IZMIR UNIVERSITY OF ECONOMICS

Workflow Application

Prepared by:

Mehmet Gülbahar
Hulki Enes Uysal
Selin Doğa Orhan
Mehmet Arda Uçar

Supervisor:
Kutluhan Erol

Abstract

This project focuses on the discrete event simulation of workflows, which are applicable to a variety of domains such as government offices, banks, hospitals, and factories. The aim is to model and simulate processes like obtaining a power of attorney document at a notary, where tasks such as ID verification, mental health document checks, photo capturing, document printing, signing, and fee payment are executed in sequence. Similarly, workflows in factories where raw materials progress through stations to become finished goods are considered.

Key entities in the simulation are jobs, each defined by a type and duration for completion. Jobs consist of sequential tasks that must be executed one at a time. Tasks vary by type and size, and their execution time depends on the size and station speed. Stations, that perform these tasks, vary in capability: some handle specific task types or multiple tasks simultaneously, while others impose constraints such as uniform task types. Station speeds may be constant or randomly vary within a defined range, influencing task execution durations.

This simulation framework provides a robust mechanism to analyze and optimize workflows by modeling real-world constraints and variations, offering insights into operational efficiency and resource utilization.

Contents

1 Introduction

This section introduces the project, its scope, and objectives. The Workflow Application project aims to simulate and optimize workflows using discrete event simulation techniques. The focus is on modeling real-world processes in various domains such as government offices and factories.

1.1 Problem Statement

Workflows in various industries, such as government offices and manufacturing plants, are often complex, involving multiple tasks and resources. Inefficiencies arise when workflows are not optimized, leading to delays, increased costs, and reduced productivity. The challenge lies in managing dependencies between tasks, allocating resources efficiently, and tracking the state of jobs and stations in real time. This project addresses these challenges by providing a simulation framework to analyze and optimize workflows.

1.2 Motivation

The motivation behind this project stems from the need to improve operational efficiency and resource utilization in industries where workflows play a critical role. By simulating workflows, organizations can identify bottlenecks, test different scheduling strategies, and predict the impact of changes before implementing them in the real world. This not only saves time and cost but also enables better decision-making. Additionally, the project aims to provide a flexible simulation tool that can be adapted to various domains, such as healthcare, banking, and manufacturing, showcasing its wide applicability and potential benefits.

2 Functional Requirements

2.1 Input and Error Handling

The program must handle inputs robustly. It reads workflow and job files from the command line and checks for their existence and accessibility. Errors such as missing files are reported descriptively to ensure proper debugging.

2.2 Parsing and Validation

Workflow files are parsed to extract task types, job types, and station details. Syntax errors are flagged with descriptive messages and line numbers, while semantic errors, such as duplicate IDs or out-of-range values, are highlighted. Similarly, job files are validated for proper IDs, start times, and durations to ensure data integrity before execution.

2.3 System State Tracking

The system dynamically tracks the state of jobs and stations. Job states include waiting, execution, and completion, while station states cover idle and busy statuses. This real-time tracking allows for accurate simulation and debugging.

2.4 Event Queue Management

Discrete event simulation is achieved using an event queue, where the system transitions directly to the next event rather than incrementing time steps. Events include task completions, job insertions, and state updates, enabling efficient simulation of real-world workflows.

2.5 Task and Station Management

Tasks are routed to appropriate stations based on their capabilities. Stations handle tasks according to predefined rules, such as first-come-first-serve or earliest-deadline-first. This ensures flexibility and adherence to workflow constraints. The system is also designed to accommodate future extensions, such as implementing alternative scheduling strategies.

2.6 Performance Metrics

At the end of the simulation, the program computes key metrics, including: - Average tardiness for late jobs, grouped by job type. - Station utilization rates reflect the percentage of time stations that are actively processing tasks.

3 Design and Implementation

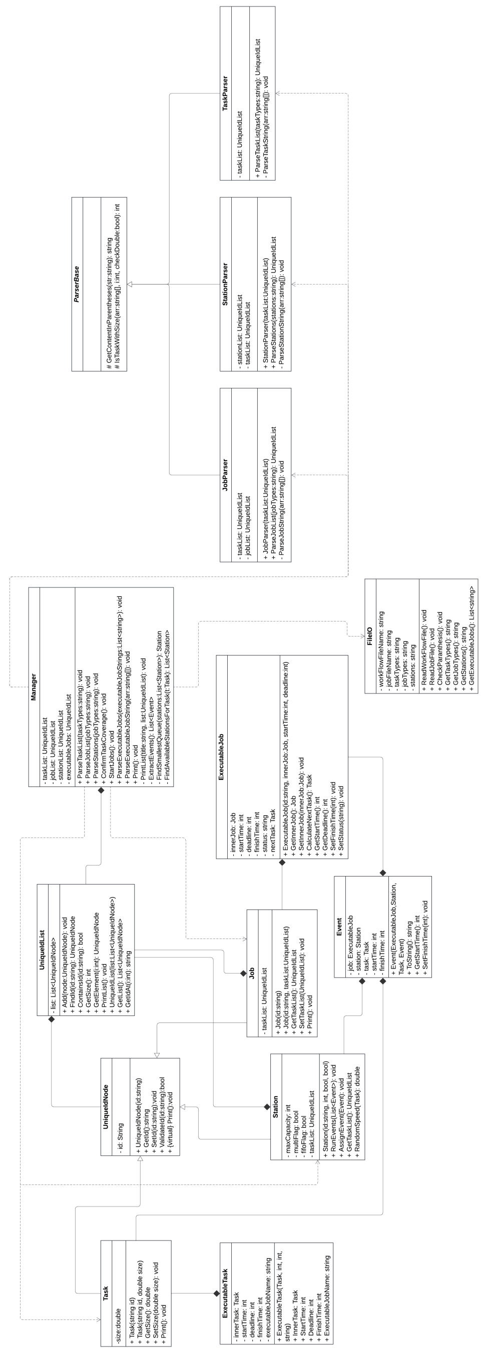

Figure 1: UML Class Diagram

3.1 UML Diagram Design and Analysis

Introduction: This report analyzes the classes, their relationships, and design decisions illustrated in the provided UML diagram. It provides a detailed examination of the use of inheritance and composition approaches, including the rationale and advantages behind these choices.

General Design Philosophy: The primary design goals of this system are:

1. **Modularity:** Ensuring classes operate independently while maintaining compatibility.
2. **Reusability:** Defining common functionalities in centralized classes to reduce code duplication.
3. **Extensibility:** Facilitating the easy addition of new features to the system.
4. **Data Integrity:** Guaranteeing the accuracy and uniqueness of all data.

Areas of Inheritance and Their Justifications:
UniqueIdNode Class:

- **Features:**
  - Serves as a base class for all entities requiring a unique identifier ("id").
  - Provides shared functionalities such as ValidateId, GetId, and SetId.
- **Classes Utilizing Inheritance:**
  1. Task: Each task has a unique identifier.
  2. Job: Each job has a unique identifier and a list of associated tasks.
  3. Station: Each station has a unique identifier and executes specific tasks.
  4. ExecutableJob: Represents executable jobs and requires a unique identifier.
- **Why Inheritance Was Used:**
  - Reducing Code Duplication: The common need for unique identifiers across multiple classes eliminates the need to rewrite these functionalities.
  - Validation and Standardization: Ensures that all identifiers conform to a specific format, with centralized validation mechanisms.
  - Generalizability: New classes requiring unique identifiers can easily inherit from UniqueIdNode.

ParserBase Class:

- **Features:**
  - Designed as an abstract class to provide a framework for parsing different types of data (e.g., tasks, jobs, stations).
  - Includes methods like GetContentInParentheses and IsTaskWithSize.
- **Classes Utilizing Inheritance:**
  1. TaskParser: Parses task information.
  2. JobParser: Parses job information.
  3. StationParser: Parses station information.
- **Why Inheritance Was Used:**
  - Reducing Code Duplication: Common parsing logic is centralized in the base class (ParserBase).
  - Abstraction: Provides a uniform logic applicable to any data type, regardless of its specifics.
  - Flexibility: New data types can be parsed by simply deriving from ParserBase.

Areas of Composition and Their Justifications:
Manager Class:

- **Features:**
  - Acts as the controller, integrating and managing core system components like Task, Job, and Station.
  - Uses components such as taskList, jobList, and stationList.
- **Why Composition Was Used:**
  - Modularity: Each component operates independently and can be replaced or modified as needed.
  - Independence: The Manager class is not concerned with the internal details of its components; it merely orchestrates their interactions.
  - Ease of Extensibility: Adding new components, such as a new task list, involves minimal changes to the Manager class.

Event Class:

- **Features:**
  - Tracks when a job starts and completes at a specific station.
  - Utilizes components like ExecutableJob, Station, and Task.

- **Why Composition Was Used:**
  - Dynamic Structure: Each event represents different stations and tasks, necessitating a composition-based design.
  - Flexibility: Events depend only on the required components, ensuring that adding new event types does not disrupt the existing structure.

Conclusion: This design strategically employs inheritance and composition to create a modular, flexible, and scalable system. Inheritance centralizes common functionalities, while composition enables independent and customizable components. Together, these approaches ensure the system remains maintainable and adaptable for future expansions.

Design choices include efficient data structures for the event queue and algorithms for task routing and state tracking. The focus is on scalability and maintainability, ensuring the system handles complex workflows efficiently.

4 Challenges and Solutions

4.1 Handling Complex Workflows

Handling complex workflows requires simulating various task types and station capabilities, presenting unique challenges. To address these, robust parsing and validation mechanisms were implemented.

4.1.1 Code Example: Parsing Job Types

The following code demonstrates how job definitions are parsed and validated to ensure correctness and consistency:

```
public UniqueIdList ParseJobList(string jobTypes) {
    jobList = new UniqueIdList();
    if (!jobTypes.StartsWith("(JOBTYPES ") || !jobTypes.EndsWith(")")) {
        throw new Exception("Job types are not formatted correctly!");
    }
    string innerJobs = jobTypes.Substring(10);
    innerJobs = innerJobs.Substring(0, innerJobs.Length - 1);
    string jobStr = "";
    do {
        jobStr = GetContentInParentheses(innerJobs);
        if (jobStr == null) break;
        string[] parts = jobStr.Split(' ');
        ParseJobString(parts);
        innerJobs = innerJobs.Substring(innerJobs.IndexOf("(") + jobStr.Length + 2);
    } while (jobStr != null);
    return jobList;
}
```

This method ensures that the jobs are correctly parsed, and syntax issues are reported appropriately.

This method ensures that the jobs are correctly parsed, and syntax issues are reported appropriately.

4.2 Code Organization

4.2.1 Challenges

1. The relationships between numerous classes and functions led to increased complexity in the codebase. 2. Ensuring code readability and maintainability required proper organization and modularization. 3. Collaboration among team members working on the same code sometimes resulted in conflicts.

4.2.2 Solutions

Modular Project Structure: The project was organized into distinct modules, each responsible for a specific aspect of the system:

- **Core:**
  - **Base:** Contains foundational classes that support unique identifier management (e.g., UniqueIdNode, UniqueIdList).
  - **Models:** Includes core data models categorized into subfolders like:
    - **Event:** Responsible for event-related data (Event.cs).
    - **Jobs:** Manages job-specific data (Job.cs, ExecutableJob.cs).
    - **Stations:** Handles station-related data (Station.cs).
    - **Tasks:** Defines task-related logic (Task.cs, ExecutableTask.cs).
- **Services:**
  - **Managers:** Contains the Manager.cs file responsible for orchestrating the simulation process.
  - **Parsers:** Handles file validation and parsing logic through specialized parsers (e.g., JobParser.cs, StationParser.cs, TaskParser.cs, ParserBase.cs).
  - **Scheduling:** Implements scheduling functionalities for tasks and events (EventScheduler.cs, SpeedCalculator.cs).
- **Infrastructure:** Provides utility classes and file operations (FileIO.cs, Program.cs).
- **Data:** Stores example input files for workflows and jobs (jobFile.txt, workflowFile.txt).

This modular structure enhanced the maintainability and readability of the code by isolating responsibilities, minimizing interdependencies, and improving scalability.

5 Execution Output

Execution results demonstrate the program's functionality using sample workflows and job files. The following outputs were generated during simulation to validate the system's accuracy:

5.1 Task List

Task	**Size (units)**
T1	1
T2	2
T3	25
T4	1
T5	4
T_1	5
T21	1

5.2 Job List

Job	**Tasks in Job**
J1	T1 (1 units), T2 (2 units), T3 (3 units)
J2	T2 (2 units), T3 (25 units), T4 (1 units)
J3	T2 (2 units)
J4	T21 (5 units), T1 (2 units)

5.3 Station List

Station	**Capacity**	**Tasks Done**
S1	1	T1 (2 units/min), T2 (3 ± 2000% units/min)
S2	2	T1 (2 units/min), T2 (4 units/min)
S3	2	T3 (1 units/min)
S4	3	T4 (1 units/min), T21 (2 ± 5000% units/min)

5.4 Execution Timeline

Event	**Details**
T1 from J1	Assigned to S1
T2 from J1	Assigned to S2
T3 from J1	Assigned to S3
...	...
T2 from J5	Station S2 starts at time 10, finishes at time 11

6 Workflow and Job File Validation

6.1 Challenges

- Workflow and Job File Errors:
  - Identifying issues such as incorrectly formatted Task IDs or missing start times was challenging.
  - Misnamed or missing files caused the program to halt, disrupting the entire workflow simulation.

6.2 Solutions

6.2.1 Base Validation with ParserBase

The `ParserBase` class provided an abstract framework for basic validation and parsing logic, allowing specific parsers to inherit and implement specialized rules.

```
public abstract class ParserBase
{
    protected bool IsTaskWithSize(string input)
    {
        var parts = input.Split(',');
        return parts.Length == 2 && int.TryParse(parts[1], out _);
    }
    public abstract void Parse(string[] lines);
}
```

The `IsTaskWithSize` method validated the format of tasks, ensuring proper structure for input data.

6.2.2 Job Validation with JobParser

The `JobParser` class implemented validation rules for job files, ensuring they adhered to the expected format.

- **Key Features:**
  - Line-by-Line Validation: Checked that each line contained three components separated by ;
  - Error Logging: Identified and logged invalid formats or missing start times with detailed error messages.

```
public class JobParser : ParserBase
{
    public override void Parse(string[] lines)
    {
        foreach (var line in lines)
        {
            var parts = line.Split(';');
            if (parts.Length != 3)
            {
                LogError($"Invalid job format at line: {line}");
                continue;
            }

            if (!int.TryParse(parts[1], out _))
            {
                LogError($"Invalid start time in job: {line}");
            }
        }
    }
}
```

6.2.3 Specialized Validation with StationParser and TaskParser

StationParser: Validated station files to ensure tasks were correctly assigned and flags (FIFO, MultiFlag) were properly formatted.

```
public class StationParser : ParserBase
{
    public override void Parse(string[] lines)
    {
        foreach (var line in lines)
        {
            var parts = line.Split(';');
            if (parts.Length != 4 || !bool.TryParse(parts[2], out _))
            {
                LogError($"Invalid station format: {line}");
            }
        }
    }
}
```

TaskParser: Ensured task IDs were unique and correctly formatted, leveraging a `HashSet` to detect duplicates.

```
public class TaskParser : ParserBase
{
    public override void Parse(string[] lines)
    {
        var taskIds = new HashSet<string>();
        foreach (var line in lines)
        {
            if (!IsTaskWithSize(line))
            {
                LogError($"Invalid task format: {line}");
                continue;
            }

            var taskId = line.Split(',')[0];
            if (!taskIds.Add(taskId))
            {
                LogError($"Duplicate task ID: {taskId}");
            }
        }
    }
}
```

7 Event Queue Management

7.1 Challenges

- Managing time-ordered events caused conflicts, especially when multiple events occurred simultaneously.
- Determining priorities for overlapping events required a robust sorting algorithm.

7.2 Solutions

7.2.1 Time-Ordered Processing

The Event and EventScheduler classes were implemented to manage events effectively.

Event Class: Tracks `startTime` and `finishTime` to ensure chronological order.

```
public Event(ExecutableJob job, Station station, Task task,
Event previousEvent)
{
    this.job = job;
    this.station = station;
    this.task = task;
    this.previousEvent = previousEvent;
    this.startTime = job.GetStartTime();
    this.finishTime = -1;
}
```

EventScheduler Class: Processes events using the `RunEvents` method to handle multiple events and update states dynamically.

```
public void RunEvents(List<Event> eventList, Func<Task, double>
↪   randomSpeed,Action<string> log)
{     foreach (var currentEvent in executionQueue)
    {
        if (currentEvent.FinishTime == -1)
        {  double speed = randomSpeed(currentEvent.Task);
            int finish = (int)Math.Ceiling(
            currentEvent.StartTime +
            (currentEvent.Task.GetSize() / speed));
            currentEvent.FinishTime = finish;

            log($"Task {currentEvent.Task.GetId()} started at " +
                $"{currentEvent.StartTime} and finishes at {finish}");
            eventList.Remove(currentEvent);
        }
    }
}
```

7.3 Result Events were processed in chronological order. Priorities ensured proper handling of simultaneous events. Tasks and stations were updated dynamically during event execution.

7.3 Outcomes

- The system processed events in correct order.
- Resolved simultaneous event conflicts with prioritization.
- Ensured smooth simulation execution.

8 Conclusion

The Workflow Application successfully models and simulates real-world workflows, providing insights into operational efficiency and resource utilization. Future work includes integrating additional scheduling strategies and enhancing the system's scalability.