C22716399 Aaron Baggot

[Company name] | [Company address]

[Document title]

[Document subtitle]

Table of Contents

[Introduction 3](#_Toc163988283)

[Restaurant Selection and Use Case Scenarios 4](#_Toc163988284)

[Class Diagram with soil implementation 5](#_Toc163988285)

[Class Diagram 6](#_Toc163988286)

[Object Diagram 7](#_Toc163988287)

[Sequence Diagram 8](#_Toc163988288)

[BookingSystem 9](#_Toc163988289)

[Booking and Reservation Class 10](#_Toc163988290)

[Implementation 10](#_Toc163988291)

[Booking System SelectBooking() 11](#_Toc163988292)

[Booking System unSelectBooking() 12](#_Toc163988293)

[Booking System changeTable() 13](#_Toc163988294)

[BookingSystem makeReservation() 15](#_Toc163988295)

[BookingSystem cancelReservation() 17](#_Toc163988296)

[Booking System recordArrival() 19](#_Toc163988297)

[State Machine for Reservation 20](#_Toc163988298)

[Arrival before the booking 21](#_Toc163988299)

[USE Model Overview 22](#_Toc163988300)

[Testing and Validation 22](#_Toc163988301)

[Booking System 22](#_Toc163988302)

[State Machines 23](#_Toc163988303)

[Booking System state machine 23](#_Toc163988304)

[selectBooking() 24](#_Toc163988305)

[unselectBooking() 25](#_Toc163988306)

[cancelBooking() 25](#_Toc163988307)

[Booking System Record arrival 26](#_Toc163988308)

[Receptionist Operation 26](#_Toc163988309)

[Walk In State Machine 34](#_Toc163988310)

[Final Class and Object Diagrams 36](#_Toc163988311)

[Pre and Post Conditions 37](#_Toc163988312)

[Booking System Record Arrival 37](#_Toc163988313)

[Booking System Cancel Conditions 38](#_Toc163988314)

[Cancel Pre-Condition Success 39](#_Toc163988315)

[Cancel post success 40](#_Toc163988316)

[Cancel pre success 40](#_Toc163988317)

[Cancel pre fail 40](#_Toc163988318)

[Booking System Change Table 41](#_Toc163988319)

[overNoOfCovers() 41](#_Toc163988320)

[Make Reservation – Underage Condition 44](#_Toc163988321)

[Restaurant Project Code 54](#_Toc163988322)

[Conclusion 63](#_Toc163988323)

# 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

The 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

In the implemented system for a booking management system, the functionality for handling walk-in customers has been added through the creation of a receptionist class. This addition allows for seamless management of both reservations and walk-ins, ensuring efficient seating and service for all patrons. The two key operations involved in this aspect of the system are greetWalkIn() and seatWalkIn().

The greetWalkIn() operation is triggered when a walk-in customer arrives at the establishment. It instantiates a new WalkIn object, capturing details such as the number of people in the party (covers) and the current time. Subsequently, the walk-in is added to the list of customers currently being attended to, and a warm welcome message is extended.

Following the initial greeting, the seatWalkIn() operation endeavours to find an available table for the walk-in customer. This process involves iterating over each table in the receptionist's list of tables. If a suitable table is found, meeting both availability criteria at the current time and having sufficient capacity to accommodate the walk-in party, the walk-in is seated at that table. This is achieved by associating the walk-in with the selected table in the system's data model. Additionally, the walk-in's state is updated to reflect its seating status, and a message prompts the customer to take a seat. In cases where no suitable table is available, an apology message is issued, and the walk-in's status is appropriately adjusted to reflect being unseated.

Receptionist Greet WalkIn

Receptionist greets the customer table for 4, time 7pm

A screenshot of a computer

Description automatically generated

Receptionist Seat WalkIn

Table 4 is allocated to the customer

A screenshot of a computer

Description automatically generated

Receptionist seatWalkIn Fail

In this scenario, the receptionist welcomes a walk-in customer who wishes to book a table for three people at 7 p.m. However, upon checking the availability, it is found that all tables are already booked. As a result, the system returns a statement indicating that no tables are currently available for seating.

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

When a walk-in arrives at the establishment, it enters a state machine that tracks its progression through various states based on the actions taken by the receptionist. Two key states within this state machine are "seated" and "not seated/left when none available."

In the "seated" state, a walk-in has successfully been assigned to a table and is currently enjoying their dining experience. This state signifies that the walk-in has been accommodated and is actively engaged within the establishment.

Conversely, the "not seated/left when none available" state captures the scenario where no tables are available to accommodate the walk-in. In this state, the walk-in has not been seated and may choose to either wait for a table to become available or leave the establishment. This state reflects the system's acknowledgment of the walk-in's presence and its inability to immediately accommodate them due to resource constraints.

These state machines provide a structured approach to managing the flow of walk-in customers within the booking system, ensuring efficient handling of their requests and maintaining a positive customer experience even in cases where immediate seating is not feasible.

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

In the depicted sequence diagram, the interaction begins with a request to the booking system (bookSys) to select a booking for a table number 2 at 8:20 PM. This action is initiated by an external entity, denoted by the use> notation, indicating a request made by a user or another system component. Following this, the booking system attempts to process the request by calling the selectBooking() operation with the specified parameters. However, an error occurs during the subsequent step, where the booking system is unable to record the arrival of a customer at 5:00 PM (17:00) due to a precondition violation. The error message indicates that a precondition required for the recordArrival() operation is not met, preventing its successful execution. This sequence highlights the importance of ensuring that all preconditions are satisfied before invoking operations within the system to maintain its integrity and functionality.

A screenshot of a computer

Description automatically generated

### Booking System Cancel Conditions

In the system's implementation, a state machine governs the cancellation of reservations, ensuring proper handling of booking cancellations. The cancellation process involves transitioning the reservation through different states to reflect its updated status within the system.

The cancel operation is subject to preconditions and postconditions to maintain consistency and integrity. The precondition, denoted as Pre1, asserts that the reservation to be cancelled (r) must currently exist within the system's list of current reservations (current). This ensures that only existing reservations can be cancelled, preventing erroneous or unauthorized cancellations.

Following the cancellation operation, the system's state is modified according to the postcondition, labelled as Post1. This condition specifies that after the cancellation, the reservation r should no longer be included in the list of current reservations (current). This postcondition guarantees that the cancelled reservation is effectively removed from the system, preventing any further actions or interactions associated with it.

By enforcing these preconditions and postconditions, the system maintains consistency and reliability in managing reservation cancellations, ensuring that only valid cancellations are processed and that the system's state accurately reflects the outcome of the cancellation operation.

context BookingSystem::cancel(r : Reservation)

pre Pre1: current->includes(r)

post Post1: current->excludes(r)

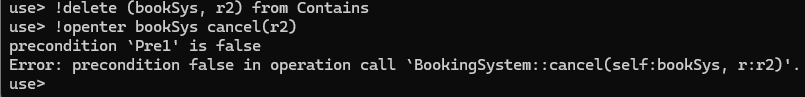
A screenshot of a computer

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

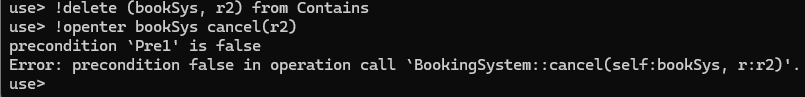
Description automatically generated

### Cancel pre success

A computer screen with white text

Description automatically generated

### Cancel pre fail



### Booking System Change Table

In the system's logic, the condition overNoOfCovers is a crucial aspect governing table changes within the booking system. This condition ensures that when a reservation requests a change in tables, the new table must have a sufficient capacity to accommodate the number of guests specified in the reservation.

The scenario provided illustrates an attempt to change a reservation (r1) to a new table (th2). However, the operation fails due to the violation of the overNoOfCovers condition. The error message indicates that the requested table is not available, preventing the reservation from being moved to the new table. Subsequently, the system reports that the postcondition for the changeTable operation is false, indicating that the reservation's table does not include the new table after the attempted change.

### overNoOfCovers()

pre-condition

A screen shot of a computer

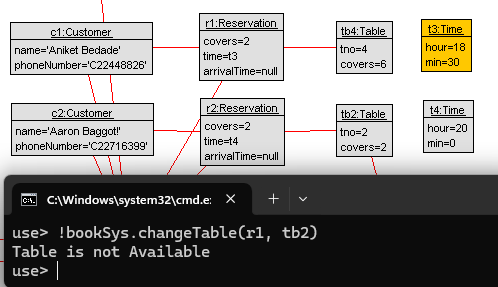
Description automatically generated

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

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

The provided code constitutes a model for a booking system tailored to manage reservations and walk-in customers within an establishment. Developed by Aaron Baggot and Aniket Bedade, this model encapsulates various classes and operations aimed at orchestrating the booking process, handling reservations, and seating customers.

The main control class, BookingSystem, governs core operations crucial for managing bookings. These include selecting a booking, recording customer arrivals, cancelling reservations, and facilitating table changes. Additionally, the BookingSystem class integrates a state machine to guide the flow of operations, ensuring a systematic progression from booking selection to finalization or cancellation.

To complement the system's functionality, auxiliary classes such as Receptionist, WalkIn, Reservation, Customer, Table, and Time have been implemented. These classes facilitate interactions between system components, manage customer data, and provide essential functionalities such as checking table availability and setting booking times.

Furthermore, the model incorporates constraints and associations to enforce business rules and maintain data integrity. Constraints such as underage booking restrictions and table booking overlap prevention enhance the system's robustness and reliability.

Overall, this model serves as a foundational framework for developing a comprehensive booking management system, offering a structured approach to handle reservations, walk-ins, and table assignments within a hospitality setting.

**-- 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

Embarking on the journey to extend and refine the restaurant management software has been an invaluable learning experience in the realm of software engineering. Through this project, I've gained hands-on experience in applying theoretical concepts to real-world scenarios, collaborating effectively with peers, and refining both technical and interpersonal skills. Each challenge encountered along the way served as an opportunity for growth, fostering a deeper understanding of the intricacies and potential of software engineering.

Reflecting on the project, I've come to appreciate the importance of proactive problem-solving and continuous improvement in software development. Our efforts to enhance the menu ordering and payment systems exemplify our dedication to delivering an exceptional dining experience for our customers. Moving forward, I'm eager to apply the knowledge and insights gained from this project to future endeavours, continually striving to innovate and elevate the quality of software solutions.

In conclusion, this project has not only deepened my understanding of software engineering principles but also instilled a sense of confidence and capability in tackling complex challenges.