Criterion C

Technique used:

- Linked List (Abstract Data Structure)
- Generating Shifts with 3D arrays (Algorithmic thinking)
- Database serialization
- Local Serialization
- Inheritance
- Encapsulation
- Recursion

Linked List

For this system a linked list was designed in order to store employees; considering employees are constantly added and deleted a dynamic data structure was required. Which is why a linked list was designed, as it allows for more flexibility, and makes them ideal for storing and organizing employees. Additionally, considering the company is likely to grow in the future, the memory efficiency of a linked list is also a strong argument to implement a linked list, as they only require small amounts of memory to store the references to the next node in the list but also the fact that it can expand freely, which leads to minimum memory wastage.

```java
1   package Model;
2
4⊕  This class is called EmployeeNode and represents a single node in a linked list of Employee objects. 
7
8
9⊖  /*
10   * Having these methods in the EmployeeNode class instead of directly accessing
11   * the next field from the EmployeeList class provides several advantages,
12   * including:
13   *
14   * Encapsulation: The setNextEmployeeNode method allows us to change the
15   * reference to the next EmployeeNode in a controlled and safe manner, as we can
16   * add additional logic or checks before setting the next field.
17   *
18   * Abstraction: By providing the getNextEmployeeNode method, we can abstract
19   * away the implementation details of the EmployeeList class, making it easier
20   * to use and understand. This means that we can change the internal
21   * implementation of the EmployeeList class without affecting the rest of the
22   * program.
23   *
24   * Code maintainability: By encapsulating the next field and providing these
25   * methods, we can ensure that the code is easier to maintain and debug, since
26   * we can ensure that the next EmployeeNode is always accessed in a consistent
27   * and correct manner.
28   *
29   * Encourages good coding practices: By using these methods, we can ensure that
30   * the EmployeeList class follows good coding practices, such as encapsulation
31   * and abstraction, which makes the code more readable, understandable, and
32   * maintainable
33   */
34
35
36  public class EmployeeNode {
37
38      /* The class has the following fields: */
39      private Employee employee; // This is an instance of the Employee class that will be stored in the node
40      private EmployeeNode nextEmployee; // This is a reference to the next EmployeeNode in the linked list
41
42⊖     /*
43       * The class has the following constructor:
44       */
45⊖     EmployeeNode(Employee employeeToBeAdded) { // Takes an Employee object as a parameter
46          employee = employeeToBeAdded; // Sets the Employee object field to the parameter
47          nextEmployee = null; // Sets the next EmployeeNode reference to null
48      }
49
50⊖     public void setNextEmployeeNode(EmployeeNode nextNode) { // Sets the next EmployeeNode reference to the provided
51                                                                // EmployeeNode parameter
52          this.nextEmployee = nextNode; // Assigns the parameter to the next EmployeeNode field
53      }
54
55⊖     public EmployeeNode getNextEmployeeNode() { // Returns the next EmployeeNode in the linked list
56          return nextEmployee; // Returns the next EmployeeNode field
57      }
58
59⊖     public Employee get_Employee_Node_Data() { // Returns the Employee object stored in the node
60          return employee; // Returns the Employee object field
61      }
```

Pointer to the next employee

Node constructor

Using method rather than direct reference to the next Employee in EmployeeList

Figure 1 -EmployeeNode class

Used for abstraction, code readability, and encouraging good coding practice

The class *EmployeeList* handles the methods and functionality of the linked list, while the *Node* class is the template for the nodes of the linked list. Each node contains a “next” pointer to the next node in the list. The code below shows its

implementations.

Head of the list

```java
20
21  public class EmployeeList extends LinkedList<Employee>
22
23  {
24      public EmployeeNode head;
25      private static final long serialVersionUID = 1L;
26
27      /*
28       * This algorithm can easily be extended to allow for more sophisticated linked
29       * list operations. For example, it could be modified to insert a new node at a
30       * specific position in the list or to remove a node from the list. To do this,
31       * the algorithm would need to be updated to take in additional arguments, such
32       * as the position or node to insert or remove, and to handle these cases
33       * accordingly. The simplicity and modularity of this algorithm make it easy to
34       * adapt to a variety of use cases.
35       */
36
37      /**
38       *
39       * Adds an Employee object to the end of a linked list.
40       *
41       * @param employee the Employee object to be added to the linked list
42       *
43       */
44
45      public void add_Employee_To_List(Employee employee) {
46          // Create a new node with the employee information
47          EmployeeNode newEmployee = new EmployeeNode(employee);
48
49          // If the list is empty, set the head to the new node and return
50          if (head == null) {
51              head = newEmployee;
52              return;
53          }
54
55          // Traverse the list to find the last node
56          EmployeeNode traversingNode = head;
57          while (traversingNode.getNextEmployeeNode() != null) {
58              traversingNode = traversingNode.getNextEmployeeNode();
59          }
60
61          // Set the next pointer of the last node to the new node
62          traversingNode.setNextEmployeeNode(newEmployee);
63      }
64
```

Figure 3 - EmployeeList class : addEmployee method

```java
/*
 * This custom method could not be achieved with the help of Java's linked list
 * library. As the purpose of this method is very specific and not just
 * searching an object, it had to be made custom. This method is quite simple
 * and works recursively. It takes the head of a list as a parameter for
 * recursive purposes and takes Employee x as a parameter also. Employee X is
 * the employee we are looking for in an instance of EmployeeList. Comparatively
 * to the default "search" method of the linked list; as every employee has a
 * unique employee number, we can identify them using only their employee
 * number, which is what a normal search algorithm would have done. This is
 * extremely useful as it makes searching for an employee much faster, rather
 * than comparing every variable of the employee we are looking for to the
 * employee currently in the node, we can simply compare a primitive type of
 * ints. This makes the program much more efficient.
 */

/**
 *
 * @param head     of the linked list
 * @param employee to search in the linked list
 *
 */
public boolean search_Employee_Node(EmployeeNode head, Employee employeeToSearch) {
    // Start at the head of the linked list
    EmployeeNode current = head;

    // Base case: if the current node is null, the employee is not found in the list
    if (current == null) {
        // Return false to indicate that the employee was not found
        return false;
    }

    // Check if the current node's employee matches the employee being searched for
    if (head.get_Employee_Node_Data().getEmployeeNumber() == employeeToSearch.getEmployeeNumber()) {
        // If the employees match, return true to indicate that the employee was found
        return true;
    }

    // Recursive case: continue searching for the employee in the next node of the
    // linked list
    return search_Employee_Node(head.getNextEmployeeNode(), employeeToSearch);
}
```

Traversing Node

Base case

Recursive call to the method

Figure 4 – EmployeeList class : search employee method

Generating Shifts – Algorithmic thinking

This program allows the user to generate a schedule of shifts based on desired parameters set by the user in the GUI. The parameters include: the number of days, the number of shifts per days, the maximum and minimum number of employees per shift, employees to include, and employees to exclude from the generation. (See Figure xx) This functionality required significant amounts of algorithmic thinking to work and produce the most efficient solution using 3-Dimensional arrays.

*When the user presses on the “Generate Shift” button, after having entered the desired parameters for the generation, the *generateShift()* method is invoked from the *UserController* class and returns a 3-dimensional array. The array has dimensions *number of days by number of shifts by maximum employees per shift*. The *Fisher Yates* shifting algorithm was used to shuffle the employee list.[1]

1) Day of the week
2) Shifts within the day
3) Employees working a specific shift

```java
public static int[][][] generateShifts(int numEmployees, int numShifts, int numDays, int minEmployeesPerShift,
        int maxEmployeesPerShift, int[] preferredEmployees, int[] unavailableEmployees) {
    // Create an empty 3D array to store the shifts
    int[][][] shifts = new int[numDays][numShifts][maxEmployeesPerShift];

    // Create a list of employee numbers
    ArrayList<Integer> employeeNumbers = new ArrayList<>();
    ArrayList<Integer> eligibleEmployees = new ArrayList<>();
    Random random = new Random();

    for (int i = 1; i <= numEmployees; i++) {
        employeeNumbers.add(i);
    }

    // Randomly assign employees to shifts

    for (int i = 0; i < numDays; i++) {
        // Create a copy of the employee numbers list for the current day
        ArrayList<Integer> availableEmployeeNumbers = new ArrayList<>(employeeNumbers);

        for (int j = 0; j < numShifts; j++) {
            // Determine the number of employees needed for the shift
            int numEmployeesNeeded = random.nextInt(maxEmployeesPerShift - minEmployeesPerShift + 1)
                    + minEmployeesPerShift;

            for (int k = 0; k < availableEmployeeNumbers.size(); k++) {

                int employeeNumber = availableEmployeeNumbers.get(k);
                if (!isUnavailable(employeeNumber, unavailableEmployees, i)
                        && isPreferred(employeeNumber, preferredEmployees)) {
                    eligibleEmployees.add(employeeNumber);
                }
            }

            // Assign the eligible employees to the shift
            int employeeIndex = 0;
            for (int k = 0; k < numEmployeesNeeded && k < eligibleEmployees.size(); k++) {
                int employeeNumber = eligibleEmployees.get(random.nextInt(eligibleEmployees.size()));

                // Assign the employee to the shift
                shifts[i][j][k] = employeeNumber;
                employeeIndex++;
            }

            // If there aren't enough eligible employees, fill the remaining spots with any
            // available employee
            for (int k = employeeIndex; k < numEmployeesNeeded && k < availableEmployeeNumbers.size(); k++) {
                int employeeNumber = availableEmployeeNumbers.get(random.nextInt(availableEmployeeNumbers.size()));

                // Assign the employee to the shift
                shifts[i][j][k] = employeeNumber;
            }
        }

        /*
         * Fisher-Yates algorithm from
         * :https://www.geeksforgeeks.org/shuffle-a-given-array-using-fisher-yates-
         * shuffle-algorithm/
         */

        int n = employeeNumbers.size(); // Get the size of the list

        // Iterate over the list in reverse order
        for (int k = n - 1; k > 0; k--) {
            int j = random.nextInt(k + 1); // Generate a ra
            // Swap the elements at index i and j
            int temp = employeeNumbers.get(k);
            employeeNumbers.set(i, employeeNumbers.get(j));
            employeeNumbers.set(j, temp);
        }

    }
```

Populate the ArrayList with employeeNumbers

Set the value in the 3D array and increment employeeIndex

If we run out of eligible employees, fill with random available employees.

Fisher Yates shuffling algorithm

Figure 5 - DAO class: shift generator

[1] (GeeksforGeeks, 2012)

Encapsulation

Encapsulation improved data security, administration, and development ease in this software. The core application was divided into three packages: Model, View, and Controller. The model package stores program data, while the view package handles the GUI. The controller package connects the model and view and contains the Main(Start) class for executing the application. Encapsulation was essential for organizing employee information and achieving success criteria 2 and 3.

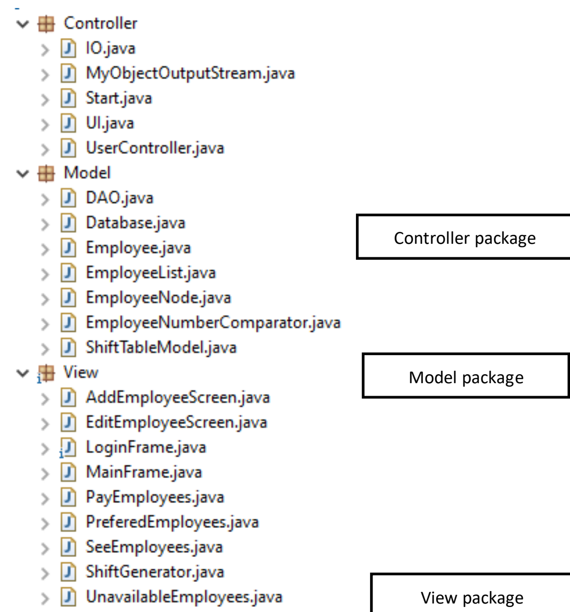


Figure 6 - program's architecture, divided in packages

Serialization - PDF

In our first encounter, the client requested that all data shall be serialized both in a database, and locally on the computer. Considering the amount of data that had to be serialized as a PDF, I made the judgement to use a pre-existing library called iTextPDF, specifically designed for the generation of PDFs, so I could focus on the algorithmic side of generating the PDF rather than the visual aspect of it, which saved an enormous amount of time.

Shift Generation

Generated on: 20/12/2022

Day 1	2,7,12,20,10,0,	21,2,3,1,5,0,	12,16,14,2,5,7,
Day 2	14,2,18,8,14,13,	15,19,13,18,3,7,	3,21,21,9,2,9,
Day 3	9,11,1,1,18,0,	4,10,1,20,19,0,	18,4,18,15,16,5,
Day 4	2,20,7,8,7,17,	2,19,6,19,12,0,	15,19,9,18,7,5,
Day 5	5,1,4,6,15,15,	6,4,6,10,12,9,	4,12,14,2,1,17,

Figure 7 - generated PDF

```
public void generateShiftPDF(JTable table) {
    // Set page size and margins
    Document document = new Document();
    document.setPageSize(PageSize.A3);
    document.setMargins(20, 20, 20, 20);

    try {

        PdfWriter writer = PdfWriter.getInstance(document, new FileOutputStream("Data\\PDF\\Shifts.pdf"));

        // Open document
        document.open();

        // Create PdfPTable

        Paragraph title = new Paragraph("Shift Generation");
        title.setAlignment(Element.ALIGN_CENTER);

        document.add(title);

        // Add date to document
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
        Paragraph date = new Paragraph("Generated on: " + dateFormat.format(new Date()));

        date.setAlignment(Element.ALIGN_CENTER);
        document.add(date);
        PdfPTable pdfTable = new PdfPTable(table.getColumnCount());
        // Iterate over the rows and cells of the JTable and add them to the PdfPTable
        for (int i = 0; i < table.getRowCount(); i++) {
            for (int j = 0; j < table.getColumnCount(); j++) {
                pdfTable.addCell(table.getValueAt(i, j).toString());
            }
        }

        // Add PdfPTable to document
        pdfTable.setSpacingBefore(10f);
        document.add(pdfTable);
        pdfTable.setSpacingBefore(10f);
        // Close document
        document.close();


        writer.close();
    } catch (Exception ex) {
        Logger.getLogger(IO.class.getName()).log(Level.SEVERE, null, ex);
    }
}

}s
```

Figure 9 - DAO class: generate shift pdf method

Serialization – Insertion, deletion, updating, selecting data from Database(Sequential File)

Considering the client's requirement a database was designed for the task. The database will store all of the employee's information, such as their names, their salary and their work location. The following two figures are from the *DAO(Direct Object Accessor)* class. Figure 7 shows the creation of an employee in the database, and figure 6 shows the deletion of an employee from the database.

```java
public void deleteEmployee(JTable table, DefaultTableModel model) {

    try {

        Statement theStatement = theConnection.createStatement();

        int row = table.getSelectedRow();
        String cell = table.getModel().getValueAt(row, 0).toString();

        Employee deleted = new Employee(Integer.valueOf(cell));
        /*Deleting data from serialized file*/

        theIO.deleteEmployee(deleted, this.list);

        /*Deleting data from serialized database*/
        theStatement.executeUpdate("DELETE FROM employee_info where employeeNumber=" + cell);

        if (row != -1) {

            // remove selected row from the model

            model.removeRow(table.getSelectedRow());

        }

        JOptionPane.showMessageDialog(theUserController.screen, "Success");

    } catch (Exception ex) {
        JOptionPane.showMessageDialog(theUserController.screen, "An error occured in contacting the database");

    }}
```


Figure 10 - DAO class : deleteEmployee method

```java
    public boolean createUser(Employee theUser) {
        try {
            PreparedStatement theStatement = theConnection
                    .prepareStatement("insert into employee_info (employeeNumber, firstName, lastName, workLocation, "
                            + "" + "annualSalary, fullTime, gender, role, password ) values (?,?, ?, ?, ?,?, ?,?,?)");

            theStatement.setString(2, theUser.getFirstName());
            theStatement.setInt(1, theUser.getEmployeeNumber());
            theStatement.setString(3, theUser.getLastName());
            theStatement.setString(4, theUser.getWorkLocation());
            theStatement.setInt(5, theUser.getAnnualSalary());
            theStatement.setBoolean(6, theUser.isFullTime());
            theStatement.setString(7, String.valueOf(theUser.getGender()));
            theStatement.setString(8, theUser.getRole());
            theStatement.setString(9, String.valueOf(theUser.getPassword()));

            theStatement.execute();

            theUserController.employeeList.add_Employee_To_List(theUser);
            this.list.add_Employee_To_List(theUser);

            return true;

        } catch (Exception ex) {

            Logger.getLogger(getClass().getName()).log(Level.SEVERE, "An error occurred while creating a new user", ex);
            return false;

        }
    }
```


Figure 11 - DAO class: createUser method

```java
import java.sql.Connection;

public class Database {

    private Connection theConnection;
    private final String link = "jdbc:mysql://localhost:3306/mydb";
    private final String userName = "root";
    private final String pass = "1234";

    private static Database empDB = new Database();

    private Database() {

        try {
            Class.forName("com.mysql.cj.jdbc.Driver"); // class name for MySQL Driver
            this.theConnection = DriverManager.getConnection(link, userName, pass);// retrieve database connection

        } catch (Exception ex) {

            JOptionPane.showMessageDialog(null, "Failed to connect to the database. Contact administrator");
        }
    }

    public static Database getDB() {
        return empDB;
    }

    public Connection getConnection() {
        return this.theConnection;
    }
}
```

Set up credential variables to access DB

Getting the connection, with aforementioned credentials

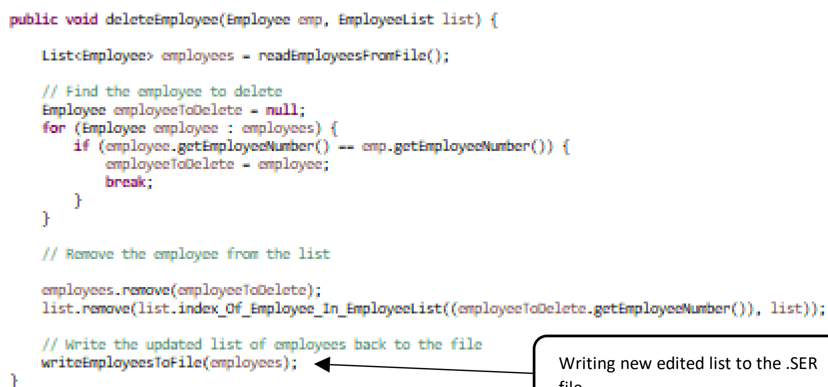
Abstraction

Figure 12 - Database class

Serialization – Local file (Sequential File)

As the client requested for the data to be both serialized in a DB and in a local file, I used Java's *FileOutputStreamn, FileInputStream, ObjectOutputStream, ObjectInputStream* classes to serialize the employees. The way it works, is that everytime a new employee is added, the currently held list is read, and then the new instance of an employee is added to the list which is then written into the file again. The two methods below are the ones used

Figure 9 shows the deletion method from the .SER file, while Figure 10, shows the read and write methods.



```java
public void deleteEmployee(Employee emp, EmployeeList list) {

    List<Employee> employees = readEmployeesFromFile();

    // Find the employee to delete
    Employee employeeToDelete = null;
    for (Employee employee : employees) {
        if (employee.getEmployeeNumber() == emp.getEmployeeNumber()) {
            employeeToDelete = employee;
            break;
        }
    }

    // Remove the employee from the list

    employees.remove(employeeToDelete);
    list.remove(list.index_Of_Employee_In_EmployeeList((employeeToDelete.getEmployeeNumber()), list));

    // Write the updated list of employees back to the file
    writeEmployeesToFile(employees);
}
```


Figure 13 - IO class: deleteEmployee method

```java
public List<Employee> readEmployeesFromFile() {
    List<Employee> employees = new ArrayList<>();
    try (FileInputStream fis = new FileInputStream("Data\\Ser\\Employees.ser");
            ObjectInputStream ois = new ObjectInputStream(fis)) {
        employees = (List<Employee>) ois.readObject();

    } catch (Exception e) {
        Logger.getLogger(IO.class.getName()).log(Level.SEVERE, null, e);
    }
    return employees;s
}

public void writeEmployeesToFile(List<Employee> employees) {
    try (FileOutputStream fos = new FileOutputStream("Data\\Ser\\Employees.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos)) {
        oos.writeObject(employees);
    } catch (Exception e) {

        Logger.getLogger(IO.class.getName()).log(Level.SEVERE, null, e);

        JOptionPane.showMessageDialog(null, e);
    }
}
```

Return the List from the .SER file

Write the list to the .SER file

Figure 14 - IO class : read and write employee method

Recursion

Recursion was used in this program for improved readability, which makes the code less cluttered. Despite recursion's complexity, the following method is used to return an actual employee from the *EmployeeList*; indeed, to make development easier and more intuitive, some temporary employees were created with only an employee number, with the only aim of returning a full employee using the *return_Employee_From_List()* method.

```java
public Employee return_Employee_From_List(EmployeeNode head, Employee employee_To_Return) {
    // Start at the head of the linked list
    EmployeeNode current = head;

    // Base case: if the current node is null, the employee is not found in the list
    if (current == null) {
        // Return null to indicate that the employee was not found
        return null;
    }

    // Check if the current node's employee matches the employee being searched for
    if (head.get_Employee_Node_Data().getEmployeeNumber() == employee_To_Return.getEmployeeNumber()) {
        // If the employees match, return the employee object
        return head.get_Employee_Node_Data();
    }

    // Recursive case: continue searching for the employee in the next node of the
    // linked list
    return return_Employee_From_List(head.getNextEmployeeNode(), employee_To_Return);
}
```

Recursive call

Figure 15 - EmployeeList class: return employee method

```java
public EmployeeNode delete_Employee_From_EmployeeList(EmployeeNode head, int employeeNodeNumber) {

    if (head == null) // If linked list is empty, no operation can be done
        return null;

    if (employeeNodeNumber == 0) // if the index passed in the parameter is the first EmployeeNode(head), no
                                 // operations can be done
        return head;

    if (employeeNodeNumber == 1) // if index is 1
    {
        EmployeeNode employeeEmployeeNode = head.getNextEmployeeNode();
        return employeeEmployeeNode;
    }

    head.setNextEmployeeNode(delete_Employee_From_EmployeeList(head.getNextEmployeeNode(), employeeNodeNumber - 1));

    return head;
}
```

Recursive call

Figure 16 - EmployeeList class: deleteEmployee method

```java
/**
 *
 * @param head of the linked list
 *
 */
public int get_count_of_nodes(EmployeeNode head) {

    // Base case
    if (head == null)
        return 0;

    // Count is this EmployeeNode plus rest of the list
    return 1 + get_count_of_nodes(head.getNextEmployeeNode());
}

public int get_Node_Count_Helper_Method() {

    return get_count_of_nodes(head);
}
```

Recursive call

Helper method

Figure 17 - EmployeeList class: get node count method

Polymorphism

As mentioned, some “dummy” employees are sometimes created to ease the development, thus a new constructor to the *Employee class* was created, as shown in figure 11.

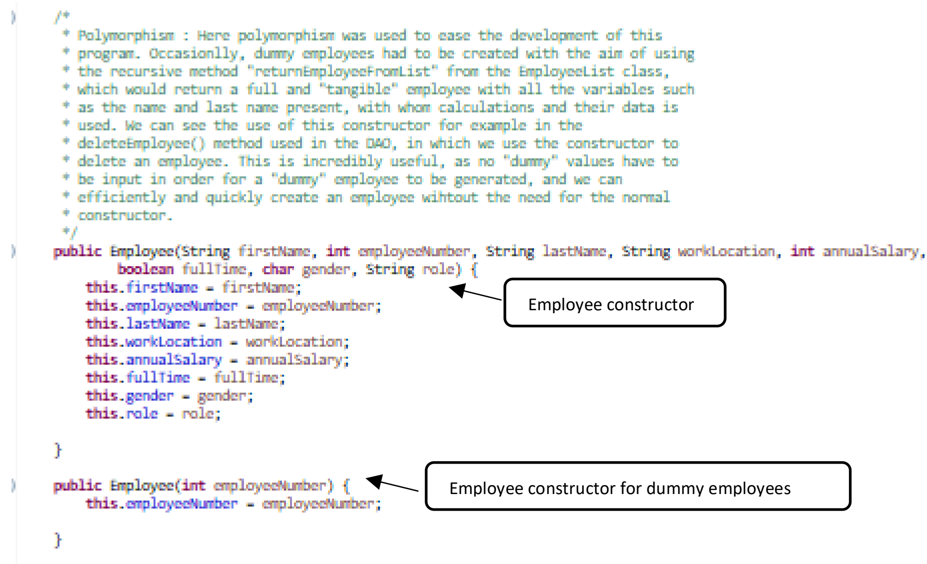


Figure 18 - Employee class: two employee constructors

Inheritance

Inheritance was used extensively in this program. Its first instance appears in the *EmployeeList* custom linked list, which also extends to *java.util.LinkedList*. class Even though most of the features used are custom made, in some instances original methods were used to supplement the custom methods. Such as the *getFirst(),* for which there was no need to create a custom method just for the sake of creating a new one.

```
8  /* Inheritance: Inheritance was used here for very specific purposes:
9   *
10  * 1)   Code reuse: Inheriting the LinkedList class allows you to reuse the existing implementation of linked list operations
11  *       such as insertion, deletion, and traversal, saving you the time and effort of writing these operations from scratch.
12  *
13  * 2)  Improved efficiency: The LinkedList class is implemented in Java and is optimized for performance, so your custom linked list
14  *      class can benefit from these optimizations.
15  *
16  * 3)  Extension purposes:  Inheriting the LinkedList class  makes it easier to integrate with other Java APIs that expect a
17  *      LinkedList object as input, which will allow for future developers of this system not to have to redesign a large portion of the cod
18  *
19  * 4) Simplicity: Inheriting the LinkedList class can make it easier to implement a custom linked list class, especially
20  *         because I did not need to customize the basic linked list operations.
21  *
22  */
23
24
25  public class EmployeeList extends LinkedList<Employee>
26
```

Figure 20 - EmployeeList class inheritance

```
/*
 * The method returns an integer value that represents the index of the first
 * occurrence of the Employee object whose employee number matches the item
 * argument. If there is no match, it returns -1.
 */

public int index_Of_Employee_In_EmployeeList(int index, EmployeeList employeeList) {

    for (int i = 0; i < employeeList.get_Node_Count_Helper_Method(); i++) {
        if (employeeList.get(i).getEmployeeNumber() == index) {
            return i;
        }
    }
    return -1;
}
```

Java.util.LinkedList default method used

Figure 19- EmployeeList class : index method that uses default linked list method

Word Count : 730

Bibliography :

GeeksforGeeks. (2012). *Shuffle a given array using Fisher–Yates shuffle Algorithm*. [online] Available at: https://www.geeksforgeeks.org/shuffle-a-given-array-using-fisher-yates-shuffle-algorithm/.