2023

COMP – 1815 JVM Programming Languages – GROUP 16

INDIVIDUAL + GROUP REPORT

SANTA S SIAN - 001108298

Table of Contents

INDIVIDUAL REPORT

Section 1 – Critical Comparison of Kotlin and Scala

Introduction

The importance of programming paradigms in software development cannot be overstated, as they influence the methods used for system architecture and problem-solving. Generally speaking, OOP (Object-Oriented Programming) and FP (Functional Programming) are two of the most widely used paradigms, with different approaches and philosophies. Modern programming languages like Kotlin and Scala are examples of these paradigms. Kotlin predominantly aligns with OOP, while Scala integrates OOP and FP elements. This section offers a comparative analysis of these paradigms and their representative languages, informed by my extensive project involvement and comprehensive research.

Overview of Object-Oriented Programming

Object-oriented programming (OOP) is a well-known paradigm that uses "objects" (data structures made up of data fields and procedures) as the primary building blocks for application design and development. The cornerstone concepts of OOP encompass encapsulation, inheritance, and polymorphism. Encapsulation facilitates the combination of data with corresponding methods, inheritance permits the derivation of new classes from existing ones, and polymorphism enables the utilization of a single interface for a broad spectrum of actions. Kotlin, renowned for its concise and seamless compatibility with Java, represents a contemporary language that adheres to the principles of OOP. It augments these principles with advanced features such as data classes and extension functions, establishing itself as a formidable option for intricate application development.

Overview of Functional Programming

As a distinct paradigm, functional programming underscores the principles of immutability and statelessness, positioning functions as central elements. It avoids mutable data and state modifications, resulting in programs that are typically more predictable and straightforward to troubleshoot. Scala, a versatile language that amalgamates Functional Programming with Object-Oriented Programming, effectively supports this paradigm by providing first-class functions, pattern matching, and comprehensive collection libraries. The functional characteristics of Scala promote the development of concise, expressive code devoid of side effects. This attribute renders Scala particularly advantageous for constructing applications that demand elevated levels of concurrency and scalability.

Comparative Analysis

In comparing these programming paradigms, Object-Oriented Programming (OOP) is notably effective in emulating real-world systems and managing intricate software architectures. Its focus on objects and classes provides an intuitive framework, particularly for developers acquainted with the physical world's structures and relationships. Nonetheless, OOP can occasionally result in convoluted and less maintainable code, attributed primarily to its reliance on mutable state and the concept of "inheritance."

Conversely, Functional Programming (FP) offers substantial benefits in developing scalable and concurrent applications. The principle of immutability in FP facilitates a more transparent comprehension of data flow. It minimizes the likelihood of side effects, enhancing the ease of reasoning about, testing, and maintaining programs. However, FP can pose a challenging learning curve and yield less intuitive code, especially for those versed in imperative programming styles.

Drawing from my learning experiences, the decision to use Kotlin or Scala, and by extension, to choose between OOP and FP, was contingent upon the specific requirements of the task. For constructing complex systems characterized by clear hierarchies and relationships, Kotlin's OOP capabilities proved exceedingly valuable. In contrast, Scala's FP features were distinctly advantageous for tasks requiring high abstraction levels and sophisticated data flow manipulation.

Conclusion

In conclusion, object-oriented and functional programming paradigms present distinctive strengths and address varied facets of software development. The selection between Kotlin and Scala, or between OOP and FP, is predominantly guided by the demands and context of the project. OOP offers a more instinctive methodology for modelling intricate systems, while FP is superior in situations that require elevated abstraction and concurrent processing. As these programming paradigms progress, adopting a hybrid approach that combines the merits of both OOP and FP is emerging as a popular and efficacious strategy in contemporary software development.

Section 2 – Evaluation of the evolution of our application

Introduction to Project and Team

The project at hand involved the creation of a comprehensive project management system specifically designed to enhance the efficiency of managing software development projects. This endeavour was tackled by a heterogeneous team, with each participant contributing distinct skills and viewpoints. In this venture, my responsibilities were diverse, encompassing the conceptualization and execution of essential features. Collaborating within this team necessitated proficient communication and cooperative efforts, a necessity amplified by the project's intricate characteristics and the team's geographically dispersed composition.

Development Challenges and Solutions

A principal challenge faced in this project pertained to the preservation of the successor task relationships. Despite numerous efforts and diverse approaches, achieving consistent and precise recording of these relationships emerged as a formidable challenge. This issue was of paramount importance, as task relationships are integral to the functionality of any project management system, forming the basis for the logic that governs task sequencing and prioritization.

1. **Task Successor Management**

The challenge originated from the requirement to accurately represent and store intricate relationships between tasks efficiently and dependably. The initial strategy adopted for this purpose entailed directly embedding these relationships within the tasks.

Alternative Approaches

Reflecting on this, a more practical approach could have involved utilising a dedicated relational database. This database could store tasks as separate records, with dependencies depicted as connections between these records. This approach would likely simplify the data structure and improve the system’s integrity and scalability. Moreover, incorporating an ORM (Object-Relational Mapping) framework could have facilitated a more seamless interaction between the application and the database, rendering data operations more intuitive and reducing the likelihood of errors. Another notable mention is that upon using AI to help generate code to assist this issue, the successor relationships seem to save in the JSON file; however, they do not load back up upon restarting the application. Perhaps further revision on the loading of the GUI could be conducted to investigate precisely why this issue occurs.

2. **Implementing Azure DevOps for Version Control**

Another notable challenge encountered involved the implementation of Azure DevOps for version control. Although I possessed prior experience with Azure DevOps, the varied technical backgrounds of the team members resulted in the setup and uniform application of this tool being less than straightforward for all involved.

Overcoming the Challenge

In response to this, I conducted a series of interactive training sessions and prepared exhaustive guides aimed at acquainting the team with Azure DevOps. This initiative transcended mere technical setup; it represented a vital component of team building and the dissemination of knowledge. It was imperative to ensure that each team member attained proficiency with Azure DevOps, as this was crucial for sustaining a seamless workflow and consistent code integration throughout the project's entire lifespan.

Implementation of the GUI

An extra challenge encountered was the development of the graphical user interface (GUI). Although the IntelliJ Integrated Development Environment (IDE) provides a UI designer that eases the process of GUI creation, we chose to construct the GUI manually. This decision, while aimed at fostering a more profound understanding and granting more significant control over the interface, consequently led to a more labour-intensive and stressful process.

Reflecting on GUI Development

The manual approach to GUI development required meticulous attention to detail and an extensive understanding of layout managers and GUI components. In hindsight, leveraging the UI designer could have expedited the development process, providing a more efficient way to create a professional and user-friendly interface.

Notable limitation - Dynamic modification

The system currently lacks the functionality to edit tasks or give tasks details, as it only allows for adding and deleting tasks from projects. The potential for task editing was recognized during the implementation phase. However, it was determined that incorporating this feature would exceed the project's scope and was deemed unnecessary for the specific academic requirements. However, it is essential to acknowledge that in real-world situations, the ability to edit tasks is crucial for project management applications, considering the ever-evolving nature of projects and tasks.

Reflection on Teamwork and Personal Contribution

Collaborating within a group presented a unique array of challenges and learning opportunities. Paramount among these was the necessity for effective teamwork, which demanded articulate communication, mutual respect, and the capability to integrate varied perspectives into a unified strategy. My responsibilities within the team were primarily centred around conventional development tasks, which included mentorship and facilitation roles, especially in domains like Azure DevOps setup and problem-solving. This experience highlighted the importance of patience, clear communication, and the adaptability of teaching methods to accommodate different learning styles.

Team Dynamics

The team dynamics were generally positive, with each member eager to contribute. However, coordinating tasks at different times and schedules sometimes led to delays and miscommunications. Regular team meetings and more structured communication channels could have mitigated these issues.

Lessons Learned and Improvement Opportunities

The project served as a significant educational experience, elucidating various areas for enhancement. Firstly, an in-depth initial planning stage with a well-defined distribution of roles and responsibilities would have established a more solid groundwork. Secondly, the integration of regular code reviews and pair programming sessions could have identified issues such as the task dependency problem at an earlier stage and cultivated an environment more conducive to collaboration and learning.

Regarding what went well, the team's flexibility and dedication were noteworthy. Despite facing challenges, there was a persistent endeavour to devise solutions and maintain project momentum. Nonetheless, there was a notable opportunity for improvement in risk management and contingency planning. Adopting a more proactive approach in foreseeing and preparing for potential hurdles might have lessened the impact of our difficulties.

Conclusion

Looking back, the development of the project management system represented not only a venture in software development but also a significant exploration of team dynamics and personal development. The challenges encountered, especially in managing task relationships, yielded crucial insights into the intricacies of application development. This experience also highlighted the criticality of choosing appropriate tools and methodologies for the task. Although teamwork was sometimes challenging, it proved immensely rewarding, imparting valuable lessons in collaboration, communication, and the necessity of a well-rounded skill set within a team.

GROUP REPORT

Section 1: Feature Checklist and Git Repository Link

Feature	Implemented	Notes
Java GUI for Project Management	✓	Allows for adding and deleting projects and tasks, as well as viewing the adjacency matrix and setting up task successors
Domain Classes in Kotlin	✓	Represents tasks and projects
Adjacency Matrix Generation	✓	Generates matrix to represent the successor relationships
Persistence (Optional)	✓	Saves and loads projects and their tasks from JSON file
Integration of Java GUI and Kotlin	✓	Kotlin classes support Java GUI functionality.

Git Repository (for version control):
https://ss8760t@dev.azure.com/ss8760t/Group%2016%20Coursework%20Team/_git/Group%2016%20Coursework%20Team

Panopto video demonstration link:

https://gre.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=a2c8bfa4-511f-45c3-816d-b0d600f82aa2

Worth noting: Video submission was done before submitting code, there were a few things (naming of "_tasks" and "_successors") that were talked about in the video, which we fixed in the actual code. You can also check the Azure DevOps git repository to verify a couple of updates we made post-submission. This report and the zip of the code contain the updated code with bugs fixed.

Section 2: Screenshots of implemented features

The interface displays a tree view of projects with expandable nodes for tasks, shown in a hierarchical structure. This user-friendly way presents the relationship between projects and their constituent tasks.

The GUI offers buttons for fundamental operations like "Add Project", "Delete Project", "Add Task", "Delete Task", "View Matrix", and "Set Successor Task".

Figure 1: GUI main window

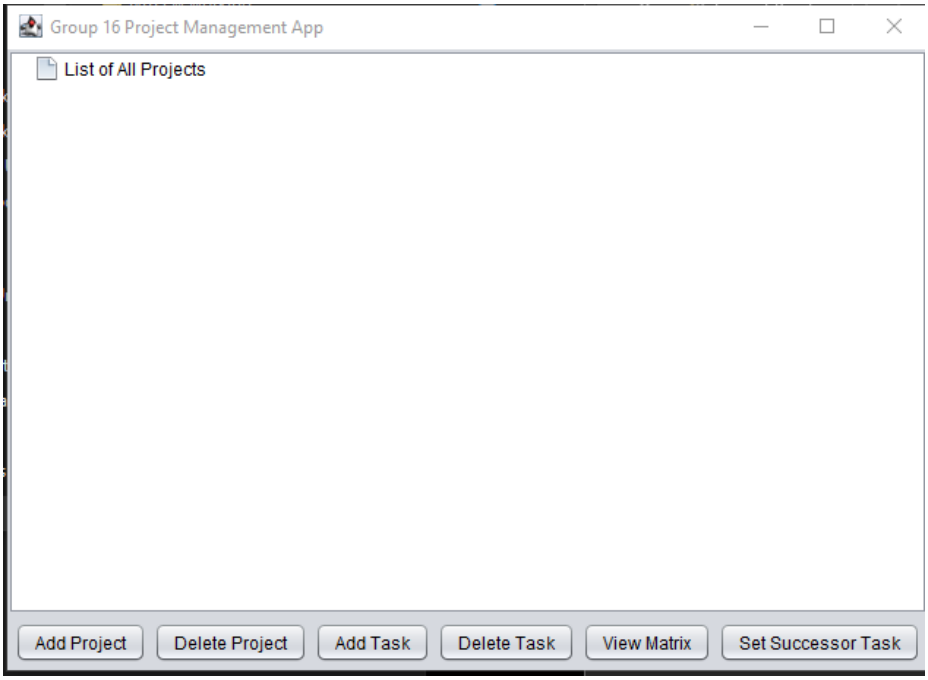


Figure 2: Adding a project.

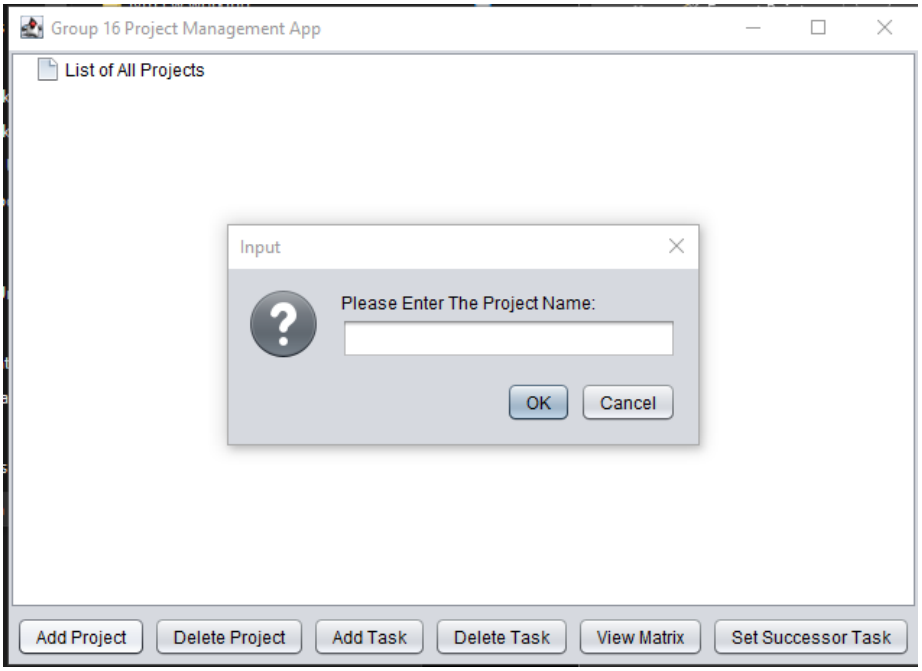


Figure 3: Project given a name and added to the GUI

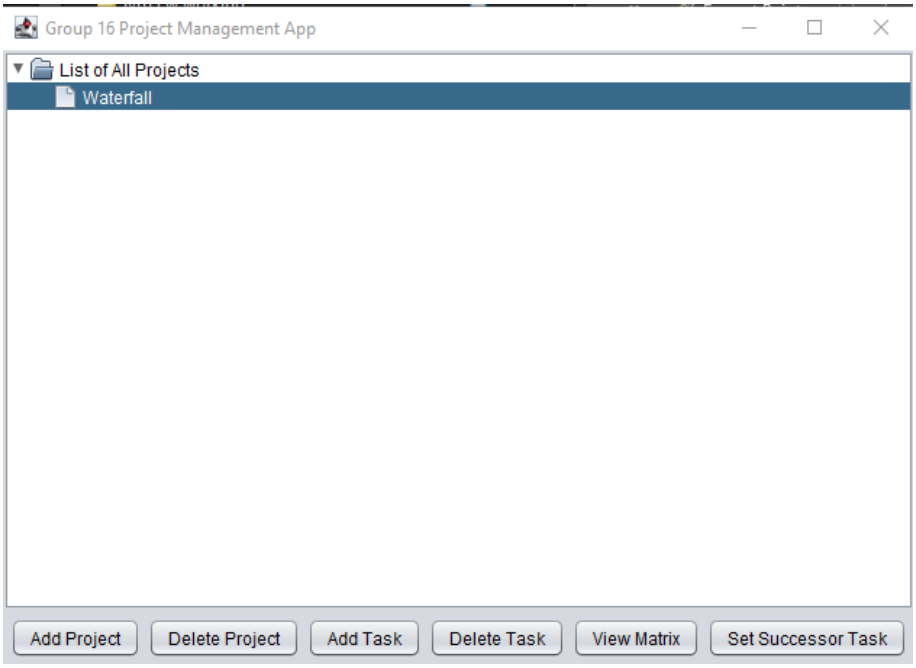


Figure 4: Adding a task to the project.

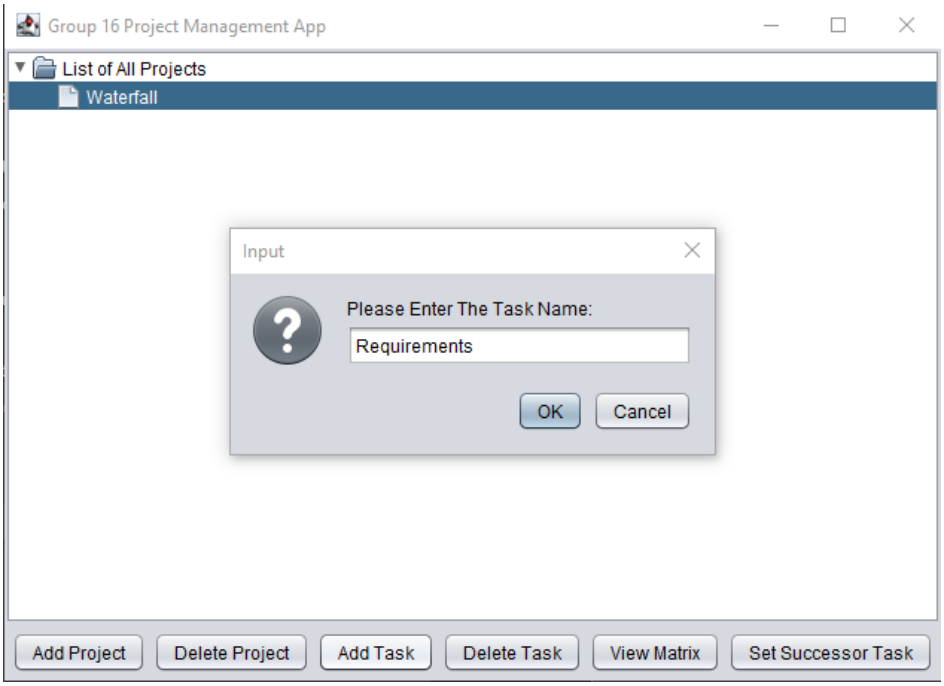


Figure 5: Adding duration to the task.

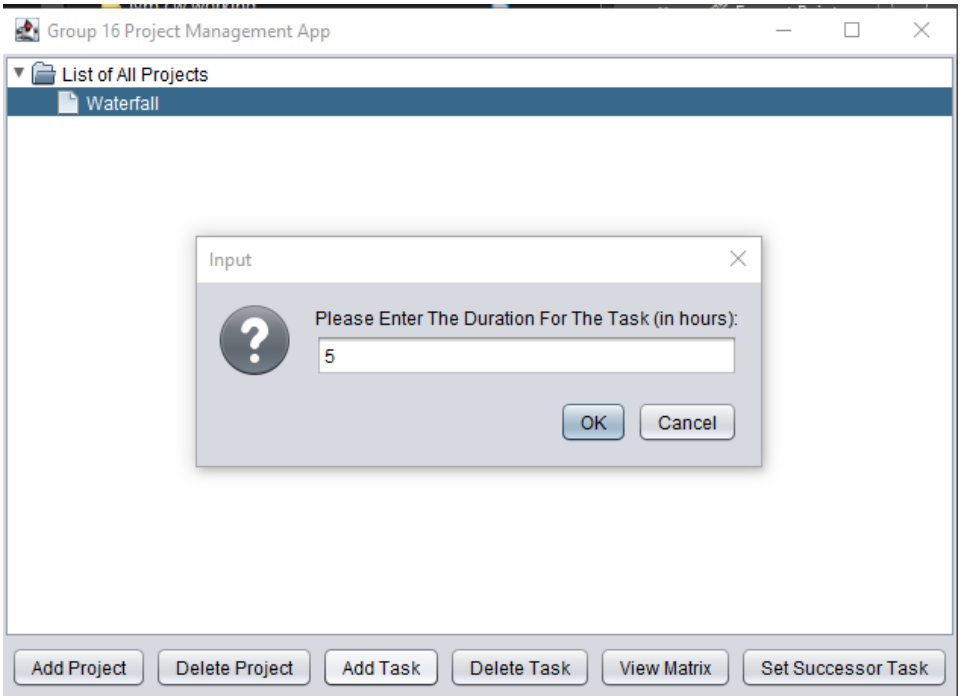


Figure 6: Repeat until we have 5 tasks in a project.

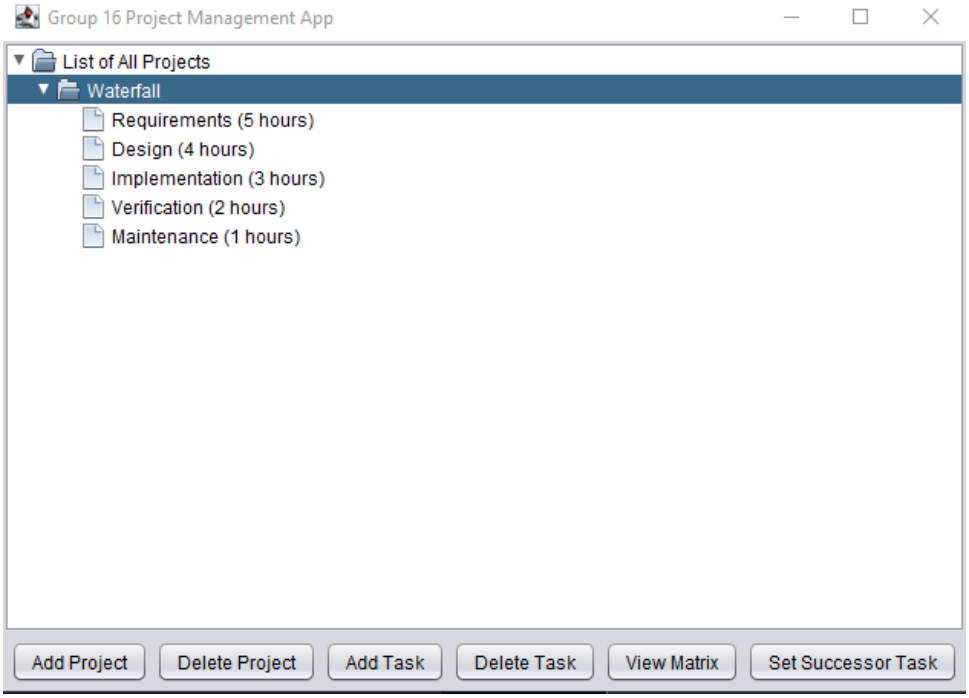


Figure 7: Setting the successor tasks.

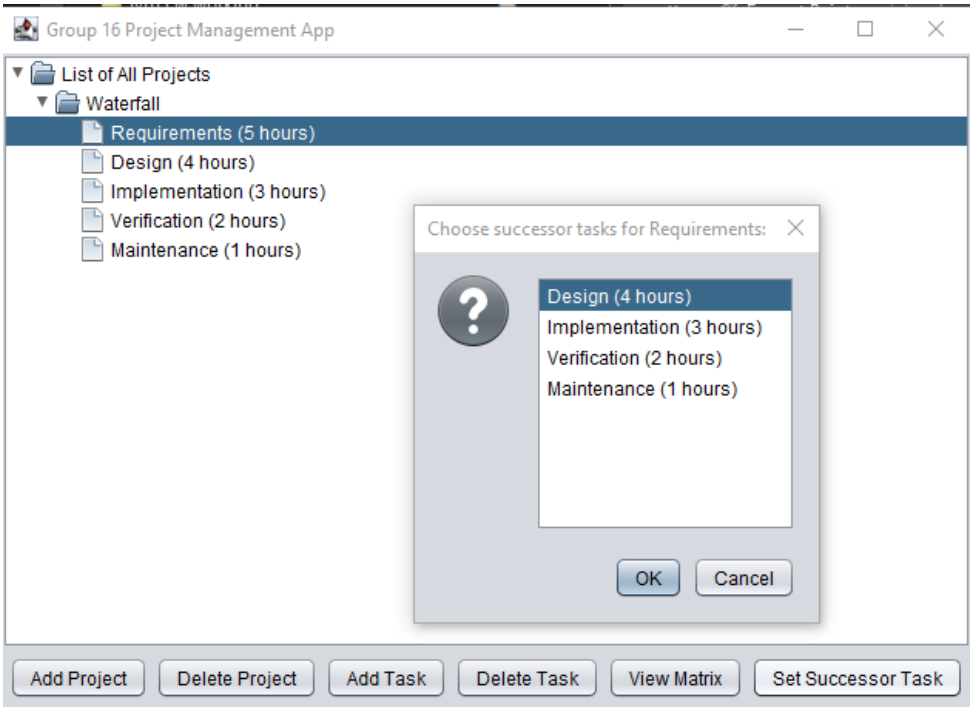


Figure 8: Repeat until we have given every task except the last task, a successor.

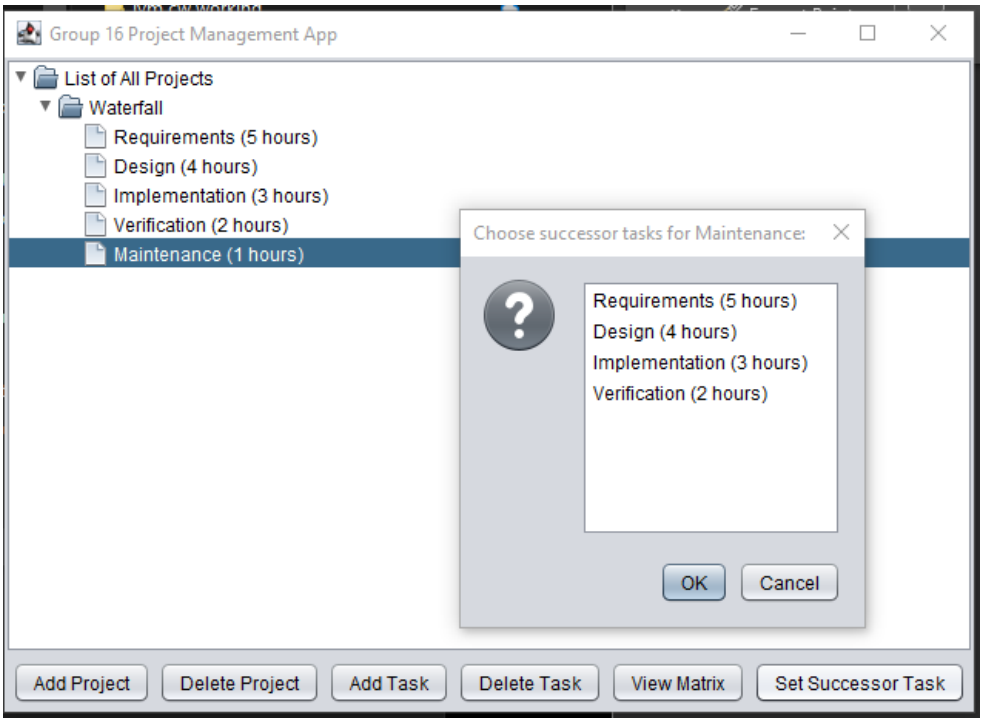


Figure 9: Displaying the generated adjacency matrix of tasks of a project.

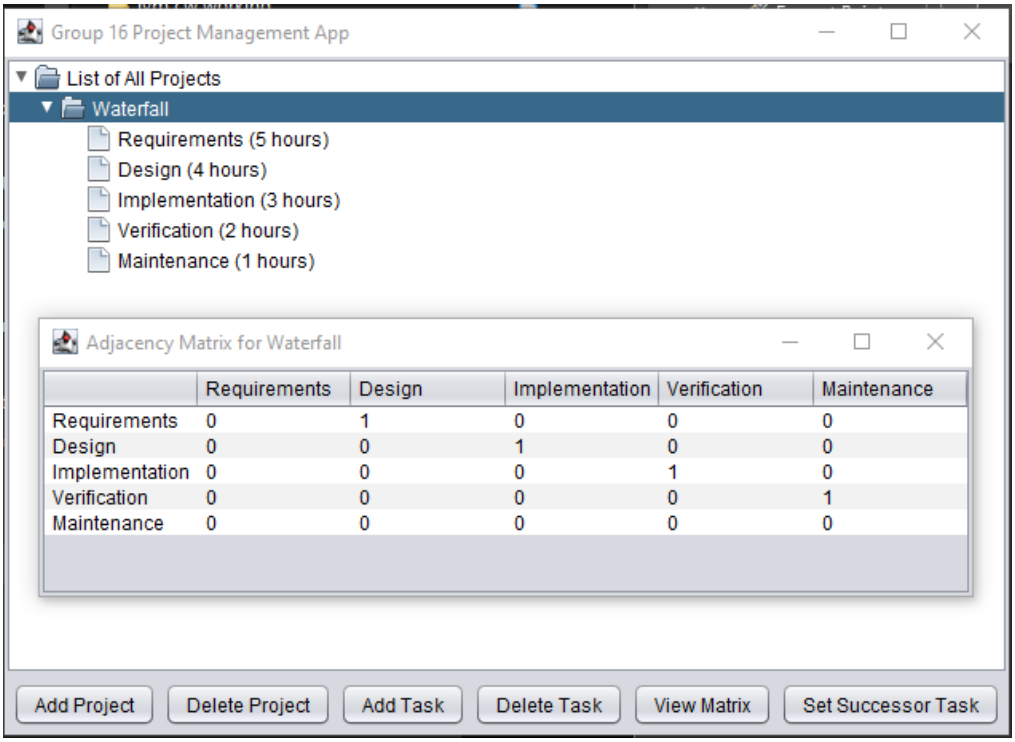


Figure 10: Giving a task more than one successor.

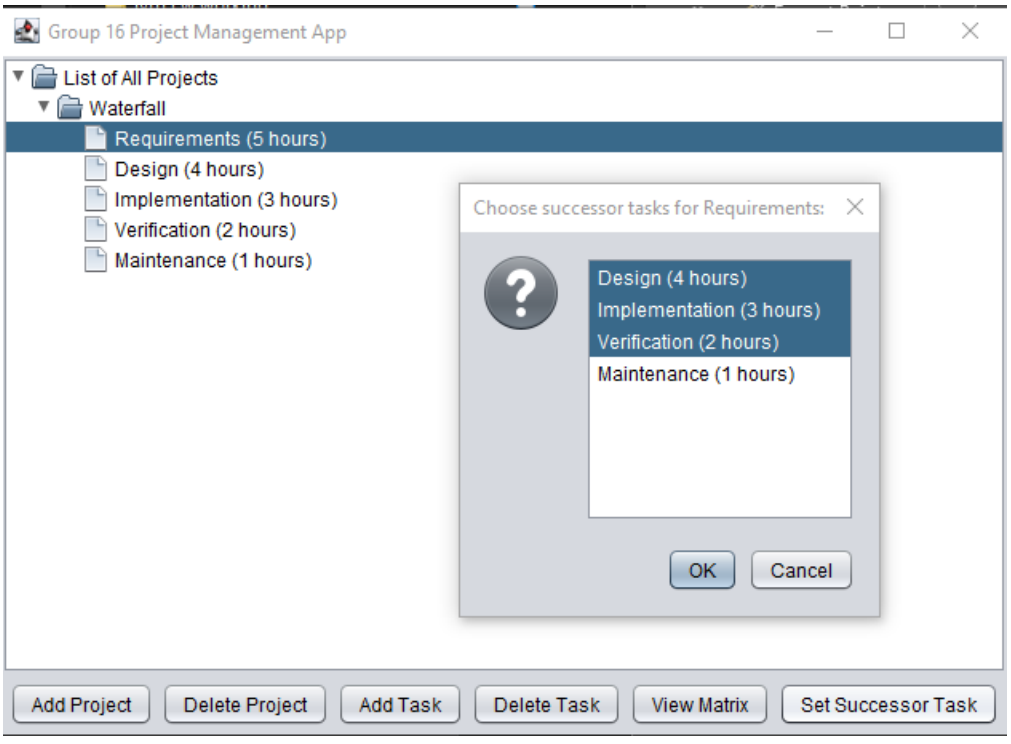
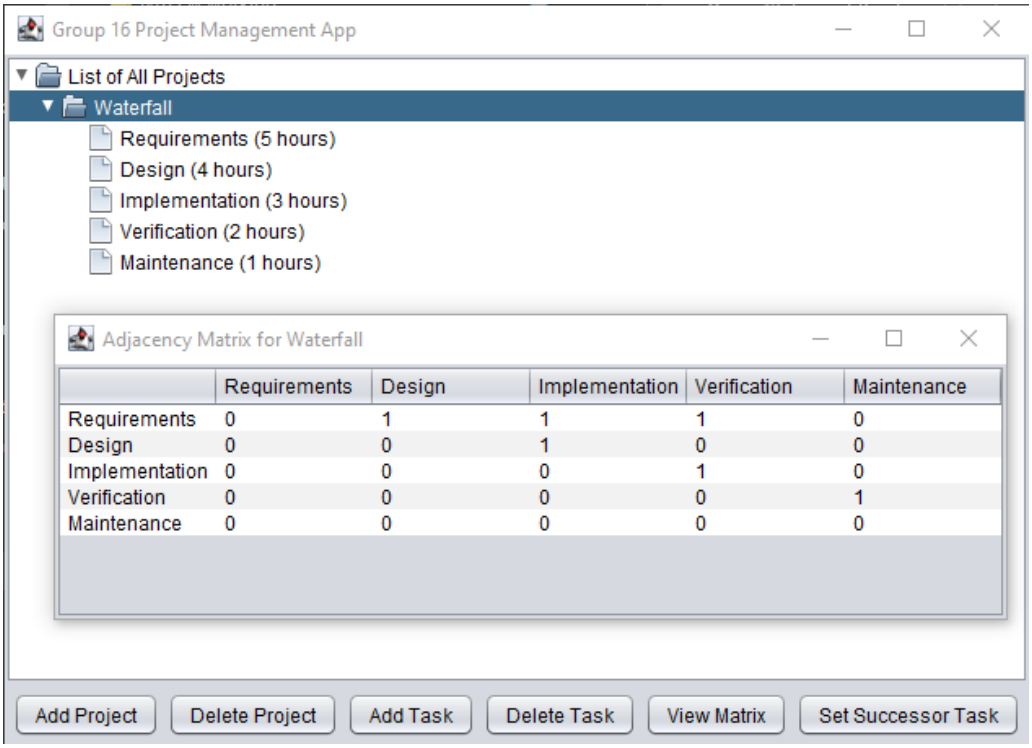


Figure 11: Showing the matrix one more time with updated successors.



	Requirements	Design	Implementation	Verification	Maintenance
Requirements	0	1	1	1	0
Design	0	0	1	0	0
Implementation	0	0	0	1	0
Verification	0	0	0	0	1
Maintenance	0	0	0	0	0

Section 3: Code listings

ProjectManager.Java [JAVA class]

```
import kotlin.Pair;
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import java.awt.*;
import java.awt.event.ActionListener;
import java.util.List;
public class ProjectManager {

    // These are the GUI Components
    private JFrame frame;
    private JTree tree;

    // For the tree view of the projects and tasks
    private DefaultMutableTreeNode rootNode;
```

```
private DefaultTreeModel treeModel;

public ProjectManager() {
    initialize();
    loadFromJSON();
}

// setting the style of the application using nimbus look and feel from the javax swing library
private void initialize() {
    try {
        UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
    } catch (Exception e) {
        e.printStackTrace();
    }

    frame = new JFrame();
    frame.setTitle("Group 16 Project Management App");
    frame.setBounds(100, 100, 700, 400);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLayout(new BorderLayout());

    // Sets up the project and task tree view
    rootNode = new DefaultMutableTreeNode("List of All Projects");
    treeModel = new DefaultTreeModel(rootNode);
    tree = new JTree(treeModel);
    frame.add(new JScrollPane(tree), BorderLayout.CENTER);

    JPanel panel = new JPanel();
    panel.setLayout(new FlowLayout());

    // adding all the buttons and their respective action listeners to the bottom of the gui frame

    addButton(panel, "Add Project", e -> addProject());
```

```
        addButton(panel, "Delete Project", e -> deleteProject());
        addButton(panel, "Add Task", e -> addTask());
        addButton(panel, "Delete Task", e -> removeTask());
        addButton(panel, "View Matrix", e -> showMatrix());
        addButton(panel, "Set Successor Task", e -> setSuccessor());

        frame.add(panel, BorderLayout.SOUTH);

        frame.pack();
    }

    // Function for creating all 6 buttons
    private void addButton(JPanel panel, String text, ActionListener actionListener) {
        JButton button = new JButton(text);
        panel.add(button);
        button.addActionListener(actionListener);
    }

    // function that allows the user to input a new project name,
    // creating a project with that name, adding it to the GUI,
    // and saving this addition so that it's remembered later.
    private void addProject() {
        String projectName = JOptionPane.showInputDialog(frame, "Please Enter The Project Name:");
        if (projectName != null && !projectName.trim().isEmpty()) { //not null and no empty spaces
            Project project = new Project(projectName); //calling the primary constructor of the Project.kt class, creating an object called project and giving it user input attributes such as name
            DefaultMutableTreeNode projectNode = new DefaultMutableTreeNode(project); // we create a tree node for the project
            rootNode.add(projectNode); // we add it to the rootNode
            treeModel.reload();
            saveToJSON();
        }
    }
```

```
// This is the method to delete a project
private void deleteProject() {
    DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
    if (selectedNode != null && selectedNode.getUserObject() instanceof Project) {
        rootNode.remove(selectedNode);
        treeModel.reload();
        saveToJSON();
    }
}

// This is the method to add a task to the selected project
private void addTask() {
    DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
//gets the currently selected item (node)(project) in the project tree and display in the GUI.
    //checks if user object "selectedNode" is not null and if it is an instance of the "Project" class. if so, it is cast to a
Project type and stores it in the "selectedProject" variable
    if (selectedNode != null && selectedNode.getUserObject() instanceof Project selectedProject) {
        String taskName = JOptionPane.showInputDialog(frame, "Please Enter The Task Name:");
        if (taskName != null && !taskName.trim().isEmpty()) {
            String durationString = JOptionPane.showInputDialog(frame, "Please Enter The Duration For The Task (in
hours):");
            if (durationString != null && !durationString.trim().isEmpty()) {
                try {
                    int duration = Integer.parseInt(durationString);
                    //calling the primary constructor of the Task.kt class, creating an object called task and giving it
attributes such as name and duration
                    Task task = new Task(taskName, duration);
                    //we call method addTask from Project.kt on the selectedProject variable, and "task" object is an
instance of the Task.kt class
                    selectedProject.addTask(task); // adds the task to the collection of tasks managed by selectedProject
                    DefaultMutableTreeNode taskNode = new DefaultMutableTreeNode(task);
                    selectedNode.add(taskNode);
                    treeModel.reload();
                    saveToJSON();
                } catch (NumberFormatException ex) {
```

```
                JOptionPane.showMessageDialog(frame, "Invalid duration. Please enter a valid number.", "Error",
JOptionPane.ERROR_MESSAGE);
            }
        }
    }
}

// This is the method to delete a task from the selected project
private void removeTask() {

    DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();



    if (selectedNode != null && selectedNode.getUserObject() instanceof Task) {

        DefaultMutableTreeNode parentProjectNode = (DefaultMutableTreeNode) selectedNode.getParent();
        if(parentProjectNode != null && parentProjectNode.getUserObject() instanceof Project parentProject) {
            Task selectedTask = (Task) selectedNode.getUserObject();



            parentProject.removeTask(selectedTask);



            parentProjectNode.remove(selectedNode);
            treeModel.reload();



            saveToJSON();
        }
    } else {
        JOptionPane.showMessageDialog(frame, "Please select a task to remove.", "Error",
JOptionPane.ERROR_MESSAGE);
    }
```

```
}

// This is the method to call the matrix and create it to a gui output representation
private void showMatrix() {
    DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
    if (selectedNode != null && selectedNode.getUserObject() instanceof Project selectedProject) {

        // Gets the task from the selected project
        List<Task> tasks = selectedProject.getAllTasks();

        // Uses the MatrixGenerator function in Kotlin to generate the adjacency matrix
        int[][] adjacencyMatrix = MatrixGenerator.Companion.generateAdjacencyMatrix(tasks);


        String[] columnNames = new String[tasks.size() + 1];

        columnNames[0] = ""; // The first column is used for row headers (task names)
        for (int i = 0; i < tasks.size(); i++) {

            columnNames[i + 1] = tasks.get(i).getName(); // Shifts the task names to the right by one
        }

        Object[][] matrixData = new Object[tasks.size()][tasks.size() + 1];
        for (int i = 0; i < tasks.size(); i++) {

            matrixData[i][0] = tasks.get(i).getName(); // Sets the task names as the first column of each row

            for (int j = 0; j < tasks.size(); j++) {

                matrixData[i][j + 1] = adjacencyMatrix[i][j]; // nested for loop, shifts the adjacency matrix to the right by
one
            }
```

```
        }

        JTable table = new JTable(matrixData, columnNames);

        table.getTableHeader().setReorderingAllowed(false); // Prevent reordering of columns
        JScrollPane scrollPane = new JScrollPane(table);

        JFrame matrixFrame = new JFrame("Adjacency Matrix for " + selectedProject.getName());
        matrixFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        matrixFrame.add(scrollPane);
        matrixFrame.setSize(600, 600);
        matrixFrame.setLocationRelativeTo(null);
        matrixFrame.setVisible(true);
      } else {
        JOptionPane.showMessageDialog(frame, "Please select a project to view its adjacency matrix.", "Error",
JOptionPane.ERROR_MESSAGE);
      }
    }



  // Method to set or update the successor tasks for a selected task in the GUI. involves displaying a list of
  // possible successor tasks, letting the user select from this list, and then saving the user's selections
  // as the new set of successors for the task

  private void setSuccessor() {
      DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent(); //
Get the currently selected node from the tree (GUI).
      if (selectedNode != null && selectedNode.getUserObject() instanceof Task selectedTask) { // selectedTask is a
variable that is set as an instance of Task class
        DefaultMutableTreeNode parentNode = (DefaultMutableTreeNode) selectedNode.getParent(); // Get the
parent node of the selected task, which should be a project.
        Project parentProject = (Project) parentNode.getUserObject(); // Retrieve the project object from the parent
node.
```

```
        // retrieves a pair containing a list of potential successor tasks and their corresponding indices for a selected task within a given project.

        Pair<List<Task>, List<Integer>> data = selectedTask.fetchTaskSuccessors(parentProject);

        List<Task> tasks = data.getFirst(); //get the first element of the pair data, which is a list of tasks, excluding the selected task.

        List<Integer> currentSuccessorsIndices = data.getSecond(); // get the second element of the pair data, which is a list of indices representing the current successors of the selected task.


        int[] indicesArray = currentSuccessorsIndices.stream().mapToInt(Integer::intValue).toArray(); // Converts List<Integer> to int[]


        DefaultListModel<Task> listModel = new DefaultListModel<>(); // Creates a DefaultListModel and populates it with tasks

        for (Task task : tasks) {

           listModel.addElement(task);

        }

        JList<Task> taskJList = new JList<>(listModel); // we create a JList for task selection, setting the selected indices to current successors.

        taskJList.setSelectedIndices(indicesArray);


        int result = JOptionPane.showConfirmDialog(

              frame,

              new JScrollPane(taskJList),

              "Choose successor tasks for " + selectedTask.getName() + ":",

              JOptionPane.OK_CANCEL_OPTION

        );


        if (result == JOptionPane.OK_OPTION) {

           List<Task> chosenTasks = taskJList.getSelectedValuesList();

           selectedTask.updateTaskSuccessors(chosenTasks);

           saveToJSON();

        }

     }

  }
```

```
    // calling primary constructor of PersistenceManager class and creating an instance of the class
    private final PersistenceManager persistenceManager = new PersistenceManager();

    private void saveToJSON() {
        persistenceManager.saveToJSON(rootNode);
    }

    private void loadFromJSON() {
        boolean success = persistenceManager.loadFromJSON(rootNode);
        if (success) {
            treeModel.reload();
        } else {
            JOptionPane.showMessageDialog(null,
                    "Error loading data from file!",
                    "Load Error",
                    JOptionPane.ERROR_MESSAGE);
        }
    }

    // Main method to run the application
    public static void main(String[] args) {
        EventQueue.invokeLater(() -> {
            try {
                ProjectManager window = new ProjectManager();
                window.frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }
}
```

Project.kt [KOTLIN class]

```
import java.io.Serializable

// serializable, allowing instances to be saved to and loaded from a file

// project class represents a project which contains a collection of tasks.

data class Project(
    val name: String) : Serializable {

    // data structure: tasks: a mutable list which is a dynamic list structure for storing Task objects
    // done to manage a collection of tasks within a project

    val tasks: MutableList<Task> = mutableListOf()



    @Synchronized
    fun addTask(task: Task) { // adding a task to the selected project
        tasks.add(task)
    }

    @Synchronized
    fun getAllTasks(): List<Task> { //Retrieves a list containing all the tasks
        return tasks.toList()
    }



    @Synchronized
    fun removeTask(task: Task) { // removes the selected task from the current project
        tasks.remove(task)
    }
```

```
    @Synchronized
    override fun toString(): String { // returns a string that contains the project name on the GUI
        return name
    }

}
```

Task.kt [KOTLIN class]

```
import java.io.Serializable

// serializable, allowing instances to be saved to and loaded from a file

// task class represents a task with a name and duration. also tracks successor tasks


data class Task(
    val name: String,
    val duration: Int) : Serializable {

    // data structure: successors: a private mutable list that tracks tasks that depend on the completing of the current task.

    private val successors: MutableList<Task> = mutableListOf()


    @Synchronized
    fun addSuccessor(task: Task) { // task is the task to be added as a successor
        if (task != this && !successors.contains(task)) {
            successors.add(task)
        }
    }

    @Synchronized
```

```
    fun clearSuccessors() { // Clears the list of successor tasks, removing all successors
        successors.clear()
    }

    @Synchronized
    fun getSuccessors(): List<Task> { //Retrieves a list containing all successor tasks
        return successors.toList()
    }

    @Synchronized
    //fetches the successor tasks of a given task within a project and returns a pair containing the list of tasks
    // and a list of indices indicating which tasks are the current successors
    fun fetchTaskSuccessors(parentProject: Project): Pair<List<Task>, List<Int>> {
        val tasks = parentProject.tasks.filterNot { it == this }
        val currentSuccessors = this.successors
        val indices = tasks.mapIndexed { index, task ->
            if (currentSuccessors.contains(task)) index else null
        }.filterNotNull()

        return Pair(tasks, indices)
    }

    @Synchronized
    // updates the successors for a chosen task
    // "chosenTasks" is a list of tasks that should be set as successor for the selected task.
    fun updateTaskSuccessors(chosenTasks: MutableList<Task>) { // takes a list of tasks, named chosenTasks, which are the tasks you want to follow after the current one
        this.clearSuccessors() // clears the current task's existing "next-in-line" tasks
        for (task in chosenTasks) {
            this.addSuccessor(task) // For each task in the new list, it adds it to the current task's "next-in-line" tasks
        }
    }
```

```
    @Synchronized
    override fun toString(): String { // returns a string that contains the tasks name and duration, on the GUI
        return "$name ($duration hours)"
    }
}
```

PersistenceManager.kt [KOTLIN class]

```
//Necessary libraries
import org.json.JSONArray
import org.json.JSONObject
import java.io.FileWriter
import java.nio.file.Files
import java.nio.file.Paths
import javax.swing.tree.DefaultMutableTreeNode

//This class deals with saving our project data and loading it back, as JSON objects
//It provides functionalities to convert tree nodes into a JSON representation and vice versa

class PersistenceManager {

    // This method converts the provided root tree node into a JSON representation
    // and writes it to a file named "projects.json"
    // main parameter is "rootNode", which is the root tree node representing a collection of projects

    fun saveToJSON(rootNode: DefaultMutableTreeNode) {
        val projectsArray = JSONArray()

        for (i in 0 until rootNode.childCount) { // Loops through all the project nodes and save their details
            val projectNode = rootNode.getChildAt(i) as DefaultMutableTreeNode
            val project = projectNode.userObject as Project
            val projectObject = JSONObject().apply {
                put("name", project.name)
```

```
        }

        val tasksArray = JSONArray()

        for (j in 0 until projectNode.childCount) { // Loops through all the task nodes within a project node, saving each task in the project
            val taskNode = projectNode.getChildAt(j) as DefaultMutableTreeNode
            val task = taskNode.userObject as Task
            val taskObject = JSONObject().apply {
                put("name", task.name)
                put("duration", task.duration)

                //used ChatGPT to try help with saving successor relationships to json
                val successorsArray = JSONArray()
                task.getSuccessors().forEach { successor ->
                    successorsArray.put(successor.name)
                }
                put("successors", successorsArray)
                //end of ChatGPT code
            }

            tasksArray.put(taskObject)
        }

        projectObject.put("tasks", tasksArray)
        projectsArray.put(projectObject)
    }

    try { // Writes the JSON representation to a file.
        FileWriter("projects.json").use { file -> file.write(projectsArray.toString()) }
    } catch (e: Exception) {
        e.printStackTrace()
    }
```

```
    }

    fun loadFromJSON(rootNode: DefaultMutableTreeNode): Boolean { // Loads projects and tasks from a file named "projects.json" and reconstructs the tree representation
        try {
            val content = String(Files.readAllBytes(Paths.get("projects.json")))
            val projectsArray = JSONArray(content)

            rootNode.removeAllChildren()

            for (i in 0 until projectsArray.length()) { // Loops through each project in the JSON array and build the tree representation
                val projectObject = projectsArray.getJSONObject(i)
                val projectName = projectObject.getString("name")
                val project = Project(projectName)
                val projectNode = DefaultMutableTreeNode(project)

                val tasksMap = mutableMapOf<String, Task>()

                val tasksArray = projectObject.getJSONArray("tasks")
                for (j in 0 until tasksArray.length()) { // Loops through each task within a project and add it to the project node
                    val taskObject = tasksArray.getJSONObject(j)
                    val taskName = taskObject.getString("name")
                    val taskDuration = taskObject.getInt("duration")
                    val task = Task(taskName, taskDuration)
                    tasksMap[taskName] = task
                    val taskNode = DefaultMutableTreeNode(task)

                    projectNode.add(taskNode)
                }
                //ChatGPT again for saving successor
                tasksArray.forEach { taskObject ->
                    val task = tasksMap[(taskObject as JSONObject).getString("name")]!!
```

```
                val successorsArray = taskObject.getJSONArray("successors")
                for (k in 0 until successorsArray.length()) {
                    val successorName = successorsArray.getString(k)
                    task.addSuccessor(tasksMap[successorName]!!)
                }
            }//end of ChatGPT code

            rootNode.add(projectNode)
        }
        return true
    } catch (e: Exception) {
        e.printStackTrace()
        return false
    }
  }
}
```

MatrixGenerator.kt [KOTLIN class]

```
class MatrixGenerator {
  //"i" is loop variable for row index(normal list of tasks)
  //"j" is loop variable for column index(successors)

  //Singleton object (like static method in java)
  companion object {
    fun generateAdjacencyMatrix(tasks: List<Task>): Array<IntArray> {
      //determine size of matrix, which is the number of tasks
      val n = tasks.size
      //creates array using number of tasks (n), and sets it with all 0's
      val adjacencyMatrix = Array(n) { IntArray(n) { 0 } }
      //loop over all tasks using their indices
      for (i in tasks.indices) {
        //retrieves current task based on its index
```

```
            val task = tasks[i]
            //loops over each successor of the current task
            for (successor in task.getSuccessors()) {
                //finds index of successor task in the list of tasks
                val j = tasks.indexOf(successor)
                // ensures if successor task is present in the list, then we set loop variable i and j as 1
                //if j is equal to -1,no successor was found in the list, so we leave it as 0
                if (j != -1) {
                    adjacencyMatrix[i][j] = 1
                }
            }
        }
        return adjacencyMatrix // 2D array of integers
    }
  }
}
```

Section 4: CW contribution form

Team member name	Student ID	individual overall work contribution (%)	Note
Student: Santa Sian	001108298	25	-
Student: Mihail Marinov	001141808	25	-
Student: Lyubomira Dimitrova	001180172	25	-
Student: Viktorija Seliviorstova	001152701	25	-
Total 100%			