Kitchen Of Secrets

COS 214 Project

Table of Contents

Overview

In this assignment, we're tasked with designing a restaurant simulator, a dynamic blend of activities in the customer floor and the bustling kitchen . On the customer floor, our role is to manage seating, orders, and ensure customer satisfaction. This includes orchestrating the seating process, taking orders, and accommodating customer expectations. Meanwhile, in the kitchen, the chaos of food preparation unfolds. Different chefs handle various aspects, from cooking to plating. Additional features like inventory and accounting can be incorporated to enhance the experience.

Technologies

Technologies

- C++
- Adobe
- JavaScript
- HTML5
- CSS3

Team

Ashley Kapaso	Ayanda Juqu	Chenoa Perkett	Eugene Mpande	Joshua Wereley	Thato Kalagobe

Research Summary

The restaurant simulator project you're working on involves designing and implementing a system that mimics the operations of a real-world restaurant. This includes managing various processes happening simultaneously, such as seating customers, taking orders, preparing food, and more. The project primarily focuses on two areas: the floor and the kitchen.

The Floor

The **floor** is where customers are seated and managed. A system is needed for customers to request seating, place orders, and pay their bills. Waiters are assigned tables to take responsibility for. The restaurant could have a booking system or walk-in service. Customer expectations need to be managed effectively to ensure satisfaction.

The Kitchen

The **kitchen** is where the food is prepared. It's a chaotic ensemble of various processes happening at the same time. The cooking of food at various workstations and being passed between chefs is a significant part of these processes.

To better understand the operations in a professional kitchen, let's look at the Brigade de Cuisine or kitchen hierarchy[1]. This French brigade system is adopted in most modern professional kitchens to ensure smooth operations[1]. The size and structure of the Brigade de Cuisine vary depending on the size and style of the restaurant[1].

Here are some key positions in the kitchen hierarchy:

- **Executive Chef**: Sits at the top of the kitchen hierarchy; their role is primarily managerial[1].
- **Chef de Cuisine (Head Chef)**: Focuses on managerial duties relating to the whole kitchen[1].
- **Sous Chef (Deputy Chef)**: Shares a lot of the same responsibilities as the head chef but is much more involved in the day-to-day operations in the kitchen[1].
- **Chef de Partie (Station Chef)**: This role is split into many different roles. There is more than one chef de partie, and each one is responsible for a different section of the kitchen[1].

The Chef de Partie roles include:

- **Sauté Chef/Saucier (Sauce chef)**: Responsible for sautéing foods and creating sauces and gravies[1].
- **Boucher (Butcher Chef)**: Prepares meat and poultry[1].
- **Poissonnier (Fish Chef)**: Prepares fish and seafood[1].
- **Rotisseur (Roast Chef)**: Responsible for roast meats and appropriate sauces[1].
- **Friturier (Fry Chef)**: Prepares fried food items[1].
- **Grillardin (Grill Chef)**: Specializes in grilled foods[1].

Each position holds an important role in the overall function of the kitchen[1]. Understanding this hierarchy can help you design your restaurant simulator more effectively.

Sources: 1. Kitchen Hierarchy Explained | The Brigade de Cuisine - High Speed Training. 2. The Kitchen Hierarchy Explained | Eight positions & roles.

3. Food Hygiene Blog | The Hub | High Speed Training. 4. What is the kitchen brigade system? - Le Cordon Bleu.

Git Standards

This document outlines the Git standards that we follow in this project.

Branching

- We use the Git Flow branching model.
- Our main branch is `main`.
- We create feature branches for each subsystem.
- We merge feature branches into `main` when they are complete and tested.
- We create release branches for each new release.

Committing

- Commit messages should be clear and concise, describing the changes made.
- Commit messages should be wrapped to 72 characters or less.
- Commit messages should not include references to pull requests or other issues.

Documentation

- Documentation should be written in Markdown and, in addition, use Doxygen.
- Documentation should be clear and concise, explaining how to use the code.
- Documentation should be kept up-to-date with the code.

Pull Requests

- Pull requests should be created for all merges to `main` and release.
- Pull requests should be reviewed by at least 2 other developers before they are merged.
- Pull requests should be merged into `main` when they are complete and approved.

Testing

- All new code should be unit tested.
- Integration tests should be written to verify that different parts of the code work together correctly.
- End-to-end tests should be written to verify the functionality of the entire system.

Deployment

- We use a continuous integration and continuous delivery (CI/CD) pipeline to deploy our code to production.
- Our CI/CD pipeline includes unit tests, integration tests, and end-to-end tests.

- Our CI/CD pipeline automatically deploys our code to production when the tests pass.

C++ Coding Standard

Introduction

In programming, consistency is key. Adhering to a set of coding standards can significantly improve the readability, maintainability, and quality of your code. These guidelines are for writing C++ code that is clean, understandable, and easy to follow. They may need to be adapted based on your specific project or organizational requirements.

Standards

1. Naming Conventions

- Use meaningful and self-explanatory names for variables, functions, and classes.
- Follow a consistent naming pattern like camelCase.
- Constants should be all uppercase with underscores.

```
int employeeCount; // Good
int e; // Bad

void calculateSalary(); // Good
void calc_sal(); // Bad

class Employee { // Good
  ...
};
class emp { // Bad
  ...
};

const int MAX_EMPLOYEES = 100; // Good
const int max = 100; // Bad
```

2. Comments

- Write comments for complex code blocks using Doxygen.
- Avoid unnecessary comments that do not add value.
- Keep comments up to date with code changes.

```
/**
 * @brief Calculates the salary of an employee based on hours worked and hourly rate.
 * @param hours The number of hours worked by the employee.
 * @param rate The hourly rate of the employee.
```

```
 * @return The salary of the employee.
 */
double calculateSalary(int hours, double rate); // Good

// This function calculates the salary of an employee // Bad (redundant)
double calculateSalary(int hours, double rate);

// TODO: Fix this bug // Good (actionable)
int x = y + z;

// This is a variable // Bad (obvious)
int x;
```

3. Formatting

- Use consistent indentation (spaces or tabs) throughout the code.
- Keep line length reasonable, e.g., 80-100 characters.
- Use spaces around operators and after commas for readability.

```
if (condition) { // Good (consistent indentation)
  statement1;
  statement2;
}

if(condition){ // Bad (inconsistent indentation)
statement1;
statement2;
}

double result = calculateSalary(hours, rate); // Good (reasonable line length)

double result;
result = calculateSalary(hours, rate) + calculateBonus(hours, rate) - calculateTax(hours, ra
// Bad (too long line length)

int sum = a + b; // Good (spaces around operator)
int sum=a+b; // Bad (no spaces around operator)

void foo(int x, int y); // Good (space after comma)
void foo(int x,int y); // Bad (no space after comma)
```

4. Error Handling

- Always check return values and handle errors appropriately.

```
if (ptr == nullptr) { // Good (check for null pointer)
  // Handle error...
```

```
}

if (fopen_s(&file, "data.txt", "r") != 0) { // Good (check for file open failure)
  // Handle error...
}

ptr->doSomething(); // Bad (assume pointer is not null)

file = fopen("data.txt", "r"); // Bad (assume file open succeeds)
```

5. Object-Oriented Programming

- Encapsulate data by using classes.
- Use inheritance and polymorphism to reuse and extend code.

```
class Employee { // Good (encapsulate data)
 private:
 int salary;
 public:
 void setSalary(int s);
 int getSalary();
};

class Manager : public Employee { // Good (use inheritance)
 private:
 int bonus;
 public:
 void setBonus(int b);
 int getBonus();
};

Employee* emp = new Manager(); // Good (use polymorphism)
emp->setSalary(1000);
emp->setBonus(500);

int salary = emp->salary; // Bad (access private data directly)
int bonus = emp->bonus; // Bad (access private data directly)
```

6. Memory Management

- Always initialize variables.
- Use smart pointers to manage memory.

```
int *ptr = nullptr; // Good practice

std::unique_ptr<int> ptr(new int(10)); // Good (use smart pointer)
```

```
int *ptr = new int(10); // Bad (use raw pointer)

delete ptr; // Bad (forget to delete pointer)
```

7. Best Practices

- Keep functions and classes small and focused on a single task.
- Use the const keyword wherever possible.

```
void printEmployeeInfo(const Employee& e); // Good (use const reference)

void printEmployeeInfo(Employee e); // Bad (use copy)

class Employee {
 public:
 void setSalary(int s);
 int getSalary() const; // Good (use const member function)
 ...
};

class Employee {
 public:
 void setSalary(int s);
 int getSalary(); // Bad (no const member function)
 ...
};
```

Conclusion

These guidelines are designed to serve as a starting point and should be adapted to suit your specific project requirements or personal coding style. The most important thing is consistency. A consistent coding style can make a significant difference in the readability and maintainability of your code.