a)

Input

- **Prompt each worker to input their hours:** The program does prompt the worker to input their hours worked for the week.
- **Allow workers to select their shift number:** The program does allow workers to input their shift number, which is either 1, 2, or 3.
- **Ask the user if they would like to participate in the retirement plan:** The program does ask the user whether they would like to participate in the retirement plan.

Output

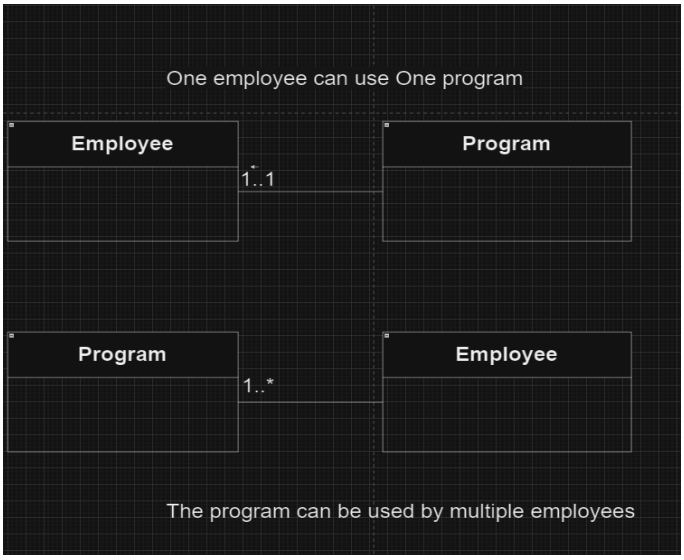
- Displays the worker's hours worked, the shift, hourly pay rate, regular pay, overtime pay, total of regular pay and overtime pay, retirement deduction, and net pay: The program displays all the relevant information for the worker.

Function

- **Calculate overtime for workers whose hours are over 40 for the week:** The program does calculate a worker's overtime based on how many hours they worked after crossing 40 hours which is 1.5 times the usual rate.
- **Deduct 5% from the worker's regular pay and display how much they will be contributing**: The program does deduct 5% from the worker's regular pay should they decide to participate in the retirement plan.

b)

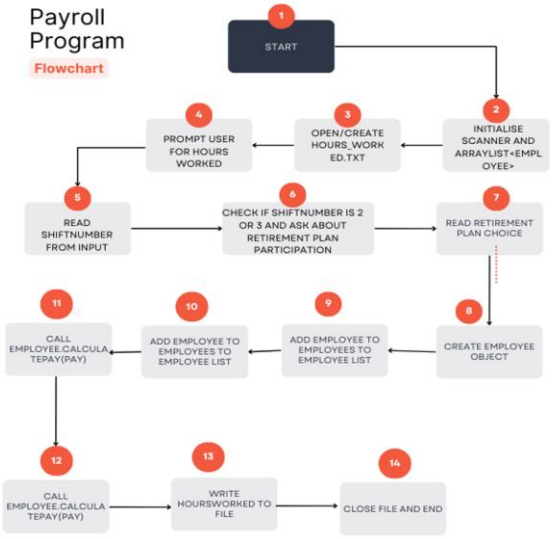
UML Diagram



- This UML diagram depicts the relationship between employees and the program.
- One employee can use one program and the program can be used by many employees.

c)

Flow Chart



Pseudocode

Main Class

```
START

Initialize Scanner for user input
Initialize ArrayList<Employee> to store employee objects

TRY
   Open/Create the file "hours_worked.txt" in read/write mode

   PROMPT user to enter hours worked
   READ hoursWorked from user input

   PROMPT user to enter shift number
   READ shiftNumber from user input

   SET retirementPlan to false

   IF shiftNumber is 2 or 3 THEN
      PROMPT user if they wish to participate in retirement plan (yes/no)
      READ user choice
      IF choice is "yes" THEN
         SET retirementPlan to true
      ENDIF
   ENDIF

   CREATE an Employee object with hoursWorked, shiftNumber, and retirementPlan
   ADD the Employee object to the employees list

   CALL calculatePay() method on the Employee object

   APPEND hoursWorked to the file "hours_worked.txt"

   CLOSE the file

CATCH IOException
   PRINT error message

END
```

Employee Class

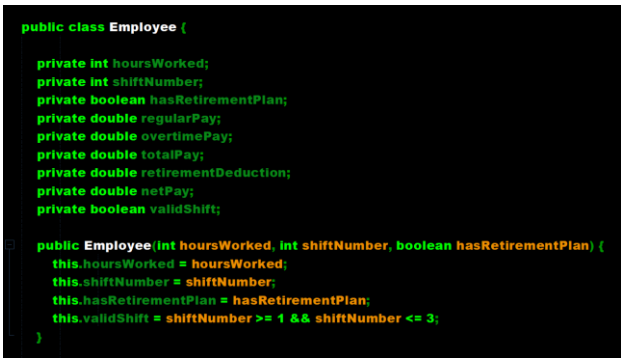
```
METHOD calculatePay()

   IF hoursWorked < 0 THEN
      THROW IllegalArgumentException("Hours worked cannot be negative.")

   SET validShift to TRUE IF shiftNumber is between 1 and 3 ELSE FALSE

   IF validShift is FALSE THEN
      SET regularPay, overtimePay, totalPay, retirementDeduction, netPay to 0
      RETURN

   SET hourlyPayRate based on shiftNumber
   IF hoursWorked > 40 THEN
      SET regularPay to 40 * hourlyPayRate
      SET overtimePay to (hoursWorked - 40) * hourlyPayRate * 1.5
   ELSE
      SET regularPay to hoursWorked * hourlyPayRate
      SET overtimePay to 0
   ENDIF

   SET totalPay to regularPay + overtimePay

   IF shiftNumber is 2 or 3 AND hasRetirementPlan THEN
      SET retirementDeduction to 0.05 * totalPay
   ELSE
      SET retirementDeduction to 0
   ENDIF

   SET netPay to totalPay - retirementDeduction

END METHOD
```

d)

Classes and Objects

- **Employee Class**:

  - **Attributes**:hoursWorked, shiftNumber, hasRetirementPlan, regularPay, overtimePay, totalPay, retirementDeduction, netPay, validShift.



```
public class Employee {

    private int hoursWorked;
    private int shiftNumber;
    private boolean hasRetirementPlan;
    private double regularPay;
    private double overtimePay;
    private double totalPay;
    private double retirementDeduction;
    private double netPay;
    private boolean validShift;

    public Employee(int hoursWorked, int shiftNumber, boolean hasRetirementPlan) {
        this.hoursWorked = hoursWorked;
        this.shiftNumber = shiftNumber;
        this.hasRetirementPlan = hasRetirementPlan;
        this.validShift = shiftNumber >= 1 && shiftNumber <= 3;
    }
```

  - **Methods**:

    - **calculatePay():** Computes regular pay, overtime pay, total pay, retirement deduction, and net pay based on the employee's hours worked, shift number, and retirement plan status.
    - **isValidShift():** Validates the shift number.
    - **Gettermethods**:getRegularPay(), getOvertimePay(), getTotalPay(), getRetirementDeduction(),getNetPay()to retrieve respective values.
    - **Constructor**: Initializes an employee object with provided hours worked, shift number, and retirement plan status, ensuring proper object creation.
    - **Validation**: Includes methods to validate input data, such as hours worked and shift number, ensuring data integrity and preventing errors.

```
public void calculatePay() {
    if (hoursWorked < 0) {
        throw new IllegalArgumentException("Hours
    }

    if (!validShift) {
        regularPay = 0;
        overtimePay = 0;
        totalPay = 0;
        retirementDeduction = 0;
        netPay = 0;
        return;
    }

    double hourlyPayRate;
    switch (shiftNumber) {
        case 1:
            hourlyPayRate = 50;
            break;
        case 2:

    if (hoursWorked > 40) {
        regularPay = 40 * hourlyPayRate;
        overtimePay = (hoursWorked - 40) * hourlyPayRate * 1.5;
    } else {
        regularPay = hoursWorked * hourlyPayRate;
        overtimePay = 0;
    }

    totalPay = regularPay + overtimePay;

    if (shiftNumber != 1 && hasRetirementPlan) {
        retirementDeduction = 0.05 * totalPay;
    } else {
        retirementDeduction = 0;
    }

    netPay = totalPay - retirementDeduction;
}
```

Lists or Arrays

- **ArrayList<Employee>**:
  - **Dynamic Resizing**: Automatically adjusts its size when elements are added or removed, providing flexibility in managing varying numbers of employees.
  - **Efficient Element Access**: Provides fast access to elements via indexing, enabling quick retrieval and updates of employee data.
  - **Iteration**: Supports easy iteration over elements, allowing for processing of employee data using loops or streams.
  - **Generic Type**: Ensures type safety by specifying Employee as the type parameter, preventing insertion of non-employee objects.

```
ArrayList<Employee> employees = new ArrayList<>();
```

Text File

- **Text File ("hours_worked.txt")**:
  - **Continuous**: Stores hours worked data across program executions, ensuring data is not lost when the program terminates.
  - **Read and Write Operations**: Facilitates reading existing data and appending new data, allowing for continuous data accumulation.
  - **Readability** Stores data in a text format that is easy to read and edit manually if needed.
  - **Backup and Recovery**: Simple text files can be easily backed up and restored, providing data safety and recovery options.

```
try {
    RandomAccessFile file = new RandomAccessFile("hours_worked.txt", "rw");

    System.out.println("Please enter how many hours you worked");
    int hoursWorked = sc.nextInt();

    System.out.println("Please enter your shift number: ");
    int shiftNumber = sc.nextInt();

    boolean retirementPlan = false;

    if (shiftNumber == 2 || shiftNumber == 3) {
        System.out.println("Do you wish to participate in the retirement plan? (yes/no)");
        String choice = sc.next();
        if ("yes".equalsIgnoreCase(choice)) {
            retirementPlan = true;
        }
    }
```

Data Structures

- **ArrayList<Employee>**:
  - **Dynamic Management**: Allows for dynamic addition, removal, and access of employee objects.
  - **Flexible and Efficient**: Provides necessary flexibility and efficiency for managing varying numbers of employees.
  - **Built-in Methods**: Offers built-in methods for sorting, filtering, and other common operations on collections, enhancing data management capabilities.
- **File Handling**:

- **RandomAccessFile**: Allows reading and writing to a specific location in a file, enabling efficient updates and access to data without reading the entire file.
- **FileWriter**: Provides a simple way to write data to a file, supporting appending or overwriting modes.
- **BufferedReader**: Allows efficient reading of text data from a file, with methods to read lines or characters, improving file read performance.
- **Error Handling**: Includes mechanisms to handle file-related errors, ensuring robustness and reliability in file operations.
- **Data Consistency**: Ensures data consistency by handling file locks and concurrent access issues, preventing data corruption.