C22716399 Aaron Baggot

[Company name] | [Company address]

[Document title]

[Document subtitle]

Table of Contents

[Introduction 3](#_Toc163913838)

[Restaurant Selection and Use Case Scenarios 4](#_Toc163913839)

[Class Diagram with soil implementation 5](#_Toc163913840)

[Class Diagram 8](#_Toc163913841)

[Object Diagram 9](#_Toc163913842)

[Sequence Diagram 10](#_Toc163913843)

[BookingSystem 11](#_Toc163913844)

[Booking and Reservation Class 12](#_Toc163913845)

[Implementation 12](#_Toc163913846)

[Booking System SelectBooking() 13](#_Toc163913847)

[Booking System unSelectBooking() 14](#_Toc163913848)

[Booking System changeTable() 15](#_Toc163913849)

[BookingSystem makeReservation() 17](#_Toc163913850)

[BookingSystem cancelReservation() 19](#_Toc163913851)

[Booking System recordArrival() 21](#_Toc163913852)

[Arrival before the booking 23](#_Toc163913853)

[Original Sequence Diagram 24](#_Toc163913854)

[USE Model Overview 26](#_Toc163913855)

[Testing and Validation 27](#_Toc163913856)

[Booking System 27](#_Toc163913857)

[State Machines 28](#_Toc163913858)

[Booking System state machine 28](#_Toc163913859)

[selectBooking() 29](#_Toc163913860)

[unselectBooking() 30](#_Toc163913861)

[cancelBooking() 31](#_Toc163913862)

[Booking System Record arrival 31](#_Toc163913863)

[Pre and Post Conditions 32](#_Toc163913864)

[Booking System Record Arriva 32](#_Toc163913865)

[Booking System Cancel Conditions 33](#_Toc163913866)

[Cancel Pre-Condition Success 34](#_Toc163913867)

[Cancel post success 35](#_Toc163913868)

[Cancel pre success 35](#_Toc163913869)

[Cancel pre fail 35](#_Toc163913870)

[Booking System Change Table 36](#_Toc163913871)

[overNoOfCovers() 36](#_Toc163913872)

[Make Reservation – Underage Condition 40](#_Toc163913873)

[Conclusion 49](#_Toc163913874)

# Introduction

In the domain of software engineering, the efficacy and elegance of a system's design are paramount. It is within this context that our project focuses on enhancing and refining the existing USE model for a restaurant management system. The restaurant industry, with its complex dynamics involving customer interactions, table management, and payment processing, offers ample opportunities to explore various software engineering principles and methodologies.

Our main goal is to expand the functionality of the restaurant management system while maintaining a strong emphasis on coupling and cohesion principles. By introducing new features such as table reservation and dynamic table allocation, we aim to improve the system's flexibility and usefulness. Additionally, we aim to implement a robust payment system to ensure secure and efficient transaction handling.

At the heart of our approach lies the concept of design by contract, where each component of the system is equipped with preconditions, postconditions, and invariants. These contractual specifications serve as guidelines for the system's behaviour and facilitate thorough testing and validation. Through the use of OCL (Object Constraint Language) contracts, we define precise conditions for invoking operations, ensuring the system's reliability and resilience.

We employ a comprehensive methodology that includes class diagrams, sequence diagrams, state machines, and object diagrams to gain insights into the system's structure, behaviour, and interactions. Each of these artifacts contributes to a thorough analysis and refinement process. Additionally, we integrate testing mechanisms such as pre- and post-operation procedures to validate system constraints and operation behaviour rigorously.

Throughout this project, we provide a detailed exploration of the extended USE model, explaining the rationale behind each design decision and offering a comprehensive assessment of the system's functionality. By adhering to industry standards and utilizing advanced software engineering techniques, our aim is to develop restaurant management systems that are robust, scalable, and user-centric, exemplifying the highest standards of software engineering excellence.

# Restaurant Selection and Use Case Scenarios

In the realm of software engineering, creating systems that are both efficient and robust is essential. This report outlines the process of enhancing and refining the current USE model for a restaurant management system. The focus is on key functionalities such as booking reservations, handling walk-ins, and managing cancellations, all while integrating state machines and pre/post conditions. Through the exploration of various software engineering principles and methodologies, the aim is to improve the system's functionality and user experience.

As part of this project, additional use case scenarios have been incorporated to broaden the system's capabilities. These include features such as table reservation, dynamic table allocation, and payment processing. These scenarios offer a comprehensive view of how the system interacts with users and enhances its overall usability.

Make Reservation

This scenario involves customers reserving tables for dining at the restaurant. Customers provide personal details, select the date, time, and number of guests, and either choose a preferred table or allow the system to assign one. Upon confirmation, a reservation is created, and the customer receives a confirmation message.

Record Walk-In

In this scenario, customers arrive at the restaurant without prior reservations. The waitstaff checks table availability, assigns an available table to the customer, and seats them. This walk-in is then recorded in the system for tracking purposes.

Cancel Reservation

This scenario allows customers to cancel their existing reservations. After logging into the system, customers navigate to the reservation management section, select the reservation to cancel, and confirm the cancellation. Once cancelled, the reserved table becomes available for other customers.

Use Case Diagram

These scenarios are depicted in a use case diagram, illustrating how customers interact with the system to make reservations, record walk-ins, and cancel reservations. This diagram aligns with the functionalities implemented in the system, ensuring effective user interaction and management of restaurant bookings.

## Class Diagram with soil implementation

***-- Script generated by USE 6.0.0***

***!new BookingSystem('bookSys')***

***!new WalkIn('w1')***

***!new WalkIn('w2')***

***!new Reservation('r1')***

***!new Reservation('r2')***

***!new Reservation('r3')***

***!w1.covers := 3***

***!w2.covers := 4***

***!r1.covers := 4***

***!r2.covers := 2***

***!r3.covers := 2***

***!insert(bookSys, w1) into Contains***

***!insert(bookSys, w2) into Contains***

***!insert(bookSys, r1) into Contains***

***!insert(bookSys, r2) into Contains***

***!insert(bookSys, r3) into Contains***

***!new Customer('c1')***

***!new Customer('c2')***

***!new Customer('c3')***

***!c1.name := 'Aniket Bedade'***

***!c1.phoneNumber := 'C22448826'***

***!c1.age := 17***

***!c2.name := 'Aaron Baggot!'***

***!c2.phoneNumber := 'C22716399'***

***!c2.age := 48***

***!c3.name := 'John Smith'***

***!c3.phoneNumber := '086 123 1234'***

***!c3.age := 25***

***!r1.setCustomer(c1)***

***!r2.setCustomer(c2)***

***!r3.setCustomer(c3)***

***!new Table('tb1')***

***!new Table('tb2')***

***!new Table('tb3')***

***!new Table('tb4')***

***!tb1.tno := 1***

***!tb2.tno := 2***

***!tb3.tno := 3***

***!tb4.tno := 4***

***!tb1.covers := 4***

***!tb2.covers := 2***

***!tb3.covers := 2***

***!tb4.covers := 6***

***!insert (w1,tb1) into IsAt***

***!insert (r1,tb3) into IsAt***

***!insert (r2,tb2) into IsAt***

***!insert (w2,tb2) into IsAt***

***!new Time('t1')***

***!new Time('t2')***

***!new Time('t3')***

***!new Time('t4')***

***!new Time('t5')***

***!t2.min := 0***

***!t2.hour := 18***

***!t1.min := 30***

***!t1.hour := 20***

***!t3.min := 30***

***!t3.hour := 18***

***!t4.min := 0***

***!t4.hour := 20***

***!t5.min := 0***

***!t5.hour := 21***

***!w1.setTime(t1)***

***!w2.setTime(t2)***

***!r1.setTime(t3)***

***!r2.setTime(t4)***

***!r3.setTime(t5)***

***!insert(bookSys, c1) into Uses***

***!insert(bookSys, c2) into Uses***

***!insert(bookSys, c3) into Uses***

The provided SOIL code essentially creates and links different objects to form a booking system for a restaurant. Creating various objects representing different aspects of the booking system, such as the BookingSystem itself, different types of bookings like WalkIn and Reservation, as well as entities like Customer, Table, and Time. These objects are like instances of classes and hold specific information related to bookings, customers, tables, and time.

Once these objects are created, the code sets specific attributes for each of them. For example, attributes like covers (number of people), time, name, age, phone number, and table number are assigned values to represent the characteristics of each instance.

After creating and configuring these objects, the code establishes connections or relationships between them. For instance, it links bookings with tables using the IsAt association, connects customers with reservations through the Makes association, records walk-ins with customers via the Arrive association, and links the booking system with its contained bookings using the Contains association.

The provided code constructs a functional booking system for a restaurant. It allows for the creation and management of various bookings, both reservations and walk-ins, enables customers to interact with the system, assigns tables to bookings, and organizes the overall booking process. This system is structured with different components like classes, operations, and constraints, ensuring a robust and comprehensive solution for managing restaurant bookings.

## Class Diagram

A class diagram is a type of static structure diagram in UML (Unified Modelling Language) that represents the structure of a system by showing the system's classes, their attributes, operations, and relationships.

A screenshot of a computer

Description automatically generated

## Object Diagram

An object diagram in USE represents a snapshot of the system at a specific moment, showing instances of classes and their relationships. It illustrates how objects interact and collaborate within the system, providing a visual depiction of runtime behaviour. Object diagrams help in understanding the runtime structure of the system and can be used to verify system design and implementation.

A screenshot of a computer

Description automatically generated

# Sequence Diagram

A sequence diagram in USE illustrates the interactions between different objects or components of a system over time. It shows the flow of messages between objects, indicating the order in which interactions occur. Sequence diagrams help visualize the dynamic behavior of a system, depicting how objects collaborate to achieve specific functionalities or scenarios.

A screenshot of a computer screen

Description automatically generated

# BookingSystem

Represents the main control class responsible for managing bookings, including selecting, unselecting, recording arrivals, cancelling reservations, and changing tables.

Booking:

Represents a generic booking with attributes such as the number of covers and time.

WalkIn:

Represents a walk-in booking, which inherits from Booking and includes additional operations specific to walk-ins.

Reservation:

Represents a reservation booking, which also inherits from Booking and includes operations specific to reservations.

Customer:

Represents a customer with attributes such as name, age, and phone number, and operations for making reservations and walk-ins.

Table:

Represents a restaurant table with attributes such as table number and covers, and a query operation to check availability.

Receptionist:

Is the control class that manages walk ins.

# Booking and Reservation Class

The Booking and Reservation classes represent different types of bookings, each with specific attributes and operations. State machines depict the transitions between booking states, such as new booking, waiting, and seated, providing a clear overview of the booking lifecycle.

create: Transition from the newReservation state to the waiting state when a new reservation is created.

setArrivalTime(): Transition from the waiting state to the seated state when the arrival time for the reservation is set.

## Implementation

The Reservation class contains attributes to represent each state.

Operations such as create and setArrivalTime() trigger transitions between states.

When a new reservation is created, the system transitions from the newReservation state to the waiting state.

Setting the arrival time for the reservation transitions it to the seated state, indicating that the customers have been seated.

## Booking System SelectBooking()

selectBooking: Selects a booking based on the table number and time provided.

A screenshot of a computer

Description automatically generated

## Booking System unSelectBooking()

unSelectBooking: Unselects the currently selected booking.

A screenshot of a computer

Description automatically generated

## Booking System changeTable()

BookingSystem::changeTable(r : Reservation, table : Table):

Precondition (overNoOfCovers): Checks if the covers required for the booking (r.covers) are less than or equal to the covers available at the new table (table.covers). This prevents moving a booking to a table that cannot accommodate the required covers.

Precondition (Pre1): Ensures that the current table of the booking (r.table) does not include the new table (r.table->excludes(table)), meaning the booking is not already assigned to the new table.

Postcondition (Post1): Ensures that after changing the table, the new table (table) includes the booking (r.table->includes(table)).

changeTable: Changes the table for a given reservation if the new table is available.

A screenshot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generated

Change table condition number of covers

A screen shot of a computer

Description automatically generated

## BookingSystem makeReservation()

Customer::makeReservation(covers : Integer, table : Table, bookingHr : Integer, bookingMin : Integer):

Precondition (underageBookingTime): Checks if the age of the customer (self.age) is less than 18 and if the provided booking time (bookingHr \* 60 + bookingMin) is before 9:00 PM (21:00 hours). This prevents underage customers from booking tables after 9:00 PM.

A screenshot of a computer

Description automatically generated

The provided command !c1.makeReservation(2,tb1,18,00) represents a reservation being made within the restaurant management system. It involves making a reservation for 2 guests at table "tb1" for 6:00 PM. The output "Reservation made successfully" confirms that the reservation was successfully processed and added to the system.

A screenshot of a computer

Description automatically generated

No overlap when making reservation

A screenshot of a computer

Description automatically generated

## BookingSystem cancelReservation()

BookingSystem::cancel(r : Reservation):

Precondition (Pre1): This checks if the booking to be cancelled (r) is included in the current bookings (current->includes(r)), ensuring that only existing bookings can be cancelled.

Postcondition (Post1): This ensures that after the cancellation operation, the cancelled booking (r) is no longer included in the current bookings (current->excludes(r)).

Sequence Diagram cancel reservation

**A screenshot of a computer screen

Description automatically generated**

Object Diagram cancel reservation

A screenshot of a computer

Description automatically generated

## Booking System recordArrival()

BookingSystem::recordArrival(hr: Integer, mn: Integer):

Precondition (arrivalBeforeBooking): This constraint ensures that the arrival time of the selected booking (self.selected.getTime()) is before or at the same time as the provided arrival time (mn + hr\*60). This is to prevent recording an arrival time that precedes the booking time.

**A screenshot of a computer

Description automatically generated**

### State Machine for Reservation

newReservation: Initial state when a new reservation is created.

waiting: Indicates that a reservation exists but the customers are yet to be seated.

seated: Represents a reservation where the customers have been seated.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Arrival before the booking

The test case attempts to execute the recordArrival operation on the BookingSystem object bookSys with a specified time of 17:00 (5:00 PM). However, the precondition arrivalBeforeBooking is not met, indicating that the selected booking's time (self.selected.getTime()) is not earlier than the provided time (17:00). As a result, the operation fails to execute, and an error is generated, preventing the record of the arrival.

A computer screen shot of a black screen

Description automatically generated

# USE Model Overview

Overview of the Unified Software Engineering (USE) modelling approach.

Explanation of how USE models will be utilized to design and test the adapted systems.

The extended USE model encompasses class diagrams, sequence diagrams, state machines, and object diagrams. Each artifact offers insights into the system's structure, behaviour, and interactions. We have meticulously defined preconditions, postconditions, and invariants for each component, ensuring a clear understanding of the system's behaviour.

# Testing and Validation

In the realm of software development, where precision and reliability are paramount, testing and validation are fundamental practices. When dealing with systems outlined using Use Case UML (Unified Modelling Language) and bound by OCL (Object Constraint Language), these procedures gain heightened importance. They act as safeguards, guaranteeing that the implemented systems seamlessly align with their intended functions and specifications.

By subjecting software systems to rigorous testing against their defined use cases, developers can confirm their functional accuracy. This entails closely examining every aspect of the system's behaviour to ensure it aligns with the expectations outlined in the UML diagrams. Additionally, validation against OCL constraints confirms that the system not only operates as intended but also adheres faithfully to the specified business rules and constraints.

This combined approach to testing and validation offers numerous advantages. It allows for the early detection of defects, reducing the likelihood of costly rework later in the development process. It also contributes to enhancing the user experience by identifying and addressing usability issues at an early stage. Moreover, these procedures help mitigate risks associated with system failures or non-compliance with business requirements.

# Booking System

The Booking System class manages booking operations such as selecting, unselecting, recording arrivals, and cancelling bookings. The state machine diagram illustrates the transitions between different states, providing a visual representation of the booking process.

The BookingSystem class serves as the main control unit, orchestrating various operations such as selecting bookings, recording arrivals, canceling reservations, and managing the state transitions of bookings.

SelectBooking()

Booking Management system allows customers to make reservations or walk-ins. Reservations are associated with specific customers and tables, with functionalities to set covers, record arrival times, and manage reservations.

Table Management are represented as objects with attributes such as table number and covers. The system ensures that tables are appropriately managed, transitioning between states of availability, reservation, and occupation.

Time Management class facilitates time-related operations, allowing the system to track booking times, record arrival times, and enforce constraints related to timing.

Customer Management allows customers can make reservations or walk-ins, with functionalities to specify covers, booking times, and contact information.

The Booking System class manages booking operations such as selecting, unselecting, recording arrivals, and cancelling bookings. The state machine diagram illustrates the transitions between different states, providing a visual representation of the booking process.

## State Machines

State machines play a crucial role in depicting the behaviour of various components within the system. They provide a visual representation of the states that objects can occupy and the transitions between these states. In our restaurant management system, state machines are used to model the lifecycle of bookings, tables, and other entities. By delineating the possible states and transitions, we gain a deeper understanding of how these entities behave in response to different operations and events.

### Booking System state machine

This state machine represents the lifecycle of a booking in the restaurant booking system.

States:

newBooking: Initial state when a new booking is created.

notSelected: Indicates that a booking exists but has not been selected.

selected: Represents a booking that has been selected.

Transitions:

newBooking to notSelected: Transition when a new booking is created.

notSelected to selected: Transition when a booking is selected via the selectBooking() operation.

selected to selected: Transition when a booking is already selected and the covers of the selected table are greater than or equal to the covers of the booking, resulting in recording the arrival via the recordArrival() operation.

selected to selected: Another transition when a booking is already selected, typically after recording arrival, possibly indicating a redundant selection.

selected to notSelected: Transition when a selected booking is unselected via the unSelectBooking() operation.

selected to notSelected: Transition when a selected booking is cancelled via the cancelReservation() operation.

notSelected to notSelected: Transition when an unselected booking is cancelled via the cancel() operation.

newBooking: This is the initial state when a new booking is created.

notSelected: Indicates that a booking exists but has not been selected.

selected: Represents a booking that has been selected.

newBooking -> notSelected { create }

selected -> selected { [self.selected.table.covers >= self.selected.covers] recordArrival() }

selected -> notSelected { unSelectBooking() }

### selectBooking()

selected -> selected { selectBooking() }

A screenshot of a computer

Description automatically generated

### unselectBooking()

notSelected -> selected { selectBooking() }

A screenshot of a computer

Description automatically generated

### cancelBooking()

selected -> notSelected { cancelReservation() }

notSelected -> notSelected { cancel() }

A screenshot of a computer

Description automatically generated

### Booking System Record arrival

A screenshot of a computer

Description automatically generated

### Receptionist Operation

Receptionist Greet Walkin

A screenshot of a computer

Description automatically generated

Receptionist Seat Walkin

A screenshot of a computer

Description automatically generated

Receptionist seat WalkIn Fail

Part 1

A computer screen shot of a computer screen

Description automatically generated

Part 2

A screenshot of a computer

Description automatically generated

Receptionist Sequence Diagram

Greet WalkIn

A screenshot of a computer

Description automatically generated

Seat Walkin

A screenshot of a computer

Description automatically generated

Seat Walkin Fail

A screenshot of a computer

Description automatically generated

## Walk In State Machine

A screenshot of a computer

Description automatically generated

Seat WalkIn

A computer screen shot of a computer screen

Description automatically generated

WalkIn Fail

A computer screen shot of a computer

Description automatically generated

# Final Class and Object Diagrams

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Pre and Post Conditions

### Booking System Record Arrival

use> !bookSys.selectBooking(2,20,0)

use> !bookSys.recordArrival(17,00)

[Error] 1 precondition in operation call `BookingSystem::recordArrival(self:bookSys, hr:17, mn:0)' does not hold:

A screenshot of a computer

Description automatically generated

### 

### Booking System Cancel Conditions

context BookingSystem::cancel(r : Reservation)

pre Pre1: current->includes(r)

post Post1: current->excludes(r)

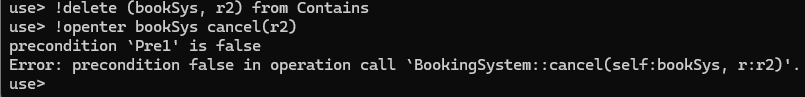
A screenshot of a computer

Description automatically generated

### Cancel Pre-Condition Success

A computer screen with white text

Description automatically generated



### Cancel post success

A black background with white text

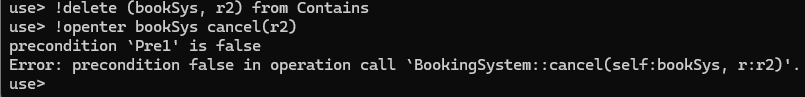
Description automatically generated

### Cancel pre success

A computer screen with white text

Description automatically generated

### Cancel pre fail



### Booking System Change Table

### overNoOfCovers()

pre-condition

A screen shot of a computer

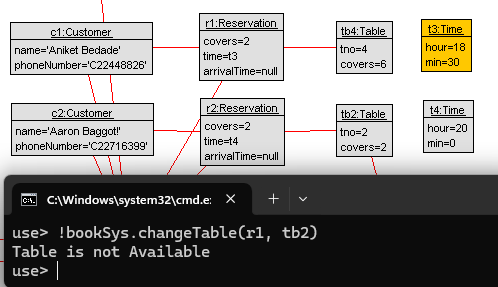
Description automatically generated

Object diagram before implementation of conditions

A screenshot of a computer

Description automatically generated

Post condition Fail – Table is not available



A screenshot of a computer

Description automatically generated

*Error: postcondition false in operation call `BookingSystem::changeTable(self:bookSys, r:r2, table:tb3)'.*

Table is not Available

[Error] 1 postcondition in operation call `BookingSystem::changeTable(self:bookSys, r:r2, table:tb3)' does not hold:

Post1: r.table->includes(table)

r : Reservation = r2

r.table : Table = tb4

r.table : Set(Table) = Set{tb4}

table : Table = tb3

r.table->includes(table) : Boolean = false

call stack at the time of evaluation:

1. BookingSystem::changeTable(self:bookSys, r:r2, table:tb3) [caller: bookSys.changeTable(r2, tb3)@<input>:1:0]

Post condition Success

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

### Make Reservation – Underage Condition

A computer screen shot of a black screen

Description automatically generated

Enhancement of Menu Ordering and Payment Systems

In our efforts to optimize the restaurant management system, we undertook a project to enhance both the menu ordering and payment systems. Below is a detailed overview of the objectives, methodologies, challenges, and insights gained from our endeavour.

Menu Ordering System Enhancement:

Our aim was to enrich the menu ordering system to offer customers a more personalized and flexible dining experience. The key objectives included:

Menu Customization: Introducing features that allow customers to customize their orders based on their preferences or dietary requirements, such as selecting specific ingredients or preparation methods.

Table Change Requests: Implementing functionality for customers to request changes to their assigned tables, facilitating smoother dining experiences, especially for larger groups or patrons with specific seating preferences.

To achieve these objectives, we developed detailed use case scenarios outlining the steps involved in placing customized orders and requesting table changes. Additionally, we defined preconditions, postconditions, and invariants to ensure the correctness and consistency of the menu ordering process.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

Payment System Adaptation:

Our goal here was to adapt the payment system to accommodate both card and cash transactions, providing customers with greater flexibility in settling their bills. The main objectives included:

Support for Multiple Payment Methods: Modifying the payment system to accept various payment methods, including credit/debit cards, mobile payments, and cash.

Use Case Scenarios: Crafting scenarios to illustrate different payment processing scenarios, such as customers choosing between card and cash payments, processing contactless payments, and managing cash transactions.

We also defined preconditions, postconditions, and invariants to ensure the security and accuracy of payment transactions, including verification of card authenticity and reconciliation of cash amounts.

Insights and Reflections:

Throughout the implementation process, we encountered challenges related to system complexity and impacts on cohesion and coupling. However, these challenges provided valuable insights into system architecture and design considerations. We learned the importance of balancing system flexibility with maintainability and the need for robust error handling mechanisms.

A computer screen shot of a black screen

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

# Restaurant Project Code

-- Aniket Bedade C22448826

-- Aaron Baggot C22716399

model BookingSys

-- main control class

class BookingSystem

operations

selectBooking(tno : Integer, hr : Integer, mn : Integer)

begin

for r in self.current do

if (r.table.tno = tno and r.getTime() = hr\*60+mn) then

insert(self,r) into Selected

end

end

end

unSelectBooking()

begin

delete (self,self.selected) from Selected

end

recordArrival(hr : Integer, mn : Integer)

begin

declare t: Time;

t := new Time;

t.setTime(hr, mn);

self.selected.setArrivalTime(t);

WriteLine('Customer has just been seated ');

self.selected.showDetails()

end

cancelReservation()

begin

Write('The booking made by ');

self.selected.getCustomerName();

WriteLine(' was cancelled');

destroy self.selected

end

changeTable(r : Reservation, table : Table)

begin

if (table.checkAvailability(r.time)) then

delete (r, r.table) from IsAt;

insert(r, table) into IsAt

else

WriteLine('Table is not Available')

end

end

-- version of cancel not defined in SOIL

cancel(r : Reservation)

statemachines

psm States

states

newBooking : initial

notSelected

selected

transitions

newBooking -> notSelected { create }

notSelected -> selected { selectBooking() }

selected -> selected { [self.selected.table.covers >= self.selected.covers] recordArrival() }

selected -> selected { selectBooking() }

selected -> notSelected { unSelectBooking() }

selected -> notSelected { cancelReservation() }

notSelected -> notSelected { cancel() }

end

end

class Booking

attributes

covers : Integer

time : Time

operations

setArrivalTime(t: Time)

begin

end

showDetails()

begin

end

getCustomerName()

begin

end

setCovers(cv : Integer)

begin

self.covers := cv

end

setTime(t : Time)

begin

self.time := t

end

-- A query operation to retun the time of the booking in minutes

getTime() : Integer = time.getTime()

end

class Receptionist

operations

greetWalkIn(covers : Integer, hr : Integer, min : Integer)

begin

declare w : WalkIn, t : Time;

t := new Time;

t.setTime(hr, min);

w := new WalkIn;

w.setCovers(covers);

w.setTime(t);

insert(self, w) into Talking;

WriteLine('Welcome! You will be seated in a moment.')

end

seatWalkIn()

begin

declare run : Boolean;

run := true;

for tb in self.table do

if (run and tb.checkAvailability(self.current.time) and self.current.covers <= tb.covers) then

insert(self.current, tb) into IsAt;

self.current.seated();

WriteLine('Please have a seat here');

delete(self, self.current) from Talking;

run := false

end

end;

if (run) then

WriteLine('Im sorry, there are no tables available.');

self.current.notSeated();

destroy self.current

end

end

end

class WalkIn < Booking

-- dummy operations to change states in statemachine

operations

seated()

begin

end

notSeated()

begin

end

statemachines

psm States

states

newWalkIn : initial

waiting

seated

left

transitions

newWalkIn -> waiting { create }

waiting -> seated { seated() }

waiting -> left { notSeated() }

end

end

class Reservation < Booking

attributes

arrivalTime : Time

operations

showDetails()

begin

Write('Selected reservation is by ');

WriteLine(self.customer.name)

end

setArrivalTime(t : Time)

begin

self.arrivalTime := t

end

setCustomer(c : Customer)

begin

insert(c, self) into Makes

end

getCustomerName()

begin

Write(self.customer.name)

end

statemachines

psm States

states

newReservation : initial

waiting

seated

transitions

newReservation -> waiting { create }

waiting -> seated { setArrivalTime() }

end

end

class Customer

attributes

name : String

age : Integer

phoneNumber : String

operations

makeReservation(covers : Integer, table : Table, bookingHr : Integer, bookingMin : Integer)

begin

declare time : Time, r : Reservation;

time := new Time;

time.setTime(bookingHr, bookingMin);

if (table.checkAvailability(time)) then

r := new Reservation;

r.setCovers(covers);

r.setTime(time);

r.setCustomer(self);

-- add the new reservation to the BookingSystem

insert(self.bs, r) into Contains;

insert(r, table) into IsAt;

WriteLine('Reservation made successfully.')

else

-- delete the new Time object

WriteLine('Table is not available.');

destroy time

end

end

end

class Table

attributes

tno : Integer

covers : Integer

operations

-- query operation to check if table is free

checkAvailability(t : Time) : Boolean = self.bookings->size() = 0 or self.bookings->forAll(b | b.getTime() + 120 <= t.getTime() or t.getTime() + 120 <= b.getTime())

end

class Time

attributes

hour : Integer

min : Integer

operations

setTime(h: Integer, m: Integer)

begin

self.hour := h;

self.min := m

end

-- a query method to return the time since midnight in minutes

getTime() : Integer = self.hour\*60 + self.min

end

association IsAt between

Booking[\*] role bookings

Table[1]

end

association Makes between

Customer[1]

Reservation[\*] role reservations

end

association Talking between

Receptionist[1]

WalkIn[1] role current

end

association Checks between

Receptionist[1]

Table[\*]

end

association Contains between

BookingSystem[1] role bs

Booking[\*] role current

end

association Selected between

BookingSystem[1] role bsys

Booking[0..1] role selected

end

association Uses between

BookingSystem[1] role bs

Customer[1]

end

constraints

context BookingSystem::recordArrival(hr: Integer, mn: Integer)

pre arrivalBeforeBooking: self.selected.getTime() <= mn + hr\*60

context BookingSystem::cancel(r : Reservation)

pre Pre1: current->includes(r)

post Post1: current->excludes(r)

context BookingSystem::changeTable(r : Reservation, table : Table)

pre overNoOfCovers: r.covers <= table.covers

pre Pre1: r.table->excludes(table)

post Post1: r.table->includes(table)

context Customer::makeReservation(covers : Integer, table : Table, bookingHr : Integer, bookingMin : Integer)

pre underageBookingTime: self.age < 18 and bookingHr \* 60 + bookingMin < 21 \* 60

context Table

inv noOverlap: self.bookings->forAll(b1, b2 | b1.getTime() + 120 <= b2.getTime() or b2.getTime() + 120 <= b1.getTime())

context BookingSystem

inv mustBeOneOf: not self.selected.isDefined() or self.current->includes(self.selected)

# Conclusion

Overall, the project to extend and refine the restaurant management software was a rewarding and enriching experience. It provided a platform to apply theoretical knowledge to real-world problems, collaborate with peers, hone technical and interpersonal skills, and foster personal and professional growth. By embracing challenges, learning from experiences, and reflecting on lessons learned, I have developed a deeper appreciation for the complexities and possibilities inherent in the field of software engineering.

Overall, our efforts to enhance the menu ordering and payment systems exemplify our proactive approach to system improvement and our commitment to delivering an exceptional dining experience for our customers.