Installation

A student can check whether Java is installed by running:

- ❖ java -version

To check the compiler:

- ❖ javac -version

If these commands show version information, Java is installed.

5.31 Installing Java on Linux

The exact installation command depends on the Linux distribution.

On Debian-based systems such as Ubuntu, a common approach is:

- sudo apt update
- sudo apt install default-jdk

After installation:

- java -version
- javac -version

On Red Hat-based systems, the package manager may be different, such as dnf or yum.

The general teaching point is that students should know the process:

- Update package list
- Install JDK
- Verify installation
- Create source file
- Compile program
- Run program

PART X: CREATING SIMPLE JAVA PROGRAMS IN UNIX/LINUX

5.32 Creating a Working Directory

Lets begin by creating a directory for Java work.

- mkdir JavaPrograms
- cd JavaPrograms

5.33 Creating the First Java Program

Create a file:

- nano HelloWorld.java

Write the program:

```
public class HelloWorld {

   public static void main(String[] args) {

      System.out.println("Hello, Software Engineering Students!");

   }

}
```

Save the file.

5.34 Compiling the Program

Use the Java compiler:

- javac HelloWorld.java

If compilation succeeds, a file named:

- HelloWorld.class

will be created.

5.35 Running the Program

Run it using:

- java HelloWorld

Expected output:

```
Hello, Software Engineering Students!
```

5.36 Important Java Naming Rule

In Java, if a class is declared as public, the file name must match the class name.

Therefore:

```
public class HelloWorld
```

must be stored in:

- HelloWorld.java

If the names do not match, compilation will fail.

PART XI: MORE JAVA PROGRAM EXAMPLES FOR STUDENTS

5.37 Program to Add Two Numbers

```
import java.util.Scanner;

public class AddNumbers {

   public static void main(String[] args) {

      Scanner input = new Scanner(System.in);

      System.out.print("Enter first number: ");

      int firstNumber = input.nextInt();

      System.out.print("Enter second number: ");

      int secondNumber = input.nextInt();

      int sum = firstNumber + secondNumber;

      System.out.println("Sum = " + sum);

   }

}
```

Compile:

- javac AddNumbers.java

Run:

- java AddNumbers

5.38 Program to Check Pass or Fail

```
import java.util.Scanner;

public class PassFail {

   public static void main(String[] args) {

      Scanner input = new Scanner(System.in);

      System.out.print("Enter marks: ");

      int marks = input.nextInt();

      if (marks >= 40) {

         System.out.println("Result: Pass");

      } else {

         System.out.println("Result: Fail");

      }

   }

}
```

This program demonstrates:

- Input,
- Condition,
- Output.

5.39 Program to Calculate Simple Interest

```
import java.util.Scanner;

public class SimpleInterest {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.print("Enter principal: ");

        double principal = input.nextDouble();

        System.out.print("Enter rate: ");

        double rate = input.nextDouble();

        System.out.print("Enter time: ");

        double time = input.nextDouble();

        double interest = (principal * rate * time) / 100;

        System.out.println("Simple interest = " + interest);

    }

}
```

5.40 Program to Write Output to a File

```
import java.io.FileWriter;

import java.io.IOException;

public class WriteFileExample {

   public static void main(String[] args) {

      try {

         FileWriter writer = new FileWriter("output.txt");

         writer.write("This file was created by a Java program.");

         writer.close();

         System.out.println("File written successfully.");

      } catch (IOException error) {

         System.out.println("An error occurred while writing the file.");

      }

   }

}
```

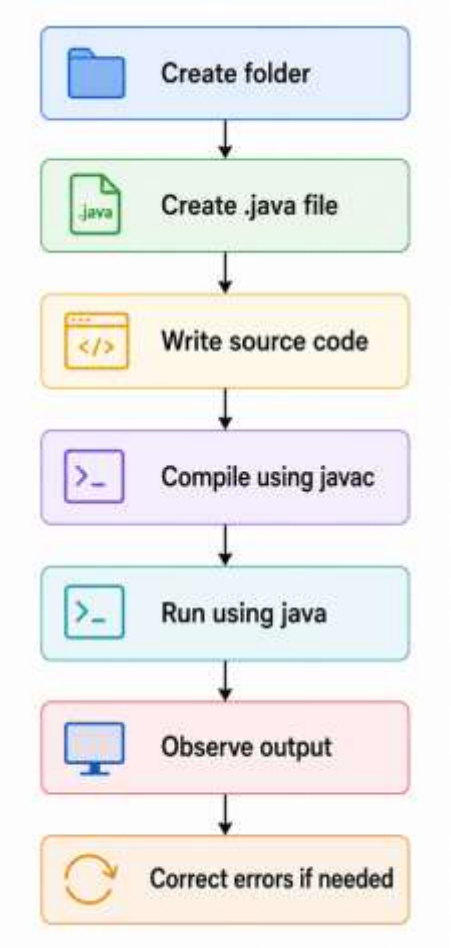
After running the program, check the file:

- cat output.txt

This example connects Java programming with the operating system file system.

PART XII: DEVELOPMENT WORKFLOW IN UNIX/LINUX

5.41 Basic Java Development Workflow



Correct errors if needed

5.42 Example Terminal Session

mkdir JavaPrograms

cd JavaPrograms

nano HelloWorld.java

javac HelloWorld.java

java HelloWorld

Expected output:

Hello, Software Engineering Students!

5.43 Common Compilation Errors

5.43.1 File name and class name mismatch

Problem:

public class HelloWorld

stored in:

hello.java

Correction:

File should be HelloWorld.java

5.43.2 Missing semicolon

Problem:

System.out.println("Hello")

Correction:

System.out.println("Hello");

5.43.3 Wrong capitalization

Java is case-sensitive.

System

is not the same as:

system

5.43.4 Missing braces

Every opening brace { must have a matching closing brace }.

PART XIII: INTEGRATED LARGE-SCALE PROJECT EXAMPLE

5.44 Example: Developing a College Management System

Let us connect the whole unit using a large software project.

A college wants a system to manage:

- Student admission,
- Course registration,
- Fee payment,
- Attendance,
- Examinations,
- Library records,
- Reports.

5.45 Choosing Methodology

If requirements are mostly stable, Waterfall may be used.

If users are unsure and need to see screens early, prototyping may help.

If the project is large and risky, Spiral may help manage risk.

If the college wants gradual delivery, Agile or iterative development may be suitable.

A practical approach may combine methods. For example, the team may use Agile sprints while still maintaining formal documentation for important requirements.

5.46 Development Environment

The team may use:

- Linux servers,
- Java programming language,
- database system,
- version control,
- issue tracking,
- testing tools,
- documentation tools,
- deployment scripts.

5.47 System Software Perspective

The application depends on:

- Hardware
- Operating system
- Java runtime
- Database server
- Network services
- File system
- Security services

If any layer fails, the application may fail.

5.48 Maintenance and Growth

After release, the system may need new modules, new reports, new grading rules, security updates, and performance improvements. Therefore, it must be designed and managed as a long-term software product.

Short Revision Summary

Software development methodologies provide structured ways to build software. Large-scale software projects require careful planning, modular design, disciplined coding, testing, documentation, version control, and maintenance. Such projects usually involve many stakeholders, teams, modules, and technologies.

A development environment includes the tools and systems needed to write, build, test, and maintain software. It may include operating systems, IDEs, compilers, databases, testing tools, version control systems, documentation tools, and deployment platforms.

From a system software perspective, a computer consists of hardware managed by operating systems and system software. Application programs depend on system software to use memory, files, devices, processors, and networks.

UNIX/Linux is important for software development because it provides a powerful command-line environment, strong file system, permissions, development tools, scripting, and server capabilities. Students should know basic Linux commands and understand how to install and use a Java programming environment.

The practical workflow for Java on Linux is simple:

Write source code → Compile with javac → Run with java → Observe output → Correct errors

The central lesson of Unit 5 is that professional software development requires more than programming knowledge. It requires understanding methodologies, large-scale project organization, development environments, system software, and practical command-line programming skills.