CHAPTER 1

INTRODUCTION

1.1 OVERVIEW OF THE PROJECT

The Payroll Management System is a comprehensive Java-based full-stack application designed to automate and streamline the core HR functions related to employee data management, salary processing, tax calculations, and financial reporting. Built using core Java, JDBC, and SQL, the system follows an organized architecture consisting of model classes, service layers, utility classes, and a database context to manage backend operations effectively.

At the heart of the system lies the Employee module, which supports complete CRUD (Create, Read, Update, Delete) operations for managing employee profiles, including personal details, position, and employment history. The Payroll module automates salary generation based on predefined salary structures, overtime, and deductions, with support for generating pay slips. The Tax module ensures accurate tax computation according to the employee's income and deduction slabs, while the Financial Record module maintains categorized financial transactions like bonuses, expenses, or tax payments.

Custom exceptions handle error scenarios like missing employee records or database failures gracefully, improving overall system reliability and maintainability. Unit testing plays a key role in validating the system's correctness. Using JUnit, the application includes test cases for salary computation, tax calculation, data validation, and payroll processing for multiple employees. This ensures that each module performs accurately under different input conditions and business rules.

1.2 END TO END JAVA DEVELOPMENT

Java Full Stack Development refers to the capability of developing an entire web application from the user interface to the database—using Java and its associated technologies. A Java Full Stack Developer works on both the front-end (client side) and back-end (server side) of an application. On the front-end, technologies like HTML, CSS, JavaScript, and frameworks such as Angular or React are used to create interactive user experiences.

On the back-end, core Java, along with frameworks like Spring Boot and Hibernate, handle the business logic, APIs, and communication with databases. A full stack developer also manages data storage using databases like MySQL or MongoDB and may interact with RESTful APIs for seamless client-server communication. This role demands versatility, as developers need to be proficient in multiple layers of software architecture, debugging across the stack, and understanding deployment and system integration. As web applications continue to become more sophisticated, Java Full Stack Developers play a key role in delivering complete and scalable solutions.

1.3 GROWTH AND ADVANCEMENT OF JAVA FULL STACK

The journey of Java Full Stack Development began with simple web applications using Servlets and JSPs (JavaServer Pages), where Java was mostly confined to the server side. As complexity increased, frameworks like Struts and JSF emerged to introduce MVC architecture, promoting code reusability and better separation of concerns. Hibernate revolutionized database interaction by introducing Object-Relational Mapping (ORM), reducing boilerplate SQL code. Spring Framework then brought modular and loosely coupled application design, with Spring Boot later simplifying application setup and development. In parallel, front-end development saw a shift from static pages to dynamic single-page applications (SPAs), supported by JavaScript frameworks like Angular and React.

This evolution pushed full stack developers to master both sides of development. With the rise of RESTful APIs, microservices, and cloud-native architecture, Java Full Stack evolved to include containerization (Docker), CI/CD (Jenkins), and cloud deployment (AWS, Azure), enabling developers to build modern, distributed, and scalable applications. With the release of Java 5 in 2004, the language took a major leap forward by introducing features like generics, enhanced for-loops, annotations, and autoboxing. These updates significantly improved developer productivity and made Java code cleaner and more expressive. he rise of microservices architecture and cloud-native applications has further boosted Java’s relevance, with tools like Spring Boot and Docker integration enabling faster deployment and better resource management.

1.4 TOOLS, COMPONENTS AND TECHNOLOGY STACK

Java Full Stack applications consist of several critical layers. The front-end layer involves technologies such as HTML5, CSS3, Bootstrap for responsive design, and JavaScript libraries/frameworks like Angular, React, or Vue.js for building dynamic user interfaces. The front-end interacts with the back-end using APIs, ensuring a seamless user experience. These technologies work together to deliver rich client-side functionality while maintaining performance and responsiveness. The back-end is powered by Java, where developers use frameworks like Spring Boot to handle the core application logic, data access, and security. Hibernate ORM helps in efficient database interaction. For persistent data storage, relational databases like MySQL or PostgreSQL are commonly used, along with NoSQL databases like MongoDB for flexible data models. Additional tools such as Maven or Gradle manage project dependencies, while Git ensures version control. Full stack developers also leverage RESTful APIs, JWT tokens for security, Docker for containerization, and CI/CD tools like Jenkins for automation, making the tech stack robust and deployment-ready.

1.5 SIGNIFICANCE AND FUTURE SCOPE

Java Full Stack Development holds critical importance in modern software development as it empowers developers to handle every aspect of an application, reducing dependency on multiple specialists. This leads to faster development cycles, lower costs, and better understanding of the application architecture. Java's wide adoption in enterprise applications ensures that full stack developers are in demand across industries such as finance, healthcare, e-commerce, and more. Their ability to switch between front-end and back-end makes them indispensable team members in agile environments. The scope for Java Full Stack developers is vast and ever-expanding. With the growing adoption of digital transformation initiatives, companies are seeking professionals who can develop secure, scalable, and efficient applications. Moreover, the rise of cloud computing, microservices, and platform-as-a-service (PaaS) has increased demand for developers who can handle full application stacks. Java has growing applications in areas such as cloud computing, big data, Internet of Things, and artificial intelligence.

This role opens doors to higher positions such as solution architect, technical lead, or DevOps engineer, making it an excellent career path with long-term prospects. Java also plays a crucial role in Android development, as the Android SDK (Software Development Kit) is primarily based on Java. Although Kotlin is now officially preferred by Google, Java is still widely used in Android applications, especially in legacy systems and existing codebases. This has opened up huge opportunities in the mobile app development industry for Java developers.

Another significant scope lies in web development, where Java is used to build dynamic websites and web applications using technologies like Servlets, JSP, and frameworks such as Spring Boot. It also supports RESTful API development and microservices architecture, which are essential for modern web applications and cloud-based solutions. Java has growing applications in areas such as cloud computing, big data, Internet of Things (IoT), and artificial intelligence. Platforms like Apache Hadoop and Apache Spark are built using Java, making it relevant for data-driven applications. With continuous updates, improved performance, and compatibility with modern tools, Java maintains a strong presence in both academic and industrial domains, ensuring a bright future and wide career scope for Java developers.

Java is also heavily used in software tools development. Many popular Integrated Development Environments (IDEs) like Eclipse and IntelliJ IDEA are themselves written in Java. Additionally, Java provides strong support for building custom desktop applications with GUI frameworks such as JavaFX and Swing.
Though desktop application development has seen a decline with the rise of web and mobile apps, Java remains relevant in specialized applications like simulators, editors, and internal tools for enterprises. In the field of cloud computing and DevOps, Java continues to evolve and integrate with modern platforms. Java-based applications are easily deployed to cloud services like AWS, Azure, and Google Cloud. With the rise of containerization technologies like Docker and orchestration tools like Kubernetes, Java applications are now more scalable and maintainable than ever.

1.6 KEY ADVANTAGES

One of the key benefits of Java Full Stack Development is the ability to manage end-to-end development, providing complete control over the application lifecycle. This results in quicker problem-solving and efficient debugging. Developers can transition between different layers of the application, reducing communication gaps and speeding up development. This holistic understanding also improves code quality, as developers can ensure consistency across the stack. Additionally, full stack developers are well-equipped to work in agile teams and startup environments where multitasking is crucial. From a career perspective, full stack development offers high employability and competitive salaries. It enables developers to adapt quickly to new roles, technologies, and industries, making them more resilient in a changing job market. Organizations also prefer full stack developers due to their versatility, which leads to reduced hiring costs. The vibrant community around Java, extensive documentation, and open-source ecosystem further contribute to faster learning and project development. In short, Java Full Stack Development is a future-ready skillset with immense professional and practical value.

Another benefit of Eclipse is its strong support for enterprise-level Java development. With features like Java EE tools, database connectivity, server management, and web services integration, Eclipse is widely used in building scalable and complex applications. It also supports frameworks such as Spring and Hibernate, further simplifying enterprise development. Built-in integration with tools like Git and SVN makes version control seamless, enabling teams to manage code changes efficiently and work collaboratively.

Finally, Eclipse offers excellent debugging and testing tools that help ensure code quality. The integrated debugger allows developers to set breakpoints, step through code, inspect variables, and evaluate expressions, making it easier to identify and fix bugs. Test frameworks like JUnit are also supported directly within the IDE, helping developers write and run unit tests conveniently. Combined with its large plugin ecosystem and active developer community, Eclipse remains a trusted and evolving platform for professional software development.

CHAPTER 2

JAVA FUNDAMENTALS

2.1 RUNTIME ENVIRONMENT AND JAVA VIRTUAL MACHINE

The Java Virtual Machine (JVM) is an abstract computing machine that enables Java programs to run on any device or operating system without modification. When a Java program is compiled, it is converted into bytecode by the Java compiler. This bytecode is a platform-independent, intermediate code that the JVM can execute. The JVM interprets or compiles this bytecode at runtime into native machine code for the host system, allowing for Java's key feature: "write once, run anywhere." The JVM also plays a crucial role in memory management, security, and execution control. It is responsible for allocating memory to objects, garbage collecting unused objects, and ensuring that malicious code does not harm the host system. The JVM has various components such as the Class Loader, Runtime Data Area, Execution Engine, and Native Method Interface, all of which work together to provide a secure and efficient runtime environment for Java applications.

2.2 MAJOR OOP CONCEPTS

In Java, a class is a blueprint or template for creating objects. It defines the properties and behaviors that the objects created from it will have. A class encapsulates data for the object (in the form of fields) and the actions that can be performed on that data (in the form of methods). Essentially, a class provides the structure that allows you to model real-world entities and their interactions within the program. A class is defined using the class keyword followed by the name of the class. Inside the class, you can declare instance variables, constructors, methods, and nested classes. These elements allow the class to hold data, initialize objects, and perform actions on the data. By using classes, developers can create more structured and reusable code, especially in larger programs where there's a need to manage complex systems. One of the key features of Java classes is the constructor. A constructor is a special method that is invoked when an object is created from a class. It allows you to initialize the object with specific values at the time of creation.

Constructors can either be default (automatically provided by Java when no constructor is defined) or parameterized (where you define the constructor to accept specific parameters and initialize the object accordingly).Classes also support the concept of access modifiers like public, private, and protected, which control the visibility and accessibility of the class and its members. By carefully controlling access to class members, Java provides a way to enforce encapsulation, ensuring that data is protected and only accessed or modified through controlled interfaces.

Polymorphism is one of the core principles of object-oriented programming (OOP) in Java, and it refers to the ability of a single function, method, or object to take on multiple forms. In simple terms, polymorphism allows an object to behave in different ways depending on its context, making the code more flexible, reusable, and easier to extend. Polymorphism is mainly categorized into compile-time polymorphism (method overloading) and runtime polymorphism (method overriding). Compile-time polymorphism, also known as method overloading, occurs when multiple methods in the same class have the same name but different parameters (either in the number of parameters or their types). The compiler determines which method to call based on the method signature at compile time. This allows developers to create methods that perform similar tasks but with different types of input, improving code readability and usability.

Runtime polymorphism, or method overriding, happens when a subclass provides a specific implementation of a method that is already defined in its superclass. Unlike method overloading, runtime polymorphism is determined at runtime. In this case, a method call is resolved to the appropriate method based on the actual object type rather than the reference type. This is particularly useful when a subclass has a more specialized behavior that overrides a generalized behavior in the parent class. An interface, on the other hand, is a contract that defines a set of methods that a class must implement. Unlike abstract classes, interfaces cannot have any method implementations at all (prior to Java 8), and all methods are abstract by default. However, interfaces provide a more flexible way to achieve abstraction because a class can implement multiple interfaces.

Abstraction helps in organizing code in a way that focuses on the essential functionalities while hiding unnecessary complexity. For example, when you interact with a car, you don't need to know how the engine or transmission works; you just need to know how to drive it. Similarly, in programming, abstraction allows a developer to interact with an object without needing to understand its intricate internal workings. This leads to cleaner and more manageable code, as developers can work with high-level concepts without getting bogged down by implementation details.

Another significant advantage of abstraction is that it facilitates code maintenance and reusability. By focusing on the essential features of a system and hiding the implementation, Java programs can be easily updated or extended without affecting the existing codebase. For instance, if the underlying implementation of a method changes, the interface or abstract class remains unchanged, and only the subclass implementations need to be updated. This makes the code more modular and easier to modify over time, reducing the risk of introducing errors when making changes.

Encapsulation is one of the fundamental principles of object-oriented programming (OOP), and it is heavily utilized in Java. It refers to the concept of bundling data (variables) and methods that operate on that data into a single unit, or class, and restricting direct access to some of the object’s components. This is achieved using access modifiers like private, protected, and public. Encapsulation helps ensure that the data is accessed only through a controlled interface, which prevents accidental or unauthorized modifications to the internal state of an object.

The primary benefit of encapsulation is that it promotes data hiding, ensuring that an object's internal state is protected from unauthorized access and modification. In Java, instance variables are often made private to restrict access. Methods that modify or retrieve the values of these variables, such as getters and setters, are provided publicly. This approach gives the developer full control over the data and prevents direct manipulation of internal variables, which could lead to errors or unexpected behaviors.

Encapsulation not only secures data but also improves code maintainability. By providing a public interface to interact with an object while keeping the internal implementation hidden, developers can change the internal working of a class without affecting other parts of the program that use the class. For example, if a method's implementation is changed, the interface (or the public method) remains the same, so the rest of the application remains unaffected, thus promoting modularity.

Another advantage of encapsulation is that it allows for better validation and business logic. The setter methods can include validation logic, ensuring that only valid data is set to an object. For example, if an object represents a person, you can ensure that the age cannot be set to a negative value by including such validation in the setter method. This level of control helps in maintaining the integrity of the data.

Encapsulation is also a key aspect of object-oriented design as it supports abstraction. It helps in organizing code and ensuring that an object's implementation is hidden, allowing developers to focus on what an object does rather than how it does it. By encapsulating the data and its associated methods within a class, Java promotes the creation of well-structured and clean code, leading to better software design. Inheritance is another core principle of object-oriented programming (OOP) that allows one class to inherit properties and behaviors (fields and methods) from another class. This creates a hierarchical relationship between the classes, where a subclass can inherit the functionality of a superclass while also adding its own unique features. In Java, inheritance is implemented using the extends keyword.

2.3 CASTING AND TYPE CONVERSION

In Java, Type Conversion refers to the automatic or manual conversion of one data type into another. There are two types: implicit (widening) and explicit (narrowing) conversion. Widening conversion occurs automatically when a smaller data type is assigned to a larger data type, such as converting an int to a double. This type of conversion is safe and does not lead to data loss. Java handles these conversions internally during operations to ensure smooth computation. Type Casting, on the other hand, is used when narrowing conversion is required—for instance, converting a double to an int.

2.4 MANAGING EXCEPTION

Exception Handling in Java is a powerful mechanism to manage runtime errors and maintain the normal flow of a program. An exception is an event that disrupts the normal execution of code, such as division by zero or accessing a null object. Java provides a structured way to handle these events using the try-catch-finally blocks. Code that may throw an exception is enclosed in the try block, and corresponding error handling is done in the catch block. The finally block, if used, is executed regardless of whether an exception occurs, typically used to release resources. Java distinguishes between checked and unchecked exceptions. Checked exceptions are checked at compile-time and must be either caught or declared using throws, whereas unchecked exceptions are checked at runtime and typically derive from RuntimeException. Custom exceptions can also be created by extending the Exception class, enabling more specific error management.

Proper exception handling not only improves code reliability and readability but also ensures that programs fail gracefully and provide meaningful feedback to users or developers. The try block is used to enclose the code that might throw an exception. This block is the starting point of exception handling. When a program encounters an error inside the try block, the exception is thrown and Java searches for a suitable way to handle it. The try block must be followed by either a catch block or a finally block, or both. This structure ensures that the potentially risky code is safely managed. The catch block is used to handle the exception that occurs in the try block. It defines what action should be taken when a specific type of exception is thrown.

You can have multiple catch blocks to handle different types of exceptions separately. This is especially useful when the code might throw various kinds of exceptions, and each one needs a different response or handling logic. The finally block contains code that will always execute after the try-catch blocks, whether an exception was thrown or not. It is typically used to release resources such as closing files, releasing database connections, or cleaning up memory. The finally block ensures that important cleanup code is executed, even if an unexpected error occurs or if a return statement is used in the try or catch blocks. Another important keyword is throw, which is used to explicitly throw an exception from a method or a block of code.

CHAPTER 3

DESIGN OF LOGIC

3.1 THE ECLIPSE

Eclipse is a popular open-source Integrated Development Environment (IDE) mainly used for Java programming but also supports other languages like C, C++, Python, and PHP through plugins. Originally developed by IBM and now maintained by the Eclipse Foundation, it provides developers with a comprehensive set of tools for building, testing, and deploying applications. Eclipse is based on a modular architecture, where different functionalities can be added via plug-ins, making it highly customizable and extensible for varied development needs.

The platform supports a range of programming activities, including code editing, compiling, debugging, version control, and GUI development. It integrates well with tools like Apache Maven, Gradle, and Git, making it suitable for enterprise-scale application development. With its strong plugin ecosystem and active user community, Eclipse remains one of the most powerful and widely adopted IDEs for Java and other language development.

3.2 KEY FEATURES OF ECLIPSE

Eclipse offers a rich set of features that enhance developer productivity and streamline software development workflows. One of its key features is code assistance, including syntax highlighting, auto-completion, and error detection, which help reduce coding mistakes. The integrated debugger provides powerful capabilities such as breakpoints, step execution, and variable inspection, allowing developers to effectively troubleshoot and fix bugs in their code. Another valuable feature is the plugin architecture, which allows users to install various tools and frameworks like Spring, Hibernate, and Java EE. Eclipse also supports version control systems like Git, enabling seamless collaboration and source code management. Additionally, tools for refactoring, code navigation, and test execution are built-in, making it a one-stop solution for full-cycle software development.

3.3 PACKAGE EXPLORER

The Package Explorer in Eclipse is a fundamental component for navigating Java projects. It presents the hierarchical structure of projects, packages, classes, and files, allowing developers to quickly locate and access code elements. Unlike a regular file explorer, Package Explorer organizes Java resources logically based on the package structure, making it easier to manage large codebases with multiple classes and packages.

Through Package Explorer, developers can perform actions such as opening, renaming, deleting, or refactoring files and folders directly within the IDE. It also integrates seamlessly with other Eclipse views like the Outline view and Problems view, improving the efficiency of debugging and editing. Its intuitive organization enhances code readability and workflow navigation, especially in multi-module or layered applications.

3.4 DEVELOPMENT HURDLES IN ECLIPSE

Despite its many advantages, Eclipse does come with a few challenges that developers commonly face. One major issue is performance lag, especially when working with large-scale projects or using multiple heavy plugins. The IDE can consume considerable system memory, leading to delays in launching or switching between projects. Additionally, the initial setup can be overwhelming for beginners due to the vast array of configurations and tools available.

Another common challenge is plugin compatibility and stability. Since Eclipse relies heavily on plugins, conflicts or outdated plugins can cause the IDE to behave unpredictably or crash. Users may also find it challenging to configure build tools like Maven or Gradle correctly within Eclipse due to its complex integration settings. Despite these issues, proper configuration and regular updates can mitigate many of these challenges and provide a smoother development experience.

CHAPTER 4

DATABASE SETUP

4.1 STRUCTURED QUERY LANGUAGE

SQL (Structured Query Language) is the standard language used to communicate with and manipulate relational databases. It allows users to perform various operations such as creating, reading, updating, and deleting data (CRUD operations). SQL works with tables that consist of rows and columns and supports data integrity, relationships, and constraints to ensure that the data is structured and consistent. It is widely supported across database systems like MySQL, PostgreSQL, Oracle, and SQL Server.

In addition to basic data operations, SQL supports advanced functions like joins, subqueries, indexing, views, and stored procedures. These features help retrieve and manage data efficiently and enable the development of complex queries for reporting and analysis. SQL is essential in backend development, data analytics, and database administration, forming the foundation for managing structured data in enterprise applications.

4.2 COMPARISON WITH NoSQL

SQL and NoSQL databases serve different purposes and are suited for different types of data models. SQL databases are relational and use structured tables with fixed schemas, which makes them ideal for applications that require complex queries, transactions, and strong data consistency. They are well-suited for traditional enterprise applications such as banking systems and CRMs, where data relationships are crucial.

NoSQL databases, on the other hand, are non-relational and offer flexible data models like document, key-value, column-family, or graph. They are better suited for handling large volumes of unstructured or semi-structured data, such as user-generated content or real-time analytics. NoSQL systems prioritize scalability and performance over strict consistency, making them ideal for big data, distributed systems, and dynamic applications such as social media platforms.

4.3 SCHEMA DEFINITION WITH DDL

Data Definition Language (DDL) is a subset of SQL used to define and manage database structures like tables, schemas, indexes, and constraints. Common DDL commands include CREATE, ALTER, DROP, and TRUNCATE. For instance, the CREATE TABLE command is used to define a new table and its columns, while ALTER TABLE modifies the structure of an existing table by adding or removing columns or constraints.

DDL commands directly affect the schema of the database, and changes made using these commands are usually permanent. These commands are automatically committed to the database, meaning they cannot be rolled back unless explicitly managed through database tools. Understanding DDL is essential for designing efficient and normalized database structures that support data integrity and performance.

4.4 DATA OPERATIONS WITH DML

Data Manipulation Language (DML) consists of SQL commands that allow users to manipulate data stored in database tables. The primary DML commands are INSERT, UPDATE, DELETE, and sometimes MERGE. These commands enable users to add new records, modify existing ones, or remove data from tables while maintaining relationships and data integrity defined by constraints.

Unlike DDL, DML operations are transactional, meaning they can be rolled back or committed depending on the outcome of a transaction. This feature ensures data consistency and reliability in applications that require frequent changes to data. DML plays a critical role in application development where dynamic interaction with the database is necessary for processing user input, transactions, and business logic.

4.5 ACCESSIBILITY CONTROL WITH DCL

Data Control Language (DCL) focuses on the authorization and security aspects of database management. It includes commands like GRANT and REVOKE, which are used to assign or remove access permissions for users and roles. With these commands, database administrators can control who is allowed to execute specific SQL operations

like SELECT, INSERT, or DELETE on database objects. DCL ensures that sensitive data is protected and that only authorized users can perform certain actions. For example, a database administrator may grant read-only access to a data analyst while allowing full access to a developer. Proper implementation of DCL is vital for maintaining data security, compliance, and user accountability in multi-user database environments

4.6 RETRIEVING DATA WITH DQL

Data Query Language (DQL) is primarily focused on retrieving data from a database and consists mainly of the SELECT statement. It allows users to fetch specific data based on conditions using clauses like WHERE, ORDER BY, GROUP BY, and HAVING. DQL is crucial for generating reports, performing data analysis, and displaying data in applications.

DQL operations do not modify the database; they are used solely for reading and analyzing data. It is highly versatile and allows complex queries involving multiple tables through joins, subqueries, and set operations. Mastering DQL is essential for developers, analysts, and administrators who need to extract meaningful information from relational databases efficiently.

4.7 ESTABLISHMENT OF DB CONNECTION

A database connection is the process of establishing communication between an application and a database system, enabling the application to execute SQL queries and retrieve or modify data. This connection is typically established using connection strings that contain information such as the database type, hostname, port, database name, username, and password. Java applications, for example, often use JDBC (Java Database Connectivity) to manage these connections. Proper handling of database connections is vital for application performance and resource management. Connections should be efficiently managed using techniques like connection pooling, which reduces the overhead of repeatedly opening and closing connections. Security practices such as encrypting credentials and limiting user privileges further ensure that database interactions are safe and reliable in multi-tier applications.

CHAPTER 5

TESTING

5.1 JAVA UNIT TESTING

JUnit is a widely-used testing framework for Java programming, primarily designed to help developers write and run tests for their Java applications. It is based on the concept of unit testing, where individual components or methods of a program are tested in isolation to ensure they work as expected. JUnit provides annotations such as @Test, @Before, @After, @BeforeClass, and @AfterClass that help define test cases, setup, and teardown operations. This simplicity and structure make JUnit an essential tool in ensuring the correctness of Java code during development.

The framework offers a variety of features, including assertions for validating test results, test suites for organizing multiple tests, and tools for generating test reports. It also integrates easily with build automation tools such as Maven and Gradle, and CI/CD systems like Jenkins, which further automate the testing process. JUnit's flexibility allows developers to write both simple and complex tests, making it a cornerstone of the Java testing ecosystem. It promotes the practice of Test-Driven Development (TDD) by encouraging developers to write tests before the actual code, ensuring better code quality and fewer defects.

JUnit has become a fundamental part of Java development, with many integrated development environments (IDEs) like Eclipse and IntelliJ IDEA offering built-in support for running and debugging tests. The integration of JUnit into these environments makes it easier for developers to write, execute, and analyze their tests directly within their development workflow. Additionally, its widespread usage across open-source and enterprise Java projects speaks to its reliability and importance in modern software development practices. TDD promotes a cycle of writing a failing test, implementing the simplest solution to pass the test, and then refactoring the code to improve quality while ensuring the test still passes.

5.2 VERSIONS OF JAVA UNIT TESTING

JUnit has gone through several versions, each bringing improvements and new features to the framework. The initial versions of JUnit (1.x to 3.x) were quite simple and focused on providing a basic structure for writing and running unit tests in Java. JUnit 4 introduced significant changes, including annotations to replace the previous use of method names for defining tests. This made the framework more flexible and readable. Key features introduced in JUnit 4 included the @Test annotation for marking test methods and @Before and @After for setup and teardown operations.

JUnit 5, which is the latest major release, introduced a new architecture with a more modular structure. It consists of three main components: the JUnit Platform, JUnit Jupiter, and JUnit Vintage. The JUnit Platform is responsible for launching tests, while JUnit Jupiter provides the programming model and extension model, allowing more advanced testing features. JUnit Vintage allows for backward compatibility with older JUnit versions, ensuring that tests written in JUnit 3 or 4 can still be executed within JUnit 5. JUnit 5 also introduces new annotations like @TestInstance for controlling the test instance lifecycle and @DisplayName for more readable test names.

JUnit's evolution reflects the growing demands of modern software development, including better integration with build tools, increased extensibility, and improved support for parallel test execution. JUnit 5, in particular, provides more flexibility for developers and allows for more sophisticated test configurations and assertions, making it the preferred version for many Java developers today. The continuous improvements in JUnit ensure that it remains a relevant and essential tool in the Java development ecosystem.

5.4 ROLE OF JUNIT

JUnit plays a central role in Test-Driven Development (TDD), a software development practice where tests are written before the actual code. TDD promotes a cycle of writing a failing test, implementing the simplest solution to pass the test, and then refactoring the code to improve quality while ensuring the test still passes. This process, often referred to as the "Red-Green-Refactor" cycle, is facilitated by JUnit's simple and intuitive testing framework.

By using JUnit in TDD, developers can immediately see if their code works as intended, which helps catch bugs early in the development process. Since JUnit tests are typically small and focused on individual units of code, they make it easier to pinpoint issues and fix them quickly. Additionally, JUnit tests can be run repeatedly as new code is written, giving developers continuous feedback on the quality of their code. This process not only improves code quality but also ensures that the code meets the specified requirements from the outset.

JUnit's role in TDD also promotes a better design of software systems. Since developers are encouraged to write tests before code, it leads to writing more modular, decoupled, and maintainable code. It also increases the confidence in the software's stability, as the comprehensive suite of tests can be executed at any time during development to ensure that existing functionality is not broken by new changes. Ultimately, JUnit's integration with TDD results in higher-quality code and more reliable software development practices.

5.5 IMPLEMENTATION OF PARAMETERIZED TESTS

Parameterized tests in JUnit allow developers to run the same test multiple times with different inputs, making it easier to test a method with a variety of data. This feature is particularly useful when the same logic needs to be validated with different sets of data, such as testing a mathematical function with multiple input values and expected results. In JUnit 4, parameterized tests can be implemented using the @RunWith(Parameterized.class) annotation, and developers can define the test parameters in a public static Collection<Object[]> data() method that returns the input values for the test.

The benefits of parameterized tests are clear: they reduce code duplication and make it easy to ensure that the tested code works under different conditions. Rather than writing multiple test methods for each set of inputs, parameterized tests allow developers to create a single test method that can be executed with various arguments. This results in more concise and maintainable test code, especially in cases where the same functionality needs to be tested with numerous different inputs.

JUnit 5 extends the parameterized testing capabilities even further with the @ParameterizedTest annotation. In JUnit 5, developers can use various sources for parameters, such as @ValueSource, @EnumSource, @MethodSource, or @CsvSource, allowing greater flexibility in defining the input data. While JUnit is a widely used and powerful framework for unit testing in Java, it also comes with certain disadvantages that developers should be aware of. One key limitation is that JUnit is primarily designed for unit-level testing, meaning it may not be suitable for more complex integration or system-level testing. It lacks built-in support for testing across multiple classes or systems, which can be essential when applications involve databases, networks, or external services.

Another drawback is that JUnit tests require the developer to write a significant amount of boilerplate code. Even simple test cases need proper setup, annotations, and assertions. This can lead to verbosity and increase development time, especially in large projects. Developers must also manually create mock data or use additional libraries like Mockito to simulate dependencies, adding further complexity. JUnit also depends heavily on annotations and naming conventions, which can be confusing for beginners. Misplacing an annotation or misunderstanding its behavior (such as @BeforeEach vs. @BeforeAll) can lead to tests that fail silently or do not execute as expected. Debugging such issues can be difficult and may lead to wasted time and effort during development.

A major limitation is JUnit's lack of support for certain asynchronous and multithreaded test scenarios. Testing concurrent code in Java can be challenging with JUnit alone, as it doesn't provide advanced features for managing threads or verifying asynchronous operations. Developers often need to integrate external tools or write custom logic to effectively test such scenarios, which reduces productivity. Lastly, JUnit tests can give a false sense of code quality. A passing test suite does not guarantee that the code is bug-free or handles all edge cases correctly. Tests must be carefully written to cover all scenarios, including exceptions and invalid inputs. Without comprehensive test coverage, JUnit tests may only confirm that code works in ideal conditions while failing to catch real-world bugs.

CHAPTER 6

CONCLUSION AND FUTURE ENHANCEMENT

6.1 CONCLUSION

The Employee Payroll and Financial Management System provides an efficient, reliable, and scalable solution for handling core HR and financial operations within an organization. By automating tasks such as salary calculation, tax deduction, and financial record management, the system reduces manual workload and human errors, resulting in improved accuracy and productivity.

The structured use of Java for backend logic and SQL for database operations ensures robust data handling and smooth performance. Through well-defined service layers, exception handling, and validation mechanisms, the system is designed with a strong focus on maintainability and reliability. This makes it adaptable to various business environments and organizational sizes. The modular design of the system allows for clear separation of concerns, making it easier to upgrade, scale, or integrate with other enterprise applications. With comprehensive testing using JUnit, the application ensures code correctness and system reliability throughout its lifecycle.

Overall, the project showcases the practical application of full-stack Java development concepts, object-oriented programming, database interaction, and test-driven development in a real-world business context. It not only enhances technical skills but also demonstrates the ability to solve real-time enterprise challenges using modern software engineering practices. The system lays a strong foundation for future enhancements such as mobile accessibility, cloud deployment, AI-based features, and broader integration with attendance or ERP systems. As business needs evolve, this project is well-positioned to grow into a full-scale enterprise payroll and financial management solution.

6.2 FUTURE ENHANCEMENT

A major future enhancement for the payroll system would be integrating it with employee attendance tracking systems, such as biometric scanners or RFID-based systems. This integration would allow real-time attendance data to directly influence payroll calculations, especially for companies with hourly wages or overtime considerations. It would eliminate manual entry errors, improve payroll accuracy, and ensure timely salary processing based on actual work hours.

Developing a mobile-friendly application or companion app would significantly enhance usability for both HR personnel and employees. Employees could view their pay stubs, tax details, and financial records from their smartphones, while HR could approve payrolls, generate reports, and monitor financial summaries on the go. This would add convenience, improve transparency, and promote better employee engagement with the system.

Incorporating AI and machine learning algorithms can lead to smarter payroll systems that predict salary trends, forecast bonuses, and automatically adjust tax deductions based on changing regulations. AI could also help in identifying anomalies in payroll data, flagging possible errors or fraudulent activity before final processing, and assisting HR in making data-driven decisions.

Another potential enhancement is the inclusion of multilingual and multi-currency support for organizations operating globally. This feature would allow the system to manage payroll across different regions, languages, and currencies, ensuring legal compliance with local tax laws and employee compensation regulations, while promoting inclusivity and a global workforce environment.

Finally, cloud-based deployment of the system would offer greater scalability, accessibility, and security. With cloud integration, the payroll system can be accessed from anywhere, backups become automatic, and updates can be rolled out seamlessly. This would also reduce infrastructure costs and make the system more resilient to data loss or system failures, ensuring business continuity.

APPENDIX – 1

SOURCE CODE

SQL QUERIES:

Table 1:

```
CREATE TABLE Employee (
 EmployeeID INT PRIMARY KEY AUTO_INCREMENT,
 FirstName VARCHAR(50) NOT NULL,
 LastName VARCHAR(50) NOT NULL,
 DateOfBirth DATE NOT NULL,
 Gender ENUM('Male', 'Female', 'Other') NOT NULL,
 Email VARCHAR(100) UNIQUE NOT NULL,
 PhoneNumber VARCHAR(15) NOT NULL,
 Address TEXT,
 Position VARCHAR(50) NOT NULL,
 JoiningDate DATE NOT NULL,
 TerminationDate DATE NULL
);
```

Data Insertion:

```
INSERT INTO Employee (FirstName, LastName, DateOfBirth, Gender, Email,
PhoneNumber, Address, Position, JoiningDate, TerminationDate)VALUES
('Sushmitha', 'S', '1995-08-20', 'Female', 'sushmitha@email.com', '9876543211',
'Bangalore, India', 'HR Manager', '2019-03-15', NULL);
INSERT INTO Employee (FirstName, LastName, DateOfBirth, Gender, Email,
PhoneNumber, Address, Position, JoiningDate, TerminationDate) VALUES
('Saravanan', 'R', '1988-12-25', 'Male', 'saravanan@email.com', '9876543212',
'Coimbatore, India', 'Team Lead', '2018-09-01', NULL);
```

Table 2:

```
CREATE TABLE Payroll (
 PayrollID INT PRIMARY KEY AUTO_INCREMENT,
 EmployeeID INT NOT NULL,
 PayPeriodStartDate DATE NOT NULL,
 PayPeriodEndDate DATE NOT NULL,
 BasicSalary DECIMAL(10,2) NOT NULL,
 OvertimePay DECIMAL(10,2) DEFAULT 0.00,
 Deductions DECIMAL(10,2) DEFAULT 0.00,
 NetSalary DECIMAL(10,2) GENERATED ALWAYS AS (BasicSalary +
OvertimePay - Deductions) STORED,
 FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID) ON
DELETE CASCADE
);
```

Data Insertion:

```
INSERT INTO Payroll (EmployeeID, PayPeriodStartDate, PayPeriodEndDate, BasicSalary, OvertimePay, Deductions)VALUES (2, '2024-03-01', '2024-03-31', 45000.00, 1500.00, 4000.00);
INSERT INTO Payroll (EmployeeID, PayPeriodStartDate, PayPeriodEndDate, BasicSalary, OvertimePay, Deductions)VALUES (3, '2024-03-01', '2024-03-31', 60000.00, 2500.00, 6000.00);
```

Table 3:

```
CREATE TABLE Tax (
 TaxID INT PRIMARY KEY AUTO_INCREMENT,
 EmployeeID INT NOT NULL,
 TaxYear YEAR NOT NULL,
 TaxableIncome DECIMAL(12,2) NOT NULL,
 TaxAmount DECIMAL(10,2) NOT NULL,
 FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID) ON DELETE CASCADE
);
```

Data Insertion:

```
INSERT INTO Tax (EmployeeID, TaxYear, TaxableIncome, TaxAmount)VALUES (4, 2024, 48000.00, 4500.00);
INSERT INTO Tax (EmployeeID, TaxYear, TaxableIncome, TaxAmount)VALUES (5, 2024, 55000.00, 5000.00);
```

Table 4:

```
CREATE TABLE FinancialRecord (
 RecordID INT PRIMARY KEY AUTO_INCREMENT,
 EmployeeID INT NOT NULL,
 RecordDate DATE NOT NULL,
 Description TEXT NOT NULL,
 Amount DECIMAL(12,2) NOT NULL,
 RecordType ENUM('Income', 'Expense', 'Tax Payment') NOT NULL,
 FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID) ON DELETE CASCADE
);
```

Data Insertion:

```
INSERT INTO FinancialRecord (EmployeeID, RecordDate, Description, Amount, RecordType)VALUES (1, '2024-03-31', 'Monthly Salary Credit', 45000.00, 'Income');
INSERT INTO FinancialRecord (EmployeeID, RecordDate, Description, Amount, RecordType)VALUES (2, '2024-03-31', 'Gadgets Purchase', 5000.00, 'Expense');
```

JAVA

MainModule.java

```
package main;
import dao.*;
import entity.Employee;
import exception.EmployeeNotFoundException;
import java.time.LocalDate;
import java.util.List;
import java.util.Scanner;
public class MainModule {
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
IEmployeeService employeeService = new EmployeeService();
IPayrollService payrollService = new PayrollService();
ITaxService taxService = new TaxService();
IFinancialRecordService financialService = new FinancialRecordService();
System.out.println(" Payroll Management System \n");
while (true) {
System.out.println("1. View All Employees");
System.out.println("2. Get Employee By ID");
System.out.println("3. Add New Employee");
System.out.println("4. Update Employee");
System.out.println("5. Delete Employee");
System.out.println("6. Exit");
System.out.print("Enter your choice: ");
int choice = Integer.parseInt(scanner.nextLine());
try {
switch (choice) {
case 1:
List<Employee> allEmployees = employeeService.getAllEmployees();
if (allEmployees.isEmpty()) {
```

```
System.out.println("No employees found.");
} else {
for (Employee emp : allEmployees) {
System.out.println(emp.getEmployeeID() + " - " + emp.getFirstName() + " " +
emp.getLastName());
}
}
break;
case 2:
System.out.print("Enter Employee ID: ");
int empId = Integer.parseInt(scanner.nextLine());
Employee emp = ((EmployeeService)
employeeService).getEmployeeByIdWithException(empId);
System.out.println("Employee Found: " + emp.getFirstName() + " " +
emp.getLastName());
break;
case 3:
Employee newEmp = new Employee();
System.out.print("First Name: ");
newEmp.setFirstName(scanner.nextLine());
System.out.print("Last Name: ");
newEmp.setLastName(scanner.nextLine());
System.out.print("Date of Birth (YYYY-MM-DD): ");
newEmp.setDateOfBirth(LocalDate.parse(scanner.nextLine()));
System.out.print("Gender: ");
newEmp.setGender(scanner.nextLine());
System.out.print("Email: ");
newEmp.setEmail(scanner.nextLine());
System.out.print("Phone Number: ");
newEmp.setPhoneNumber(scanner.nextLine());
System.out.print("Address: ");
```

```
newEmp.setAddress(scanner.nextLine());
System.out.print("Position: ");
newEmp.setPosition(scanner.nextLine());
System.out.print("Joining Date (YYYY-MM-DD): ");
newEmp.setJoiningDate(LocalDate.parse(scanner.nextLine()));
System.out.print("Termination Date (optional, blank if none): ");
String termDate = scanner.nextLine();
newEmp.setTerminationDate(termDate.isEmpty() ? null :
LocalDate.parse(termDate));
employeeService.addEmployee(newEmp);
System.out.println("Employee added successfully.");
break;
case 4:
System.out.print("Enter Employee ID to update: ");
int updateId = Integer.parseInt(scanner.nextLine());
Employee updateEmp = ((EmployeeService)
employeeService).getEmployeeByIdWithException(updateId);
System.out.println("Updating: " + updateEmp.getFirstName() + " " +
updateEmp.getLastName());
System.out.print("New Email: ");
updateEmp.setEmail(scanner.nextLine());
System.out.print("New Phone Number: ");
updateEmp.setPhoneNumber(scanner.nextLine());
System.out.print("New Address: ");
updateEmp.setAddress(scanner.nextLine());
employeeService.updateEmployee(updateEmp);
System.out.println("Employee updated.");
break;
case 5:
System.out.print("Enter Employee ID to delete: ");
int deleteId = Integer.parseInt(scanner.nextLine());
```

```
employeeService.removeEmployee(deleteId);
System.out.println("Employee removed.");
break;
case 6:
System.out.println("Thank you \n");
scanner.close();
System.exit(0);
break;
default:
System.out.println("Invalid choice.");
break;
}
} catch (EmployeeNotFoundException e) {
System.out.println("Error: " + e.getMessage());
} catch (Exception ex) {
System.out.println("An error occurred: " + ex.getMessage());
}}}}
```

TestCases.java

```
package junittest;
import dao.*;
import entity.Employee;
import exception.InvalidInputException;
import org.junit.jupiter.api.*;
import java.time.LocalDate;
import java.util.Arrays;
import java.util.List;
import static org.junit.jupiter.api.Assertions.*;
public class TestCases {
private static IEmployeeService employeeService;
private static IPayrollService payrollService;
private static ITaxService taxService;
```

```
@BeforeAll
public static void setup() {
employeeService = new EmployeeService();
payrollService = new PayrollService();
taxService = new TaxService();
Employee emp2 = new Employee(2, "Sushmitha", "S", LocalDate.of(1995, 8, 20),
"Female","sushmitha@email.com", "9876543211", "Bangalore, India", "HR
Manager", LocalDate.of(2019, 3, 15), null);
emp2.setBasicSalary(50000);
emp2.setHra(9000);
emp2.setAllowances(7000);
emp2.setTaxDeductions(6000);
emp2.setInsuranceDeductions(2000);
employeeService.addEmployee(emp2);
Employee emp3 = new Employee(3, "Saravanan", "R", LocalDate.of(1988, 12, 25),
"Male","saravanan@email.com", "9876543212", "Coimbatore, India", "Team Lead",
LocalDate.of(2018, 9, 1), null);
emp3.setBasicSalary(70000);
emp3.setHra(12000);
emp3.setAllowances(10000);
emp3.setTaxDeductions(15000);
emp3.setInsuranceDeductions(3000);
employeeService.addEmployee(emp3);
Employee emp4 = new Employee(4, "M", "A", LocalDate.of(1992, 7, 10), "Female",
"m@email.com", "9876543213", "Goa, India", "Data Analyst", LocalDate.of(2021, 2,
20), null);
emp4.setBasicSalary(60000);
emp4.setHra(10000);
emp4.setAllowances(8000);
emp4.setTaxDeductions(7000);
emp4.setInsuranceDeductions(2500);
```

```
}
@Test
public void testVerifyTaxCalculationForHighIncomeEmployee() {
int employeeId = 103;
int taxYear = 2024;
assertDoesNotThrow(() -> taxService.calculateTax(employeeId, taxYear), "Tax
calculation failed");
}
@Test
public void testProcessPayrollForMultipleEmployees() {
List<Integer> employeeIds = Arrays.asList(101, 102, 103);
LocalDate start = LocalDate.of(2023, 4, 1);
LocalDate end = LocalDate.of(2023, 4, 30);
for (Integer id : employeeIds) {
assertDoesNotThrow(() -> payrollService.generatePayroll(id, start, end),
"Payroll failed for Employee ID: " + id);
}
}
@Test
public void testVerifyErrorHandlingForInvalidEmployeeData() {
Exception exception = assertThrows(InvalidInputException.class, () -> {
Employee emp = new Employee();
emp.setFirstName(""); // Invalid input
emp.setLastName(null);
emp.setDateOfBirth(null);
emp.validate(); // Custom validation
});
String expectedMsg = "Invalid input data";
assertTrue(exception.getMessage().contains("Invalid"), "Should throw exception for
invalid input");
}
```

DBConnUtil.java

```
package util;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class DBConnUtil {
private static final String PROPERTY_FILE = "db.properties";
public static Connection getDbConnection() {
Connection con = null;
try {
String connString = DBPropertyUtil.getConnectionString(PROPERTY_FILE);
con = DriverManager.getConnection(connString);
} catch (IOException e) {
System.out.println("Failed to read DB configuration from properties file.");
e.printStackTrace();
} catch (SQLException e) {
System.out.println("Error while establishing database connection.");
e.printStackTrace();
}
return con;
}
}
```

Employee.java

```
package entity;
import exception.InvalidInputException;
import java.time.LocalDate;
import java.time.Period;
public class Employee {
private int employeeID;
private String firstName;
```

```
private String lastName;
private LocalDate dateOfBirth;
private String gender;
private String email;
private String phoneNumber;
private String address;
private String position;
private LocalDate joiningDate;
private LocalDate terminationDate;
private double basicSalary;
private double hra;
private double allowances;
private double taxDeductions;
private double insuranceDeductions;
public Employee() {}
public Employee(int employeeID, String firstName, String lastName, LocalDate
dateOfBirth, String gender,String email, String phoneNumber, String address, String
position,LocalDate joiningDate, LocalDate terminationDate) {
this.employeeID = employeeID;
this.firstName = firstName;
this.lastName = lastName;
this.dateOfBirth = dateOfBirth;
this.gender = gender;
this.email = email;
this.phoneNumber = phoneNumber;
this.address = address;
this.position = position;
this.joiningDate = joiningDate;
this.terminationDate = terminationDate;
}
public int getEmployeeID() { return employeeID; }
```

```
public void setEmployeeID(int employeeID) { this.employeeID = employeeID; }
public String getFirstName() { return firstName; }
public void setFirstName(String firstName) { this.firstName = firstName; }
public String getLastName() { return lastName; }
public void setLastName(String lastName) { this.lastName = lastName; }
public LocalDate getDateOfBirth() { return dateOfBirth; }
public void setDateOfBirth(LocalDate dateOfBirth) { this.dateOfBirth = dateOfBirth;
}
public String getGender() { return gender; }
public void setGender(String gender) { this.gender = gender; }
public String getEmail() { return email; }
public void setEmail(String email) { this.email = email; }
public String getPhoneNumber() { return phoneNumber; }
public void setPhoneNumber(String phoneNumber) { this.phoneNumber =
phoneNumber; }
public String getAddress() { return address; }
public void setAddress(String address) { this.address = address; }
public String getPosition() { return position; }
public void setPosition(String position) { this.position = position; }
public LocalDate getJoiningDate() { return joiningDate; }
public void setJoiningDate(LocalDate joiningDate) { this.joiningDate = joiningDate; }
public LocalDate getTerminationDate() { return terminationDate; }
public void setTerminationDate(LocalDate terminationDate) { this.terminationDate =
terminationDate;
// Setters for payroll-related fields
public void setBasicSalary(double basicSalary) { this.basicSalary = basicSalary; }
public void setHra(double hra) { this.hra = hra; }
public void setAllowances(double allowances) { this.allowances = allowances; }
public void setTaxDeductions(double taxDeductions) { this.taxDeductions =
taxDeductions; }
```

```
public void setInsuranceDeductions(double insuranceDeductions) {
this.insuranceDeductions = insuranceDeductions; }
public double getTaxDeductions() {
return this.taxDeductions;
}
public double getInsuranceDeductions() {
return this.insuranceDeductions;
}
public double calculateGrossSalary() {
return basicSalary + hra + allowances;
}
public double calculateNetSalary() {
return calculateGrossSalary() - (taxDeductions + insuranceDeductions);
}
public int calculateAge() {
return Period.between(this.dateOfBirth, LocalDate.now()).getYears();
}
public void validate() {
if (firstName == null || firstName.trim().isEmpty()|| lastName == null ||
lastName.trim().isEmpty()|| dateOfBirth == null) {
throw new InvalidInputException("Invalid input data for employee");
}
}
}
```

Payroll.java

```
package entity;
import java.time.LocalDate;
public class Payroll {
public int payrollID;
public int employeeID;
public LocalDate payPeriodStartDate;
```

```
public LocalDate payPeriodEndDate;
public double basicSalary;
public double overtimePay;
public double deductions;
public double netSalary;
public Payroll() {}
public Payroll(int payrollID, int employeeID, LocalDate payPeriodStartDate,
LocalDate payPeriodEndDate, double basicSalary, double overtimePay, double
deductions, double netSalary) {
this.payrollID = payrollID;
this.employeeID = employeeID;
this.payPeriodStartDate = payPeriodStartDate;
this.payPeriodEndDate = payPeriodEndDate;
this.basicSalary = basicSalary;
this.overtimePay = overtimePay;
this.deductions = deductions;
this.netSalary = netSalary;}
public int getPayrollID() { return payrollID; }
public void setPayrollID(int payrollID) { this.payrollID = payrollID; }
public int getEmployeeID() { return employeeID; }
public void setEmployeeID(int employeeID) { this.employeeID = employeeID; }
public LocalDate getPayPeriodStartDate() { return payPeriodStartDate; }
public void setPayPeriodStartDate(LocalDate payPeriodStartDate) {
this.payPeriodStartDate = payPeriodStartDate; }
public LocalDate getPayPeriodEndDate() { return payPeriodEndDate; }
public void setPayPeriodEndDate(LocalDate payPeriodEndDate) {
this.payPeriodEndDate = payPeriodEndDate; }
public double getBasicSalary() { return basicSalary; }
public void setBasicSalary(double basicSalary) { this.basicSalary = basicSalary; }
public double getOvertimePay() { return overtimePay; }
public void setOvertimePay(double overtimePay) { this.overtimePay = overtimePay; }
```

```
package util;
import entity.Employee;
import entity.Payroll;
import entity.Tax;
import entity.FinancialRecord;
import java.util.List;
public class ReportGenerator {
public ReportGenerator() {}
public void generateEmployeeReport(List<Employee> employees) {
System.out.println("--- Employee Report ---");
for (Employee e : employees) {
System.out.println(e);
}
}
public void generatePayrollReport(List<Payroll> payrolls) {
System.out.println("--- Payroll Report ---");
for (Payroll p : payrolls) {
System.out.println(p);
}
}
public void generateTaxReport(List<Tax> taxes) {
System.out.println("--- Tax Report ---");
for (Tax t : taxes) {
System.out.println(t);
}
}
public void generateFinancialRecordReport(List<FinancialRecord> records) {
System.out.println("--- Financial Record Report ---");
for (FinancialRecord r : records) {
System.out.println(r);
}
```

```
}
}
package util;
import java.time.LocalDate;
public class ValidationService {
public ValidationService() {}
public boolean isValidEmail(String email) {
return email != null && email.matches("^[A-Za-z0-9+_.-]+@(.+)$");
}
public boolean isValidPhoneNumber(String phoneNumber) {
return phoneNumber != null && phoneNumber.matches("\\d{10}");
}
public boolean isValidDate(LocalDate date) {
return date != null && date.isBefore(LocalDate.now());
}
public boolean isPositiveAmount(double amount) {
return amount >= 0;
}
package util;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;
public class DBPropertyUtil {
public static String getConnectionString(String fileName) throws IOException {
Properties props = new Properties();
try (FileInputStream fis = new FileInputStream(fileName)) {
props.load(fis);
}
String user = props.getProperty("user");
String password = props.getProperty("password");
String protocol = props.getProperty("protocol");
```

```
String system = props.getProperty("system");
String database = props.getProperty("database");
String port = props.getProperty("port");
return String.format("%s//%s:%s/%s?user=%s&password=%s",
protocol, system, port, database, user, password);
}
}
try (FileInputStream fis = new FileInputStream(fileName)) {
props.load(fis);
}
String user = props.getProperty("user");
String password = props.getProperty("password");
String protocol = props.getProperty("protocol");
public void generatePayrollReport(List<Payroll> payrolls) {
System.out.println("--- Payroll Report ---");
for (Payroll p : payrolls) {
System.out.println(p);
}
}
```

APPENDIX – 2

SCREENSHOTS

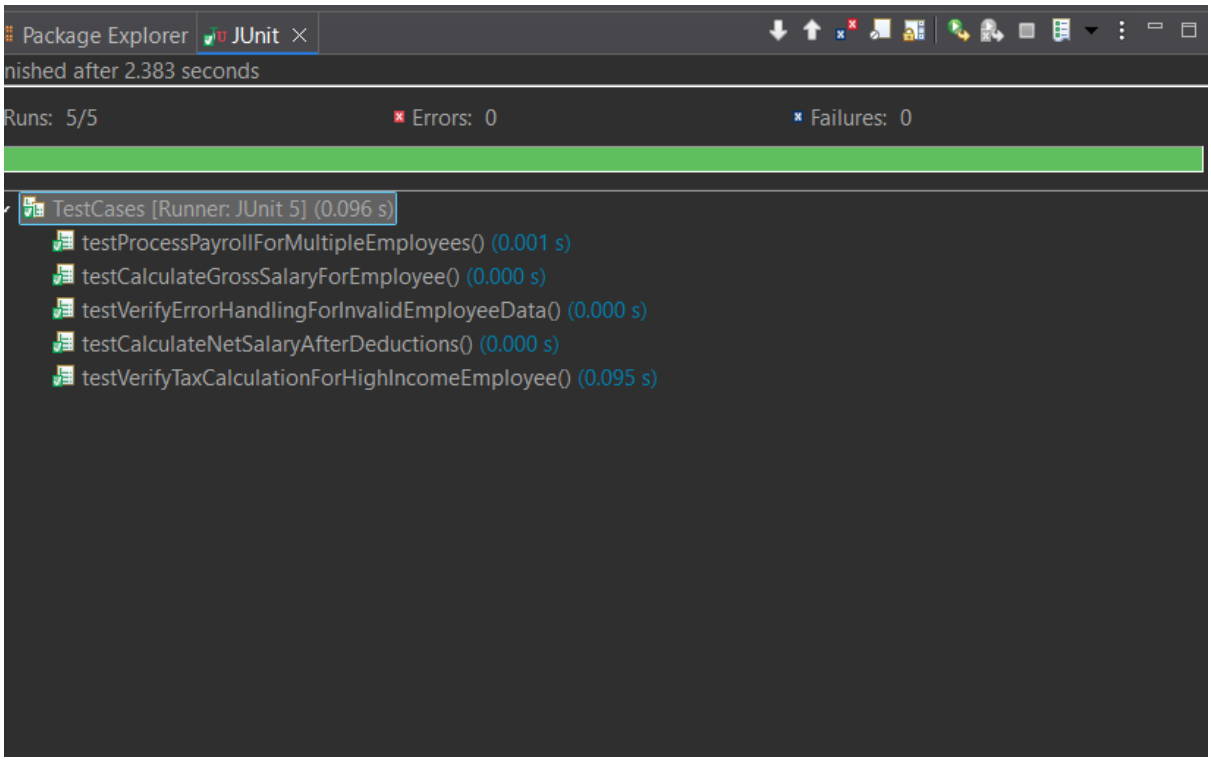

Figure 2.1: Passed Testcases (JUnit)

```
 Payroll Management System

1. View All Employees
2. Get Employee By ID
3. Add New Employee
4. Update Employee
5. Delete Employee
6. Exit
Enter your choice: 1
1 - Madhu K
3 - Saravanan R
5 - Sibi J
146 - sri s
152 - Sushmitha S
1. View All Employees
2. Get Employee By ID
3. Add New Employee
4. Update Employee
5. Delete Employee
6. Exit
Enter your choice:
```

Figure 2.2: Viewing all employee

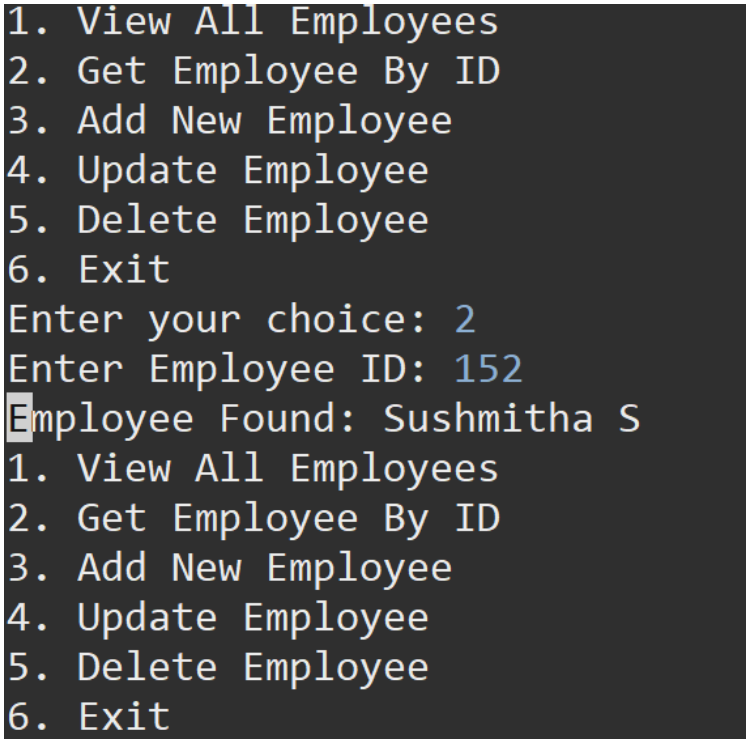

Figure 2.3: Getting data of employee

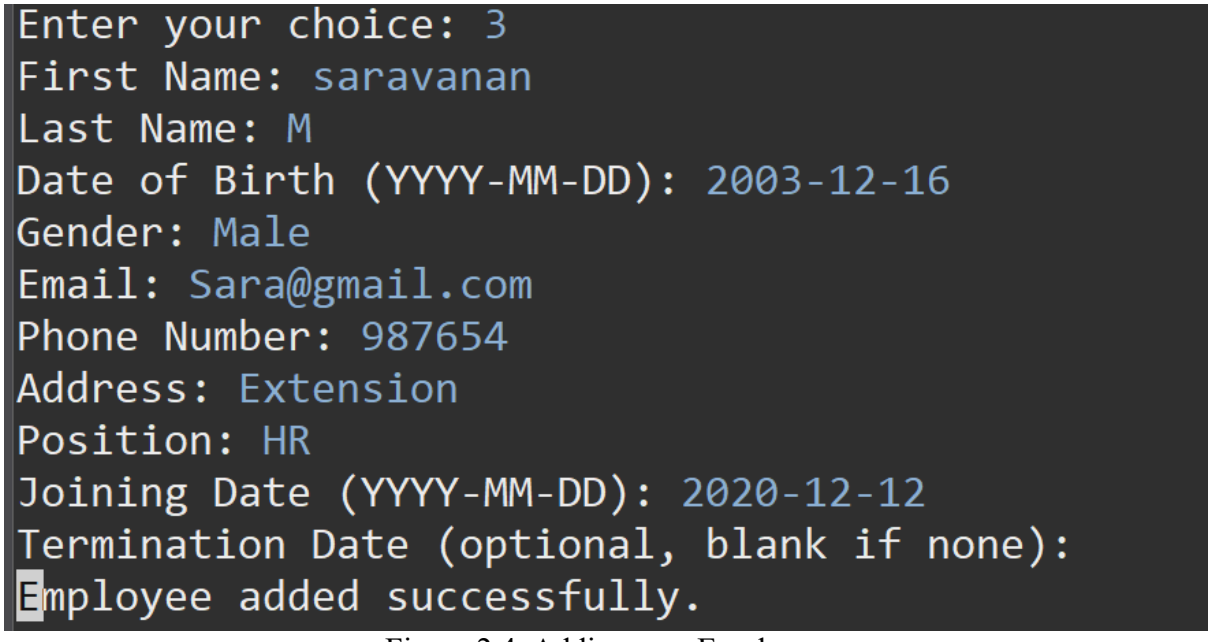

Figure 2.4: Adding new Employee