COS214 Practical Project Report

Code is an art where each mistake can be costly.

Zethembe Danise, 20704209
Lerato Kgomoeswana, 22542354
Lwando Msindo, 21507067
Nokukhanya Ndlovu, 22768352
Khwezi Ntsaluba, 22515012
Future Phahlamohlaka, 22524798
Nthati Rampa, 20475102

Research

A project of this scale could have been approached by something other than jumping head-first into writing code. There were processes followed preceding the implementation of the project. These were planning, research, and design. A separate document has been compiled containing the design of the project. The research about the project involved includes investigating coding and Git standards as well as the inner workings of a restaurant and how it operates. A few key points stood out while conducting the research:

With regards to the coding standards, the idea behind them is that completing a project is a team effort that should be coordinated lest it devolves into a discordant endeavor that results in a poorly implemented system. Coordination and planning ensure that the code produced is consistent, simple to understand, maintainable, and reusable. The consequences incurred by inefficient and complicated code can be costly since the intention is to complete a project in good time (and within budget).

Tools are needed to streamline the completion of a project.

Enter GitHub.

A GitHub repository consists of a single central branch — often called the main branch. To the uninitiated user, committing code to the main branch is a clever idea. This could be correct. Features branches created for the development of different sections of a project. Code is pushed to said feature branches. A mock main branch (which could be named the development branch) can serve the purpose of merging the code and compiling the feature branches. This ensures that any conflicts and inconsistencies can be managed without inadvertently introducing mixtures of old and new code, introducing bugs into the code, and ruining the project flow. What is described above is the purpose of Gitflow. Furthermore, developers should comment on the code and offer their input on what can be improved or changed. These are called commit reviews.

Once all parties concerned have completed bug fixes, adding features, and making improvements, the completed project is merged into the main branch. At this point, no commits or changes should be made unless necessary.

Reasoning for design decisions

In a project with multiple moving parts, such as the one being discussed, it is left to the developers to decide how they will achieve the outcomes of the system being developed. Numerous decisions were made with careful consideration of their impact on various aspects of the project, from the integration of the various sections of the code to the execution of the code itself. This ties in closely with the assumptions made about the project – this will be discussed later in the document.

Time constraints introduced the possibility of the project not being completed in time, and as such, a sequential implementation was chosen over a multithreaded implementation.

Firstly, the basic requirements (that is, the system requirements) were considered when designing the system.

How would a simple dining establishment be structured in code?

That is where the process of building the system began. The idea would be to keep the creation of the simulation simple. As such, patterns that would fit the criteria of a simple, efficient system that achieves what it has been created for were chosen – the patterns chosen will be discussed later in this document.
Each subsystem was developed independently with the overarching system in mind. The idea was to create loosely coupled code that can function even if one part of the system fails for reasons varying from edge cases that have been omitted to the system's inability to handle incorrect input.

How the patterns have been used and which pattern solved which problem

The practical project required that at least ten design patterns should be implemented in the simulation. This is to display our accumulated understanding and proficiency in designing a system as well as putting our programming skills to the test. As such, the following patterns were considered when developing the simulation:
Singleton: Based on the assumption that there is only one maître d' hotel in a restaurant, it was decided that the singleton pattern would best fit the implementation of the maître d' hotel. The maître d' hotel seats the customers and serves a superior to the waiters. The maître d' hotel also oversees complaints by delegating or communicating them to the appropriate staff.
Chain of Responsibility: Numerous responsibilities come with the running of a restaurant, such as taking orders, preparing meals, valet services, and bartending – to name a few. Every staff member is tasked with resolving issues within their job description. If this is not possible, the messages can be passed on to the relevant staff member, who would ensure that the problem is resolved in an adequate and timely manner.
State: The customers have moods based on how their dining experience is going. Meals start uncooked and, in the process of being prepared, transition to be cooked and served to the customers. Additionally, the customers may wish to settle in before ordering, or they may wish to get into the business of eating. These events depend on the state, and it was decided that this would be a suitable pattern for modeling these events.
Façade: The heart of the project is the simulation and seeing it in action. Instead of having the user interact with the system's intricacies, the operations have been obscured by a simple interface that will produce the desired outcomes from the simulation. Instead of having to acquaint themselves with how the system works, the users can enjoy seeing the system in action by clicking a few buttons and following simple input instructions.
Mediator: In a busy (and sometimes chaotic) environment such as a restaurant, messages must be communicated (between the staff members of the dining establishment) to ensure that the restaurant runs smoothly. As such, a mediator oversees the communication between the various simulation classes. The mediator also allows for the introduction of new employees into the restaurant without disturbing the communication channels that already exist while ensuring that messages get to the people they are intended.
Iterator: In a real-life restaurant, waiters walk up to the tables that they are in charge of and serve the patrons at that table. In an environment simulating a restaurant, moving between tables is more challenging than in real life. As such, a means of traversing the tables is needed. Enter the iterator. Regardless of the data structure used to represent the restaurant, an iterator ensures that the various parts of the restaurant will be traversed as required by the user while ensuring that the inner workings of the traversed structures are not exposed.
Composite: At the end of every meal comes the bill. Depending on the diners, one or more customers may be responsible for paying the bill. The decorator was considered for creating a bill, but the pattern was used for modeling the food items (more on that later). If every patron is responsible for paying his/her part of the bill, then separate slips are produced. Otherwise, if multiple combined bills are produced, each will be composed of the patrons whose meal is being paid.

Template Method: The majority of the items on the menu are prepared in similar ways, save for a few final additions to be added by a specialty chef. How does this align with the pattern? One could think of cooking the food as an "algorithm" where specific steps are the same, and the variation comes from who is preparing the meal and what type of delicacy is being prepared. Additionally, the simulation has a head chef whose purpose is to garnish the meal and ensure the quality of the food prepared. This is similar to what the other chefs would do, except with the expertise specific to the head chef – or the extra steps, if one may call them that.
Decorator: A decorator extends the functionality of an object. One cannot precisely extend the functionality of food items. How else can one extend something beyond its sole purpose? On a serious note, the decorator was used as a wrapper. Customers could order meals in various forms. That is, the meal could be plain, or they could have extras served alongside or as part of the meal.

Assumptions made

The practical project that we were tasked with completing was open-ended. It was left to our interpretation as to what details should or should not be included in the creation of the restaurant simulation. As such, the following assumptions were made while implementing the simulation (in no particular order):
It was assumed that a restaurant has one maître d' hotel based on a dictionary definition of what a maître d' hotel is.
The restaurant would work solely based on walk-ins. The addition of a booking system would incur additional overhead and require the use of multithreading in the implementation. However, multithreading was not a requirement in the simulation, and time constraints forced the implementation of a sequential program.
Customers can combine tables to seat larger customers up to the maximum capacity of the restaurant. Then onwards, customers would be seated in a waiting area where they would remain until a table becomes available. This was implemented merely to mimic the goings-on of a classy establishment. Furthermore, any additional customers who arrive while the restaurant is at maximum capacity (waiting areas included) would not be included in the simulation.
Since this is a simulation, the customers' moods would be randomized. This, in turn, results in the randomization of complaints or compliments. The severity and type of complaints were chosen to be those typically experienced by diners in a restaurant.
The chefs would be preoccupied and, thus, unable to check in on the customers to gauge their mood and enquire about their dining experience.
A basic version of a build-your-own meal restaurant would be sufficient to display the design patterns at work. Implementing exquisite meals and the choices thereof would not achieve the purpose of this project.

UML diagrams to support arguments

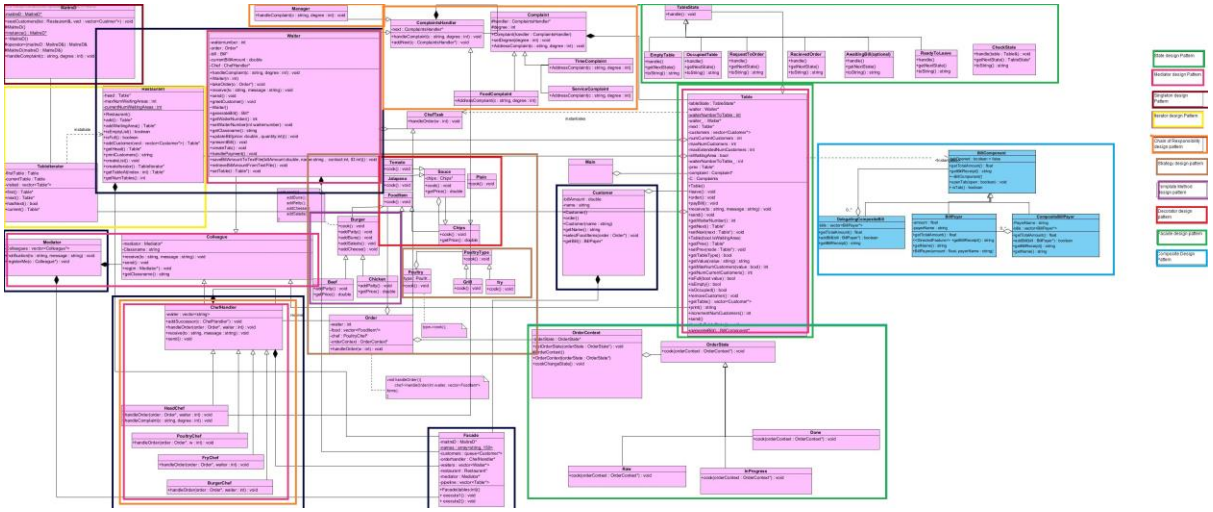

FIGURE 1: UML DIAGRAM CONSISTING OF ALL THE PATTERNS USED AND CLASSES IMPLEMENTED (PLEASE SEE SYSTEM REQUIREMENTS DOCUMENT FOR TASK TWO)

References

linearb.io. (n.d.). *What is Code Review & Why It is Important for Code Quality | Dev Interrupted Powered by LinearB*. [online] Available at: https://linearb.io/blog/what-is-code-review-why-its-important-for-code-quality#what-is-a-code-review [Accessed 5 Nov. 2023].

Haddad, R. (2022). *Git Branching Strategies: GitFlow, GitHub Flow, Trunk Based...* [online] AB Tasty. Available at: https://www.abtasty.com/blog/git-branching-strategies/.

linearb.io. (n.d.). *The Best Way to Do a Code Review on GitHub | Dev Interrupted Powered by LinearB*. [online] Available at: https://linearb.io/blog/code-review-on-github [Accessed 5 Nov. 2023].

Forsyth, M. (2022). *Git Commit Messages: Best Practices & Guidelines*. [online] Initial Commit. Available at: https://initialcommit.com/blog/git-commit-messages-best-practices#:~:text=Follow%20these%20guidelines%20when%20writing%20good%20commit%20messages%3A [Accessed 5 Nov. 2023].