Projet de Gestion de Restaurant

Groupe 7 X3 - UCAC-ICAM

December 10, 2024

Contents

[1 Introduction 2](#_Toc4783)

[2 Architecture Globale 2](#_Toc4784)

[2.1 Model (Modèle) 2](#_Toc4785)

[2.2 View (Vue) 2](#_Toc4786)

[2.3 Controller (Contrôleur) 2](#_Toc4787)

[3 Design Patterns Utilisés 2](#_Toc4788)

[3.1 MVC (Model-View-Controller) 2](#_Toc4789)

[3.2 Observer 2](#_Toc4790)

[3.3 Singleton 3](#_Toc4791)

[3.4 State 3](#_Toc4792)

[3.5 Factory Method 3](#_Toc4793)

[4 Programmation Système 3](#_Toc4794)

[4.1 Concurrence avec Threads 3](#_Toc4795)

[4.2 Communication Inter-Processus (IPC) 4](#_Toc4796)

[4.3 Synchronisation 4](#_Toc4797)

[5 Conclusion 4](#_Toc4798)

# Introduction

Ce document décrit l’architecture logicielle et les principes de programmation adoptés pour le projet de gestion de restaurant. Il combine une approche basée sur les design patterns et le paradigme de la programmation système en C++ avec Qt.

# Architecture Globale

Le projet est structuré selon une architecture MVC (Model-View-Controller), enrichie de concepts de programmation système. Cette architecture permet de séparer la logique métier, la gestion des données, et l’interface utilisateur.

## Model (Modèle)

Le modèle comprend les classes représentant les entités principales :

* Commande : Gestion des commandes des clients.
* Stock : Gestion centralisée des ressources (ingrédients, outils).
* Employés : Acteurs du système (serveurs, chefs, maître d’hôtel).

## View (Vue)

L’interface utilisateur est implémentée à l’aide de Qt, avec des éléments comme QGraphicsScene pour visualiser la salle de restaurant, QTableWidget pour afficher les commandes ainsi que QPushButton pour les boutons.

## Controller (Contrôleur)

Le contrôleur gère les interactions entre la vue et le modèle, notamment via les mécanismes de signaux et slots fournis par Qt tels que QPushButton ou Q\_OBJECT.

# Design Patterns Utilisés

## MVC (Model-View-Controller)

L’architecture MVC assure une séparation claire des responsabilités et fonctionne de la manière décrite précédemment.

## Observer

## Le mécanisme de signaux et slots de Qt est une implémentation directe du pattern Observer :

## Sujet (Signal) : Un bouton, un timer, ou un modèle de données émet un signal lorsqu'une action ou un événement se produit.

## Observateurs (Slots) : Les slots connectés réagissent au signal.

Les modèles Qt comme QSqlTableModel implémentent également le pattern Observer. Lorsqu'un modèle change, tous les observateurs connectés (comme un QTableView) sont automatiquement notifiés et se mettent à jour.

## Singleton

Implémentation du Singleton pour la connexion MySQL

Dans ce projet, la connexion à la base de données peut être gérée par une classe Singleton pour éviter d'ouvrir plusieurs connexions inutiles.

## State

Le design pattern State est particulièrement utile pour gérer les comportements dynamiques d’un objet en fonction de son état courant. Dans ce projet, il peut être utilisé pour gérer :

* L’état des tables dans la salle de restauration.
* Les états des commandes en cuisine.
* L’état global de la simulation (en cours, en pause, arrêtée).

Le pattern State permet de :

* Encapsuler les différents comportements dans des classes distinctes représentant les états.
* Changer dynamiquement le comportement d’un objet en changeant son état interne.
* Simplifier le code en évitant les longues chaînes de conditions.

## Factory Method

Dans ce projet, le design pattern Factory Method peut être utilisé pour créer dynamiquement des objets en fonction de certains paramètres ou types, sans exposer directement les classes concrètes au code appelant. Ce pattern permet de déléguer la responsabilité de l'instanciation des objets à des sous-classes ou à des méthodes spécialisées.

# Programmation Système

## Concurrence avec Threads

L'utilisation de threads dans ce projet permet de gérer efficacement des tâches parallèles, telles que :

* Gestion des commandes en cuisine.
* Supervision des tables dans la salle de restauration.
* Analyse continue des seuils critiques pour les ressources.
* Qt fournit des classes comme QThread pour implémenter et gérer des threads. Voici comment les threads peuvent être intégrés et implémentés dans ce projet.

## Communication Inter-Processus (IPC)

La salle et la cuisine communiquent via des sockets ou des pipes nommées :

Dans ce projet, l'Inter-Process Communication (IPC) peut être utilisé pour permettre aux différentes parties de l'application (comme la cuisine, la salle de restauration, et le tableau de bord) de communiquer entre elles efficacement, surtout si elles fonctionnent en tant que processus distincts.

Qt propose plusieurs options pour l'IPC, en fonction des besoins et de la complexité :

Sockets (QTcpSocket, QUdpSocket) Pour la communication réseau locale ou distante.

Pipes nommés (QProcess) Pour une communication rapide entre processus locaux.

Shared Memory (QSharedMemory) Partage de données volumineuses (ex. état global) entre processus.

Message Queues (QDataStream) Envoi de messages séquentiels entre modules.

## Synchronisation

1. Ressources partagées dans ce projet

a) Base de données MySQL

Partage :

Plusieurs threads ou processus (cuisine, salle, tableau de bord) peuvent lire ou écrire dans les tables SQL simultanément.

Problèmes possibles :

Incohérences si deux threads essaient de modifier les mêmes données en même temps (par exemple, mise à jour des quantités ou des états des tables).

b) État des tables

Partage :

Les tables de la salle de restauration ont des états partagés (Disponible, Occupé, Réservé).

Problèmes possibles :

Conflits si plusieurs threads tentent de modifier l'état d'une même table simultanément.

c) Ressources (Ingrédients)

Partage :

Les ingrédients dans la cuisine ou le tableau de bord sont suivis pour détecter les seuils critiques.

Problèmes possibles :

Des conflits peuvent survenir si des threads indépendants lisent ou modifient les quantités en même temps.

d) Mémoire partagée (QSharedMemory)

Partage :

L'état global des tables ou des ressources peut être stocké dans la mémoire partagée pour un accès rapide entre processus.

Problèmes possibles :

Accès concurrent peut corrompre les données si des mécanismes de verrouillage ne sont pas utilisés.

e) Fichiers de logs

Partage :

Plusieurs threads ou processus peuvent vouloir écrire dans le même fichier de log (application.log).

Problèmes possibles :

Données de logs mélangées ou écrasées en cas d'accès concurrent.

2. Mécanismes de synchronisation pour protéger les ressources

a) Verrouillage avec QMutex (Threads locaux)

Description :

QMutex est un mécanisme de verrouillage fourni par Qt qui empêche plusieurs threads d'accéder simultanément à une ressource partagée.

# Conclusion

Cette architecture robuste intègre des design patterns éprouvés et des concepts de programmation système pour garantir performance et modularité. Qt et C++ permettent une implémentation efficace, adaptée aux besoins de simulation en temps réel.