Software
Architecture Documentation

Easy Food Tracker

13th February, 2024

Noakhali Science and Technology University
Institute of Information Technology

Architectural Document on "Easy Food Tracker"
Course Title: Software Design & Architecture
Course Code: SE - 3211

Submitted By:
Mossa. Sumaiya Akter (ID: BKH2025017F)
Md. Mamun Hossain (ID: MUH2025019M)
Prity Rani Das (ID: BFH2025021F)
Md. Foysal Mahmud (ID: MUH2025025M)
Toriqual Islam Shobuj (ID: MUH2025035M)

Submitted To:
Dipak Chandra Das
Assistant Professor
Institute Of Information Technology
Noakhali Science and Technology University

Submission Date: 13/02/2024

Table of Contents

1. Introduction **4**
1.1 Purpose. 4
1.2 Project Scope 4
1.3 Definitions, Acronyms, and Abbreviation 4
1.3.1 Definitions 4
1.3.2 Acronyms and Abbreviation 5
1.4 References 5
1.5 Overview 5
2. Architecturally Significant Requirements **6**
3. Quality Attributes **7**
3.1 Quality Attribute Scenarios 9
3.1.1 Availability 11
3.1.2 Usability 12
3.1.3 Maintainability 13
3.1.4 Testability 14
3.1.5 Security 15
3.1.6 Performance 16
3.1.7 Scalability 16
3.1.8 Interoperability 17
3.1.9 Reliability 17
3.2 Utility Tree 19
3.3 Design Tractices 19
3.3.1 Availability 20
3.3.2 Interoperability 21
3.3. 3 Performance 22
3.3.4 Security 24
3.3.6 Usability 25
3.3.7 Scalability 26
3.3.8 Reliability 27
3.3.9 Maintainability 27
4. Architectural Representation **28**
4.1 Context Diagram 28
4.2 Use Case Diagram 30
4.3 Logical View 31
4.4 Sequence diagram 32
4.5 Sequence Diagram Description 34
4.6 Deployment View 34
4.7 Implementation View 36

5. Architectural Goals: 36
5.1 Architectural Constraints: 37
5.2 Easy Food Tracker Design Process 40
5.3 Architecture Patterns 41
5.3.1 Model-View-Controller (MVC) pattern 43
5.3.2 Layered Pattern 45
5.3.3 Broker Pattern 46
5.3.4 Client Server Pattern 46
5.3.5 SOA Pattern (Service Oriented Pattern) 48

List of Tables

Table-1: Scenario-1 for Availability 11
Table-2: Scenario-2 for Availability 11
Table-3: Scenario-3 for Availability 11
Table-4: Scenario-4 for Availability 12
Table-5: Scenario-1 for Usability 12
Table-6: Scenario-2 for Usability 13
Table-7: Scenario-3 for Usability 13
Table-8: Scenario-1 for Maintainability 14
Table-9: Scenario-2 for Maintainability 14
Table-10: Scenario-1 for Testability 15
Table-11: Scenario-2 for Testability 15
Table-12: Scenario-1 for Security 16
Table-13: Scenario21 for Security 16
Table-14: Scenario-1 for Performance 17
Table-15: Scenario-1 for Scalability 17
Table-16: Scenario-1 for Interoperability 18
Table-17: Scenario-1 for Reliability 18

List of Figures

Figure-1: Quality Attributes Scenario 22
Figure-2: Utility Tree 23
Figure-3: Availability Tactics 24
Figure-4: Interoperability Tactics 25
Figure-5: Performance Tactics 26
Figure-6: Security Tactics 27
Figure-7: Testability Tactics 27
Figure-8: Usability Tactics 29
Figure-9: Scalability Tactics 29
Figure-10: Reliability Tactics 30
Figure-11: Maintenance Tactics 31
Figure-12 : Context Diagram 32
Figure-13: Use case Diagram 34
Figure-14 : Logical View 34
Figure-15: Sequence Diagram 36
Figure-16: Deployment View 38
Figure-17: Implementation view 40
Figure-18: Easy Food Tracker Design Process 44
Figure- 19: MVC Pattern 47
Figure-20: Layered Pattern 49
Figure-21: Broker Pattern 50
Figure-22: Client Server Pattern 50
Figure-23: SOA pattern 52

1. Introduction

The Software Architecture Documentation is a comprehensive guide that outlines the structural organisation and design principles of a software system. It offers insights into modularization, high-level components, and their relationships. The document explains architectural decisions, patterns, and technologies used, providing a roadmap for developers and facilitating collaboration. It also addresses non-functional requirements like performance and security, ensuring a holistic view of the system's design. This documentation serves as a crucial resource for stakeholders, aiding in decision-making and fostering a shared understanding of the system's structure for effective development, maintenance, and scalability.

1.1 Purpose

The main purpose of this project named “Easy Food Tracker” is to make a user-friendly system for customers who love to experiment on different tested food in restaurants and go to restaurants to eat or test food. It might be helpful for people who are newcomers to Noakhali city and have no idea about the locations of the restaurants. Also this system will be eagerly used to help restaurant owners increase their network and spread their restaurant name among huge people.

1.2 Project Scope

The main scope of this project named Easy Food Tracker is to develop a web-based application. As more than 90% of people use smartphones or PCs and use the internet, they can easily use this application by browsing in any web browser. For this reason, we are targeting to implement our system for web applications. This SRS is also aimed at specifying software requirements to be developed but can also be applied to assist in the selection relation between the different stakeholders. The standard can be used to create software requirements specifications directly or can be used as a model for defining the system requirements.

1.3 Definitions, Acronyms, and Abbreviation

1.3.1 Definitions

Easy Food Tracker: The name of the web-based application designed to help users find and explore restaurants in Bangladesh.

Stakeholders: Individuals or groups with an interest or concern in the Easy Food Tracker system, including customers, restaurant owners, and developers.

Web Application: A software application that runs on a web server and is accessed through a web browser.

Search Bar: A user interface element that allows users to input search queries to find specific information within the Easy Food Tracker application.

Five Star Rating: A rating system where users can assign a rating from one to five stars to rate the quality or experience of a restaurant visited through the Easy Food Tracker application.

1.3.2 Acronyms and Abbreviation

SRS: Software Requirements Specification
PC: Personal Computer
API: Application Programming Interface
UI: User Interface
UX: User Experience
GPS: Global Positioning System
HTML: Hypertext Markup Language
CSS: Cascading Style Sheets
JS: JavaScript
DB: Database

1.4 References

I. Software Requirements, Third Edition(Karl Wiegers and Joy Beatty)

1.5 Overview

Finding best food is hard now a day. Easy Food Tracker will be a web application for all Bangladesh restaurant Owners and all customers who like to travel and eat different delicious food. People can easily find their desired restaurant by using this application and can find out the other place restaurant's views, locations, food information, food prices, Special Offer, Facilities, and Restaurant Quality using the search bar. There will be a five-star rating option for every user who visits the restaurant and by applying this procedure we can identify the friendly restaurant.

2. Architecturally Significant Requirements

Architecturally significant requirements outline the key features and functionalities that directly impact the architecture and design of the software system. Based on the Software Requirements Specification for Easy Food Tracker, here are some architecturally significant requirements:

1. **Web-Based Architecture:** The system shall be designed as a web-based application, allowing users to access it through any standard web browser without needing additional software installation.
2. **Performance Efficiency:** The system should provide quick response times for actions like searching for restaurants, placing orders, and ensuring a smooth user experience.
3. **Scalability:** The architecture should support scalability to accommodate a potentially large number of users and restaurant listings without compromising performance.
4. **Usability:** The system should be user-friendly with an intuitive interface and navigation for both customers and restaurant owners.
5. **Responsive Design:** The user interface shall be designed to be responsive, ensuring a consistent and optimal user experience across different devices and screen sizes, including smartphones, tablets, and desktop computers.
6. **Location-Based Services:** The system shall integrate with location-based services or APIs to enable users to search for nearby restaurants based on their current location.
7. **Real-Time Updates:** The architecture shall support real-time updates for restaurant information, including menu changes, special offers, and user reviews.
8. **Secure Authentication:** The system shall implement secure authentication mechanisms to ensure that only authorised users can access certain features, such as leaving reviews or updating restaurant information.
9. **Reliable Data Storage:** The architecture shall include a reliable and scalable database system to store restaurant listings, user profiles, reviews, and other relevant data.
10. **Integration with External Systems:** The system may need to integrate with external systems or APIs for additional functionalities, such as online payment processing or social media sharing.
11. **Performance Monitoring:** The architecture shall include mechanisms for monitoring system performance, identifying bottlenecks, and optimising resource usage to ensure a smooth user experience.
12. **Compliance with Web Standards:** The system shall adhere to industry-standard web development practices and guidelines, including accessibility standards, browser compatibility, and security best practices.
13. **Maintainability**: Design the system architecture to be easily maintainable, allowing for updates, bug fixes, and improvement.

Collecting Architecturally Significant Requirements (ASR) involves engaging with stakeholders and employing various techniques to identify and prioritize the most critical requirements. Here are ways to collect ASR:

Stakeholder Interviews: Conduct one-on-one or group interviews with key stakeholders, including restaurant owners, customers, and top management.

Utility Trees: Create utility trees to model stakeholders' goals and expectations. Break down high-level goals into specific requirements.

Workshops: Organize collaborative workshops with stakeholders, architects, and developers to discuss system requirements and architecture.

Prototyping: Develop prototypes or proof-of-concept implementations to elicit feedback from stakeholders.

Benchmarking: Perform benchmarking studies to measure and analyze the performance of similar systems.

Use Case Analysis: Analyze critical use cases to identify the key functionalities and performance requirements of the system.

3. Quality Attributes

Quality attributes, also known as non-functional requirements, are the criteria used to evaluate the performance, reliability, usability, maintainability, and other characteristics of a software system. They are essential for ensuring that the software meets the needs and expectations of its users.

Software quality attributes help us to measure the quality of the “Easy Food Tracker” from different angles. It determines software system usefulness and success. Often the software quality attributes safeguard us from reputation damage. Software quality attributes should be introduced at the early stage of the software development life cycle (SDLC). While architecting your application you should consider the quality attribute. If all the software quality attributes are taken care then our application can sustain in the market for a long time.

Adapting software quality attributes is not a one-time task, rather it is continuous and it can evolve. The management should enforce adapting software quality attributes to achieve high standards in “Easy Food Tracker”. Keep in mind that adapting quality attributes at a later stage might be challenging and it can be pricier. As many software quality attributes focus on core-level architecture, it should be introduced at an early stage.

Some quality attributes for “Easy Food Tracker” are described here:

- **Reliability**

Reliability is the ability of our applications to behave as expected and function under the maximum possible load. In Easy Food Tracker users should be able to rely on the system to consistently provide accurate information about restaurants, their menus, and their availability. It should minimise downtime and errors to ensure a smooth user experience.

- **Maintainability**

Maintainability refers to how easily software developers can add new features and update existing features with new technologies. The application architecture plays a critical role in maintainability. The well-architected software makes maintenance easier and more cost-effective. The system should be easy to maintain and update over time. This includes well-documented code, modular design, and adherence to coding standards and best practices to facilitate future enhancements and bug fixes.

- **Performance**

The system should be responsive and performant, even during peak usage times. Response times for actions such as searching for restaurants, placing orders, and loading restaurant profiles should be fast to prevent user frustration.

- **Usability**

The Usability attributes refer to the end user’s ease of use. Usability is tied to application performance, application UX design, and accessibility. To understand usability better let’s consider an e-commerce page the user has purchased an item and wants to return the item. Good usability makes the return option available on the orders page. In some cases, the return option may appear on to Contact Us page – in this situation user easily gets confused and faces difficulty in finding the return option. In our application, the system should be user-friendly and intuitive, allowing both restaurant owners and customers to easily navigate through menus, search for restaurants, place orders, and update profiles without encountering confusion or frustration.

- **Salability**

Scalability is how easily your system can handle increasing demands without affecting the application’s performance. Vertical Scalability and Horizontal Scalability are two primary areas that help to meet the scalability criteria. The system should be able to handle a growing user base and increasing amounts of data without sacrificing performance. It should be scalable both horizontally (adding more servers) and vertically (increasing server capacity) to accommodate future growth.

- **Testability**

Testability is how easily QA members can test the software and log a defect and how easy it is to automate the software applications. Your application design should focus on making the

testing easier and faster. Easy Food Tracker ensures that testing processes are efficient, effective, and thorough, ultimately leading to the delivery of a high-quality and reliable software product.

- **Security**

Security attribute focuses on the ability to safeguard applications, data, and information from unauthorised entities. This is very crucial as the data leaks may incur huge losses.As the system deals with sensitive information such as user accounts, payment details, and restaurant profiles, it must implement robust security measures to protect data confidentiality, integrity, and availability. This includes secure authentication, encryption of sensitive data, and protection against common security threats like SQL injection and cross-site scripting (XSS).
Interoperability
Interoperability refers to the ability to communicate or exchange data between different systems. Which can be operating systems, databases, or protocols. The interoperability problem arises due to the legacy code base, poorly architectured applications, and poor code quality.

- **Availability**

Availability is the percentage of time the application is available for use. 100% availability refers to the system being always available and never going down under any circumstances. Easy Food Tracker should be highly available, ensuring that users can access it whenever they need to. This includes implementing redundancy and failover mechanisms to minimize downtime in case of server failures or maintenance activities.

3.1 Quality Attribute Scenarios

Quality attribute scenarios, also known as quality attribute requirements, are specific scenarios or use cases that describe how a system should behave under certain conditions in order to satisfy a particular quality attribute. These scenarios are used to elicit, define, and specify requirements related to various quality attributes such as performance, reliability, security, and usability. Quality attribute scenarios typically include the following components:

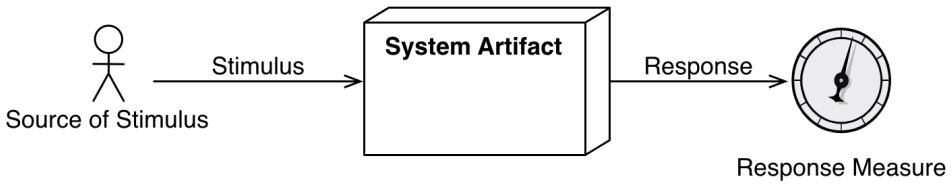

Figure-1: Quality Attributes Scenario

3.1.1 Availability

Table-1: Scenario-1 for Availability

Scenario	The application server fails or becomes unresponsive, causing the entire system to fail for too many customer requests.
Source	Internal to application
Stimulus	Fault within the application server
Artifact	Application server
Environment	Normal operating conditions, with the system serving customer requests.
Response	The application should detect the failure of the application server promptly and initiate failover procedures to switch to a backup or redundant server.
Response Measure	● The application should have a monitoring mechanism in place to continuously check the health and availability of the application server. ● The application should automatically redirect traffic to a backup server within X seconds/minutes.

Table-2: Scenario-2 for Availability

Scenario	The application is unable to handle a large number (over 6,000) of concurrent customer requests such as searching for restaurants at the same time.
Source	External to application
Stimulus	The application receives more concurrent requests that it can handle
Artifact	Web server
Environment	Normal operation
Response	The application blocks additional concurrent customer requests and displays a message to the customers to try accessing the system again later.
Response Measure	Load time(Takes no longer than 10 seconds for pages to load)

Table-3: Scenario-3 for Availability

Scenario	A bug or fault in the application causes a system-wide failure such as showing wrong results for searching for food or restaurants.
Source	Internal to application
Stimulus	A bug or fault in the application
Artifact	The component in which the bug or fault occurred
Environment	Degraded
Response	The application handles any and all exceptions that may occur, so that the system may fail gracefully.
Response Measure	A meaningful error message is logged, indicating what it was that caused the application to fail

Table-4: Scenario-4 for Availability

Scenario	The application is compromised by a Denial of Service (Dos) attack.
Source	External to application
Stimulus	The system receives more requests per second than it can handle
Artifact	Web service
Environment	Normal operation
Response	Standard response according to RIT systems and Operations protocol.
Response Measure	Length of attack (The Dos attack does not continue for longer than 1 minute).

3.1.2 Usability

Table-5: Scenario-1 for Usability

Scenario	The customer wants to discover what features are available to them.
Source	Customer
Stimulus	Customer wants to learn application features
Artifact	Web service
Environment	At runtime
Response	The application provides an interface that feels familiar to the customer and follows good practice in website interface design to improve learnability
Response Measure	Number of errors in completing a task, ratio of successful operations to total operations.

Table-6: Scenario-2 for Usability

Scenario	The customer wants quick access to core features such as ordering food, search for restaurants, viewing restaurant profiles for their user class to improve efficiency of use.
Source	Customer
Stimulus	Customer wants to improve efficiency
Artifact	Web service
Environment	At runtime
Response	The application displays pertinent information by default on the customer's home screen without forcing them to dig into nested menus to find the information for which they are looking.
Response Measure	Task time, average 5 seconds

Table-7: Scenario-3 for Usability

Scenario	The customer wants to receive user and situation appropriate error messages when an error occurs.
Source	Customer

Stimulus	Minimise impact of errors
Artifact	Web service
Environment	At runtime
Response	The application will provide visual feedback stating whether or not a given action was successful. If it was not successful, the application will provide details on what went wrong and how to rectify the situation, when possible. Furthermore, the system will send the customer a followup email with the status of any submissions.
Response Measure	Customer satisfaction, amount of time lost, amount of data lost

3.1.3 Maintainability

Table-8: Scenario-1 for Maintainability

Scenario	Lack of documentation, management and future upgrades.
Source	Developer, Maintainer
Stimulus	Documentation was not made a priority throughout the development of the application and thus does not provide the most uptodate information on the system features and functionality
Artifact	System documentation
Environment	At runtime
Response	Developers spend time improving documentations.
Response Measure	Time spent understanding the application that would have been aided by more robust documentation.

Table-9: Scenario-2 for Maintainability

Scenario	Excessive dependencies between components and layers and inappropriate coupling to concrete classes prevents easy replacement, updates, and changes.
Source	Developer, Maintainer

Stimulus	Developers wish to replace, update or modify part of the application.
Artifact	System documentation
Environment	At runtime
Response	Re-design the system with well defined layers, or areas of concern, that clearly delineate the application's UI, business processes, and data access functionality. Application configuration for commonly changed parameters, such as URLs, will be maintained outside the code base.
Response Measure	Time spent redesigning and refactoring.

3.1.4 Testability

Table-10: Scenario-1 for Testability

Scenario	There is a lack of test planning
Source	System verifier
Stimulus	Subsystem integration completed. Testing did not start early during the development life cycle
Artifact	System documentation
Environment	At runtime
Response	Prepare a test environment.
Response Measure	Test coverage, percent executable statements executed, length of time to prepare test environment.

Table-11: Scenario-2 for Testability

Scenario	Possible tests such as unit testing
Source	Unit tester
Stimulus	Code unit, completed
Artifact	Code unit
Environment	Development

Response	Results captured
Response Measure	85% path coverage in three hours.

3.1.5 Security

Table-12: Scenario-1 for Security

Scenario	Sensitive customer information, such as credit card details, in a database. To protect this data from unauthorised access, they implement encryption and authentication measures.
Source	An external actor attempting to gain unauthorised access to the sensitive data stored in the company's database.
Stimulus	The external actor sends a request to access the sensitive data stored in the database.
Artifact	The sensitive customer information stored in the database, such as credit card details, is the artifact that the external actor is attempting to access.
Environment	The database servers, network architecture, encryption algorithms, authentication protocols, and security policies.
Response	The application verifies the identity of the actor attempting to access the data. This could involve checking credentials such as username, passwords.
Response Measure	The percentage of access attempt that are successfully authenticated.

Table-13: Scenario21 for Security

Scenario	The application is transferring sensitive data over a network and wants to ensure that it remains confidential during transit.
Source	The sensitive data that needs to be encrypted, possibly including financial records, personal information or proprietary business data.
Stimulus	The action of sending the sensitive data over the network, either from one system to another or from a client to a server.
Artifact	The sensitive data itself, which needs to be protected from unauthorised access or interception while in transit.
Environment	The network infrastructure over which the data is being transmitted, including routers, switches, and communication

	channels. This also includes the systems sending and receiving the data, as well as any intermediate systems through which the data passes.
Response	The encryption process applied to the sensitive data before transmission. This involves using an encryption algorithm and a cryptographic key to transform the data into ciphertext, which is unreadable without the corresponding decryption key.
Response Measure	The level of security provided by the chosen encryption algorithm and key length. Stronger encryption algorithms and longer keys generally provide higher levels of security.

3.1.6 Performance

Table-14: Scenario-1 for Performance

Scenario	Users initiate transactions under normal operations. The system processes the transactions with an average latency of two seconds.
Source	Customers, Restaurant owner
Stimulus	Initiate transactions
Artifact	System
Environment	Normal Operation
Response	Transactions are processed
Response Measure	Average latency of 2 seconds

3.1.7 Scalability

Table-15: Scenario-1 for Scalability

Scenario	Handling Increased User Traffic Through Load Balancing
Source	External to the system
Stimulus	A sudden increase in customer traffic due to a promotional campaign or peak hours.

Artifact	Load Balancer
Environment	Normal operating conditions, with the system serving customer requests.
Response	The application should distribute incoming customer requests across multiple applications servers to prevent overload and ensure optimal performance.
Response Measure	The load balancer should continuously monitor the traffic load on each application server.

3.1.8 Interoperability

Table-16: Scenario-1 for Interoperability

Scenario	Integration with external payment gateway
Source	External to the system
Stimulus	User initiates a payment transaction on Easy Food Tracker platform.
Artifact	Payment gateway API
Environment	Normal operating conditions, with the system facilitating user transactions.
Response	The application should seamlessly communicate with external payment gateway to process customer transaction.
Response Measure	The integration should handle various payment methods, currencies, and transaction types supported by the payment gateway.

3.1.9 Reliability

Table-17: Scenario-1 for Reliability

Scenario	Database failure handling
Source	Internal to the system
Stimulus	Database server failure or data corruption.

Artifact	Database Management system (DBMS)
Environment	Normal operating conditions, with the system facilitating user transactions.
Response	The application should gracefully handle database failures to prevent data loss and ensure uninterrupted service.
Response Measure	The application should implement data redundancy and backup mechanisms to minimise the risk of data loss in case of database failures.

3.2 Utility Tree

Utility trees are a way to organise these quality attributes. In regard to ATAM they serve as a way to prioritise quality attributes and later to evaluate the suitability of a candidate architecture vs. the requirements. It also represents the utility or value provided by different components, modules, or features of a software system. It helps stakeholders, architects, and developers understand how various parts of the system contribute to achieving the overall objectives and requirements.

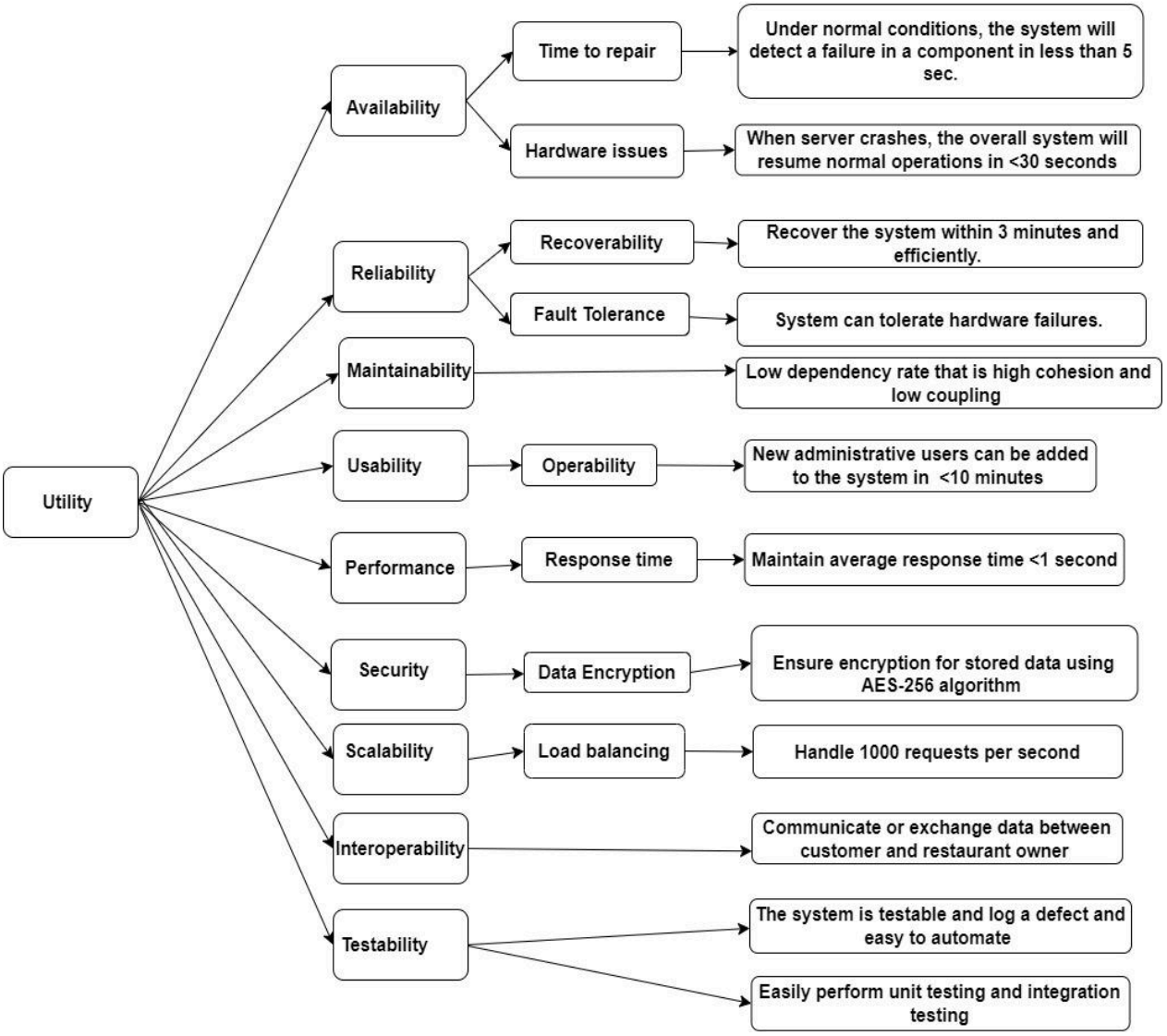

Figure-2: Utility Tree

3.3 Design Tractices

Design tactics, in the context of software engineering, refer to specific strategies or techniques employed during the design phase of a system to address architectural concerns, achieve quality attributes, or meet design goals. These tactics are applied to various components or aspects of the system's architecture to improve its overall quality, performance, maintainability, and other key characteristics.

Design tactics are typically derived from design patterns, architectural styles, and best practices, and they are tailored to the specific requirements and constraints of the system being developed. They help guide the decision-making process during system design and implementation.

3.3.1 Availability

A failure occurs when the system no longer delivers a service that is consistent with its specification; this failure is observable by the system's actors. A fault (or combination of faults) has the potential to cause a failure. Availability tactics, therefore, are designed to enable a system to endure system faults so that a service being delivered by the system remains compliant with its specifications. The tactics we discuss in this section will keep faults from becoming failures or at least bound the effects of the fault and make repair possible. Availability tactics may be categorised as addressing one of three categories: fault detection, fault recovery, and fault prevention.

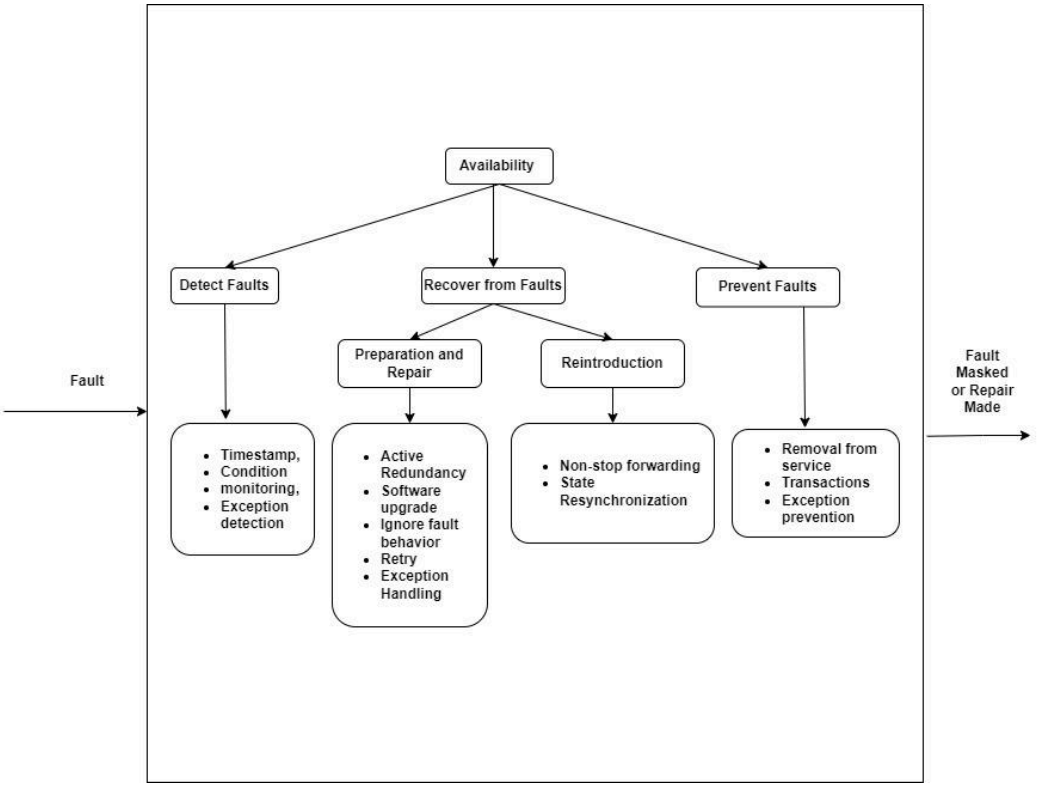

Figure-3: Availability Tactics

3.3.2 Interoperability

Interoperability is about the degree to which two or more systems can usefully exchange meaningful information via interfaces in a particular context. The definition includes not only having the ability to exchange data (syntactic interoperability) but also having the ability to correctly interpret the data being exchanged (semantic interoperability). A system cannot be interoperable in isolation.

Interoperability is affected by the systems expected to interoperate. If we already know the interfaces of external systems with which our system will interoperate, then we can design that knowledge into the system. Or we can design our system to interoperate in a more generic fashion, so that the identity and the services that another system provides can be bound later in the life cycle, at build time or runtime.

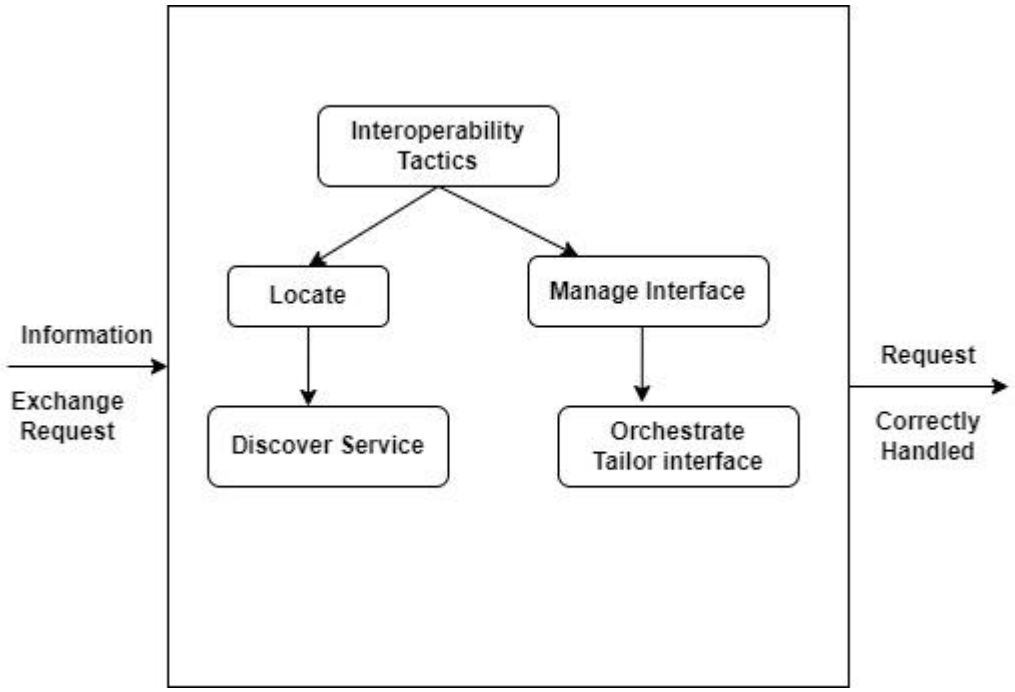

Figure-4: Interoperability Tactics

3.3. 3 Performance

Performance, that is: It's about time and the software system's ability to meet timing requirements. When events occur interrupts, messages, requests from users or other systems, or clock events marking the passage of time the system, or some element of the system, must respond to them in time. Characterising the events that can occur (and when they can occur) and the system or element's time based response to those events is the essence of discussing performance. Performance is often linked to scalability, that is, increasing your system's

capacity for work, while still performing well. Technically, scalability is making your system easy to change in a particular way, and so is a kind of modifiability. The goal of performance tactics is to generate a response to an event arriving at the system within some time-based constraint. The event can be single or a stream and is the trigger to perform computation. Performance tactics control the time within which a response is generated.

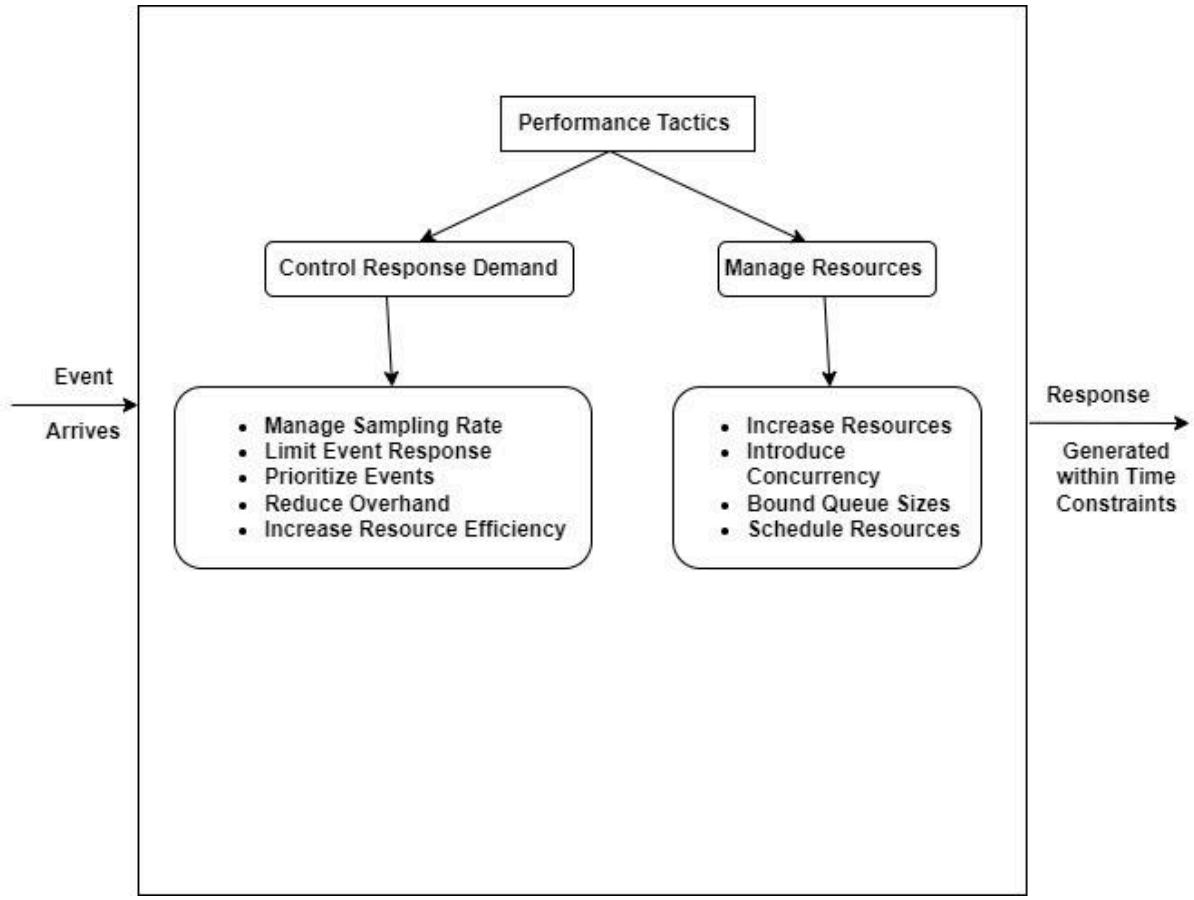

Figure-5: Performance Tactics

3.3.4 Security

Security is a measure of the system's ability to protect data and information from unauthorised access while still providing access to people and systems that are authorised. An action taken against a computer system with the intention of doing harm is called an attack and can take a number of forms. It may be an unauthorised attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.
Several tactics are intended to respond to a potential attack:

- Revoke access: If the system or a system administrator believes that an attack is underway, then access can be severely limited to sensitive resources, even for normally legitimate users and uses. For example, if your desktop has been compromised by a virus, your access to certain resources may be limited until the virus is removed from your system.
- Lock the computer: Repeated failed login attempts may indicate a potential attack. Many systems limit access from a particular computer if there are repeated failed attempts to access an account from that computer. Legitimate users may make mistakes in attempting to log in. Therefore, the limited access may only be for a certain time period.
- Inform actors: Ongoing attacks may require action by operators, other personnel, or cooperating systems. Such personnel or systems the set of relevant actors must be notified when the system has detected an attack.

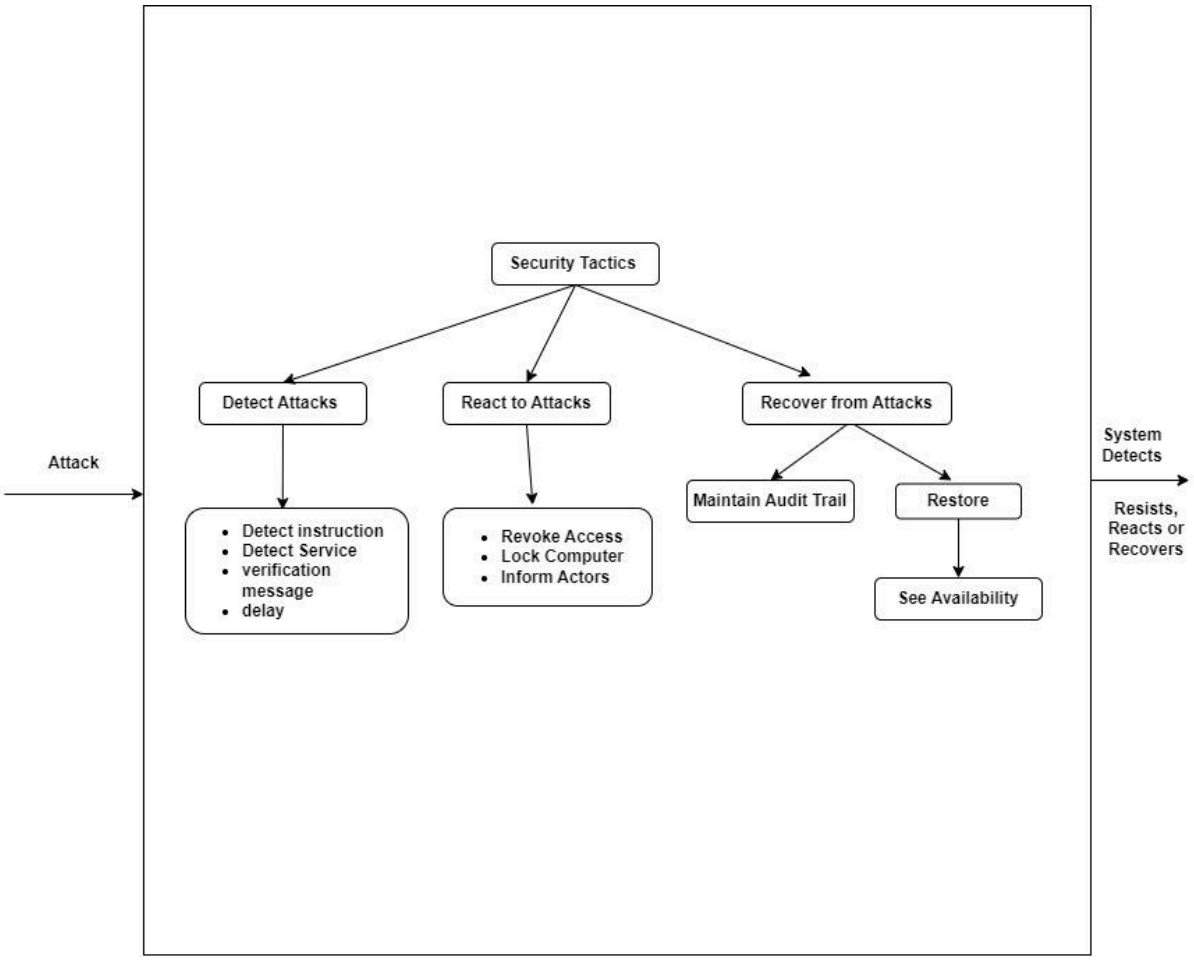

Figure-6: Security Tactics

3.3.5 Testability

The goal of tactics for testability is to allow for easier testing when an increment of software development is completed.Architectural techniques for enhancing the software testability have not received as much attention as more mature quality attribute disciplines such as modifiability, performance, and availability, but as we stated before, anything the architect can do to reduce the high cost of testing will yield a significant benefit.

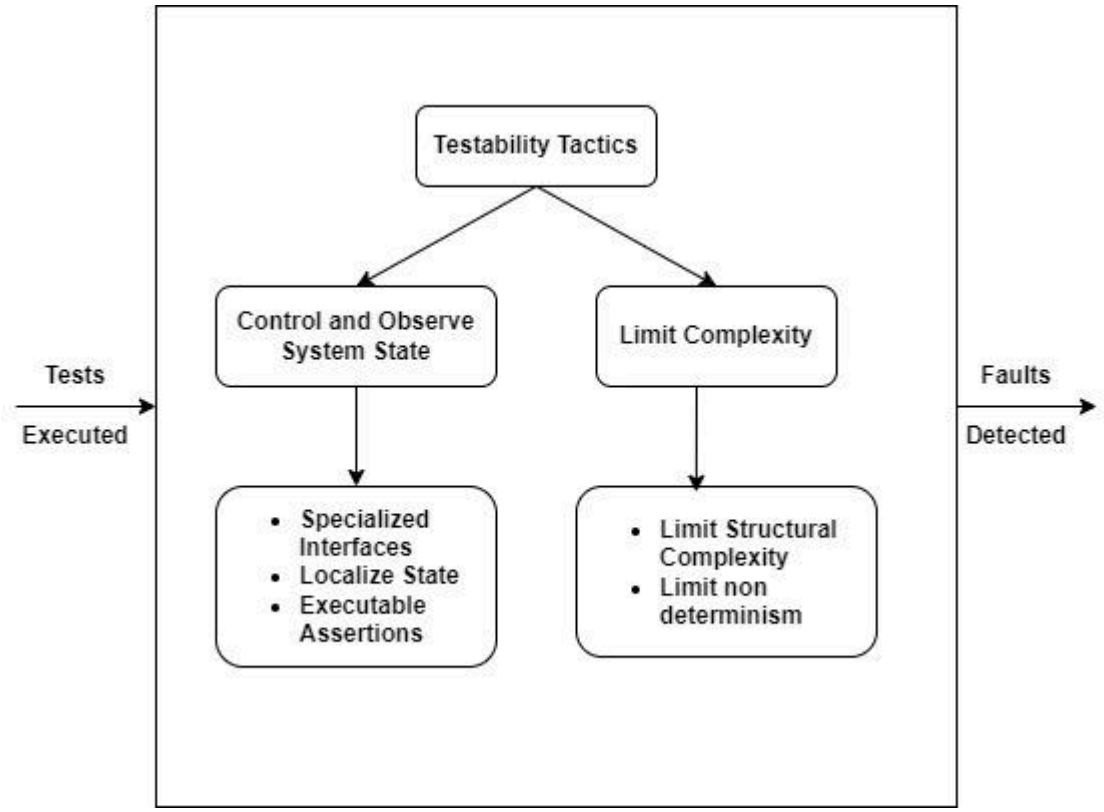

Figure-7: Testability Tactics

3.3.6 Usability

Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides. Over the years, a focus on usability has shown itself to be one of the cheapest and easiest ways to improve a system's quality (or more precisely, the user's perception of quality).

Usability comprises the following areas:

- Learning system features. If the user is unfamiliar with a particular system or a particular aspect of it, what can the system do to make the task of learning easier? This might include providing help features.
- Using a system efficiently. What can the syste1n do to make the user more efficient in its operation? This might include the ability for the user to redirect the system after issuing a command. For example, the user may wish to suspend one task, perform several operations, and then resume that task.

- Minimising the impact of errors. What can the system do so that a user error has minimal impact? For example, the user may wish to cancel a command issued incorrectly.
- Adapting the system to user needs. How can the user (or the system itself) adapt to make the user's task easier? For example, the system may automatically fill in URLs based on a user's past entries.
- Increasing confidence and satisfaction. What does the system do to give the user confidence that the correct action is being taken? For example, providing feedback that indicates that the system is performing a long-running task and the extent to which the task is completed will increase the user's confidence in the system.

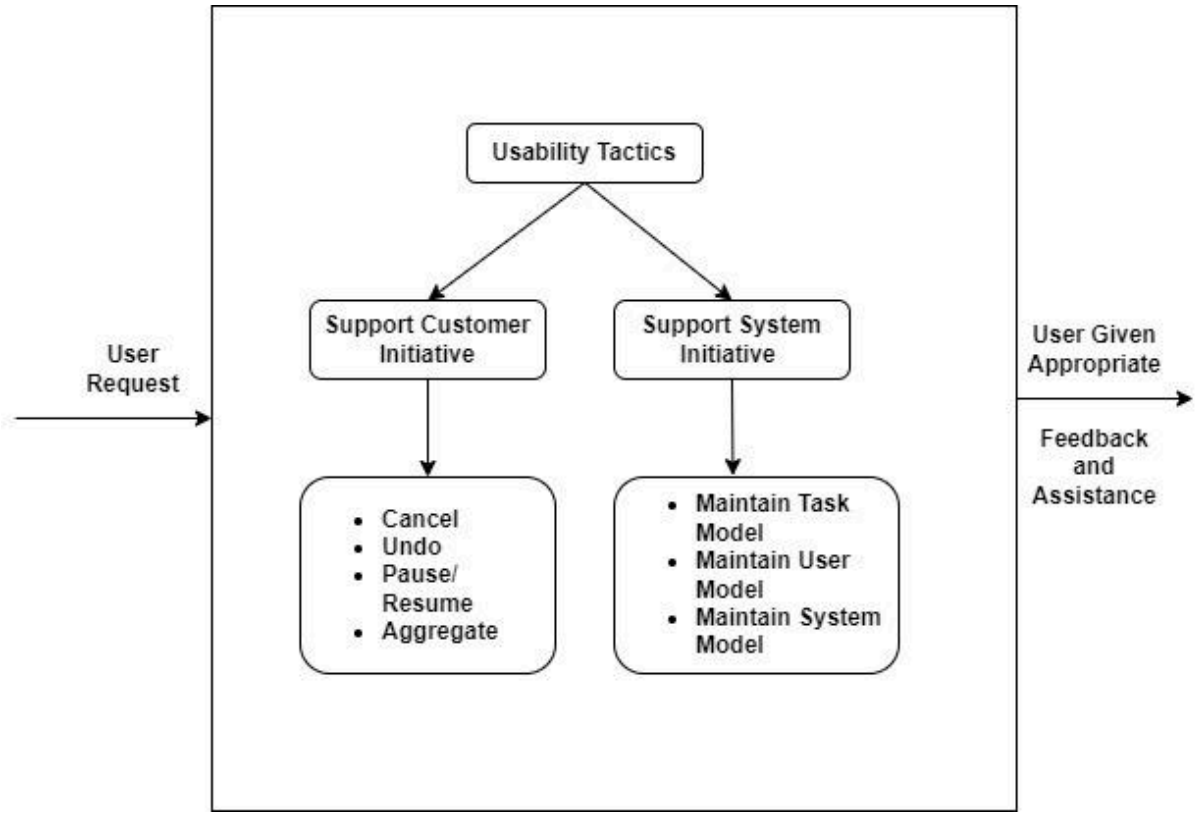

Figure-8: Usability Tactics

3.3.7 Scalability

Two kinds of scalability are horizontal scalability and vertical scalability. Horizontal scalability (scaling out) refers to adding more resources to logical units, such as adding another server to a cluster of servers. Vertical scalability (scaling up) refers to adding more resources to a physical unit, such as adding more memory to a single computer. The problem that arises with either type of scaling is how to effectively utilise the additional resources. Being effective means that the additional resources result in a measurable improvement of some system quality, did not require undue effort to add, and did not disrupt operations. In cloud environments, horizontal scalability is called elasticity.

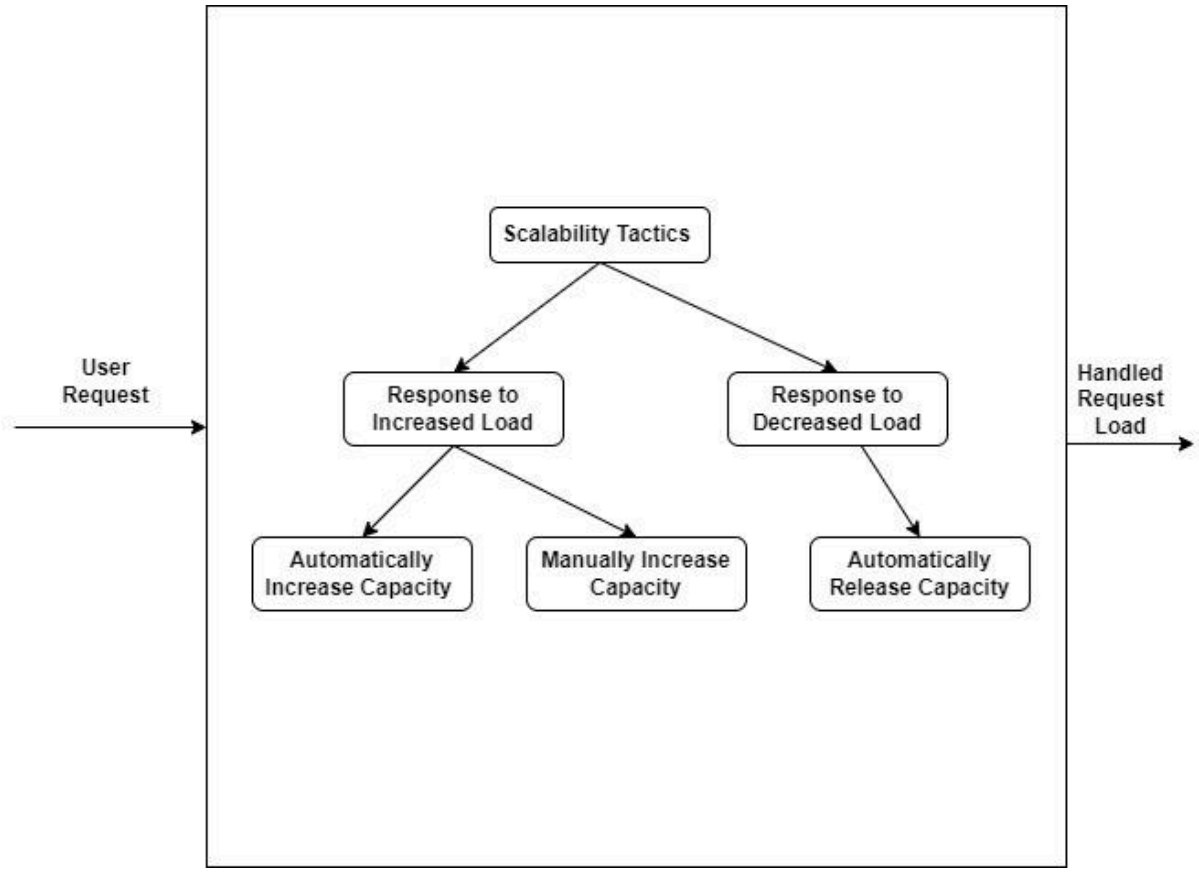

Figure-9: Scalability Tactics

3.3.8 Reliability

Reliability refers to the probability that a system will perform its intended function adequately for a specified period of time under stated conditions. In simpler terms, it's about how consistently a system operates without failure. Reliability engineering aims to identify and mitigate potential points of failure within a system to ensure that it meets its performance requirements over its operational

lifespan. Reliability focuses on ensuring consistent performance over time, while fault tolerance focuses on maintaining system functionality in the face of component failures. Both are crucial aspects of system design and operation, particularly in mission-critical or high-availability environments such as data centres, telecommunications networks, and aerospace systems.

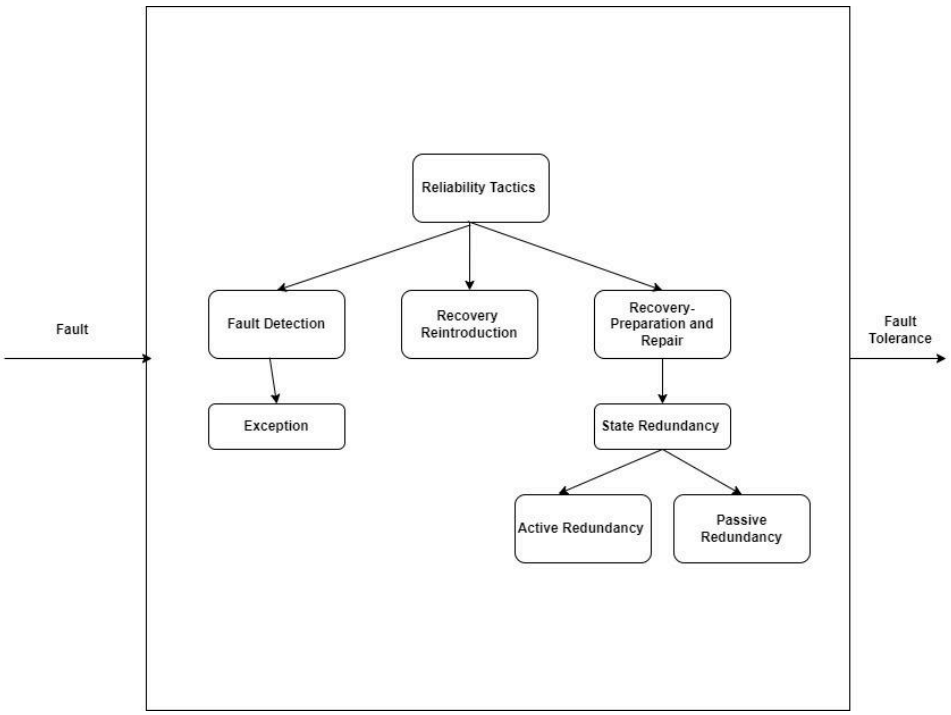

Figure-10: Reliability Tactics

3.3.9 Maintainability

Maintainability is a crucial aspect of software engineering and system design, referring to the ease with which a system or software can be maintained, modified, or repaired. A system or software application that is highly maintainable is typically easier and less costly to manage over its lifecycle.

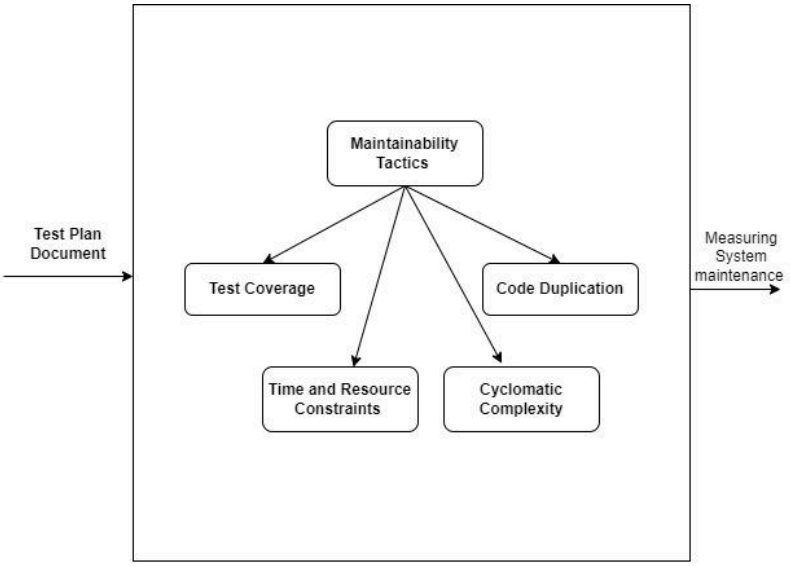

Figure-11: Maintenance Tactics

4. Architectural Representation

4.1 Context Diagram

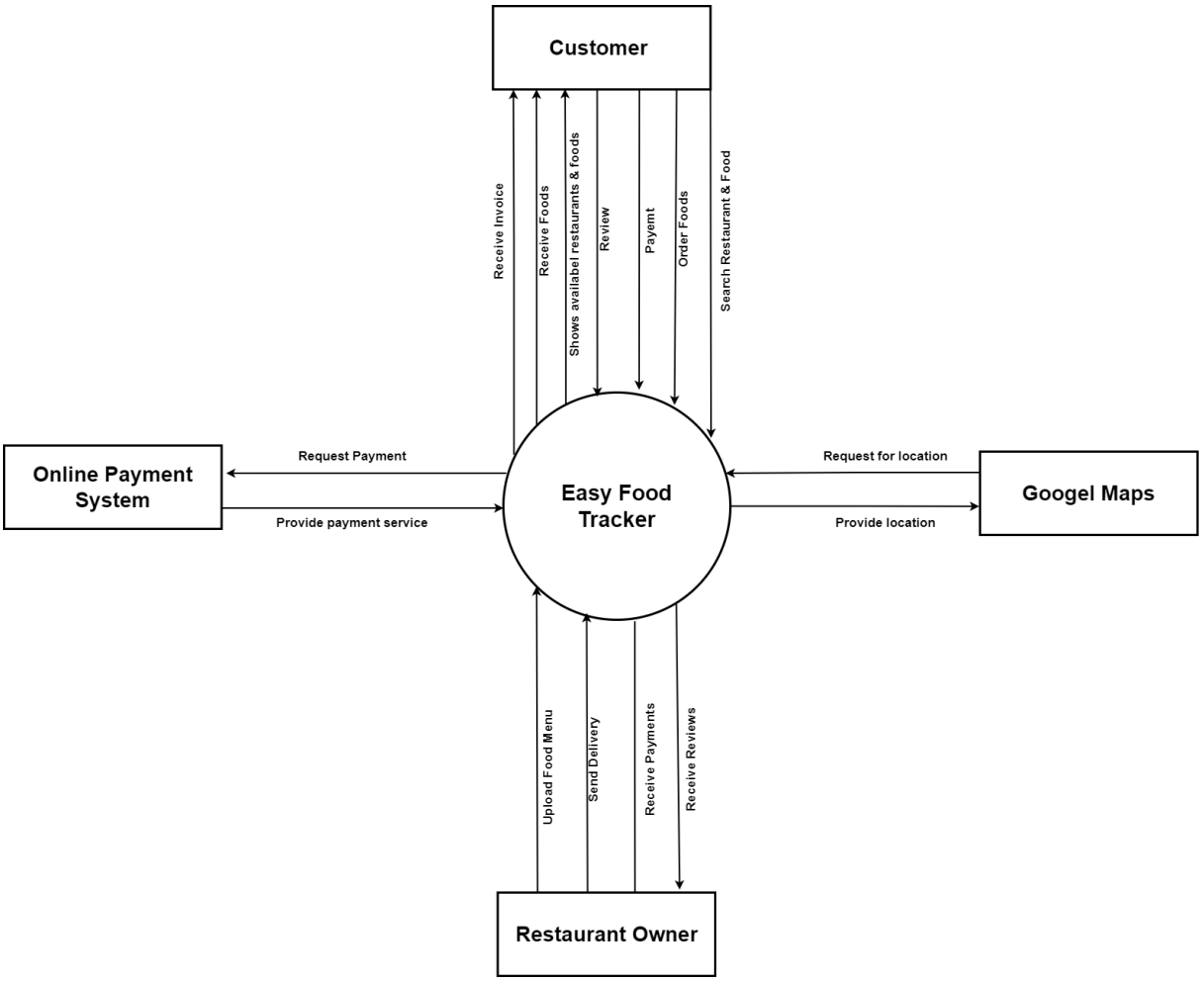

Figure-12 : Context Diagram

A context diagram provides a high-level view of a system and its interactions with external entities. In the case of the "Easy Food Tracker" system, the context diagram outlines the system's boundaries and the external entities it interacts with. Here's a brief description of the diagram:

Easy Food Tracker Context Diagram

The "Easy Food Tracker" system is depicted at the centre of the diagram, representing the core system being developed. Surrounding the system are four external entities:

Customer: This represents individuals who interact with the system to place food orders. Customers provide input to the system by selecting food items, specifying delivery locations, and making payments.
Restaurant Owner: This external entity represents the owners or managers of restaurants participating in the system. Restaurant owners interact with the system to receive and fulfill food orders placed by customers.
Online Payment System: This entity signifies the external service or system responsible for processing online payments. It handles financial transactions between customers and the system, ensuring secure and efficient payment processing.
Google Maps: Google Maps is depicted as an external entity, suggesting integration with mapping services for location-based features. The system may utilize Google Maps to facilitate delivery route planning, location tracking, or providing accurate delivery estimates to customers.

Interactions:

- Customers interact with the system to search restaurants or foods, place food orders, make payments,and give reviews.
- Restaurant owners interact with the system to receive and manage incoming orders, update menu items, and monitor order statuses.
- The online payment system interfaces with the system to process payments securely and efficiently.
- The system interacts with Google Maps to utilize mapping services for features such as address validation, geolocation, and route optimization.

Overall, the context diagram provides a clear overview of the "Easy Food Tracker" system's external interactions, helping stakeholders understand its scope and boundaries. It serves as a foundation for further detailing system requirements and design considerations.

4.2 Use Case Diagram

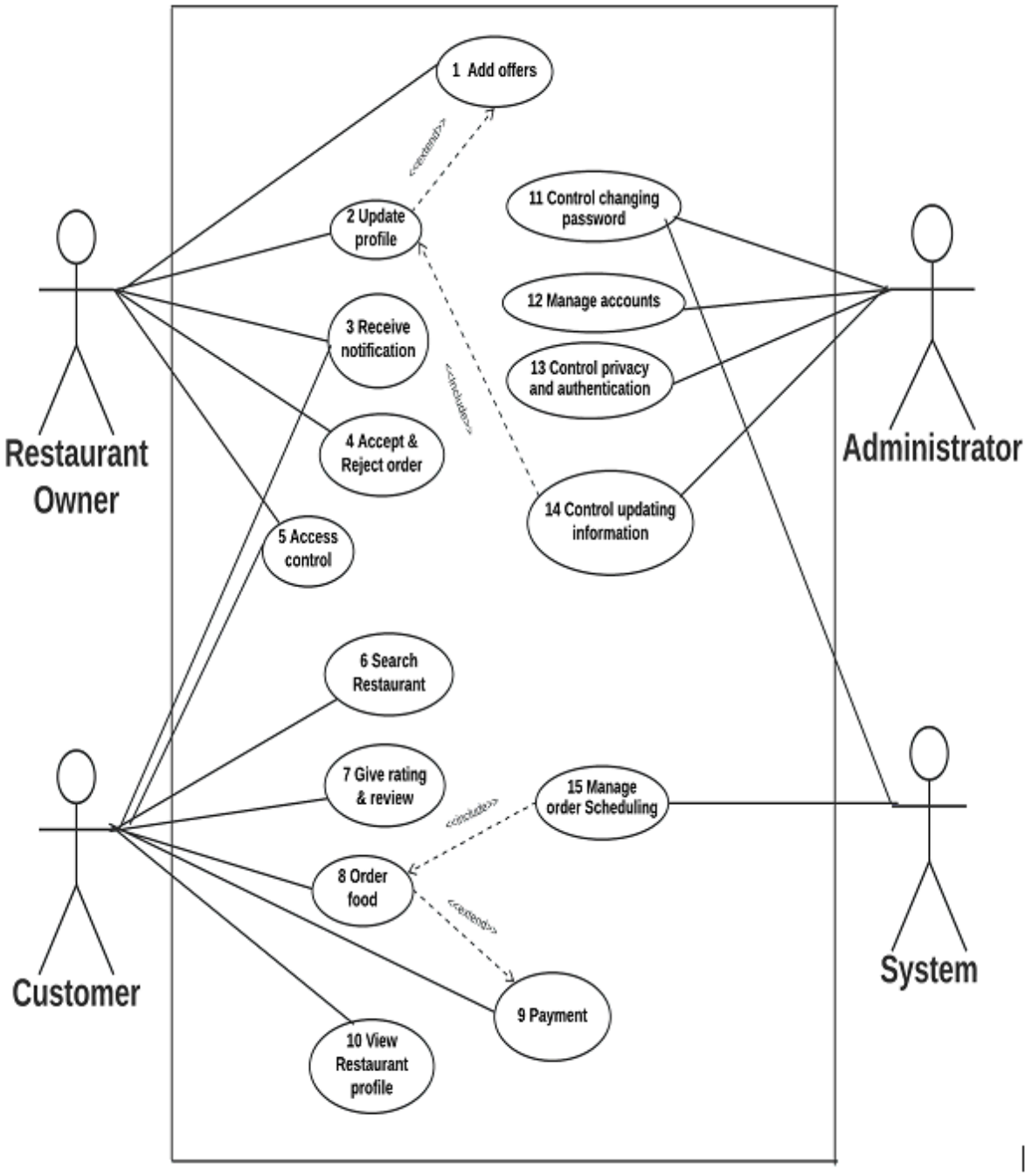

Figure-13: Use case Diagram

4.3 Logical View

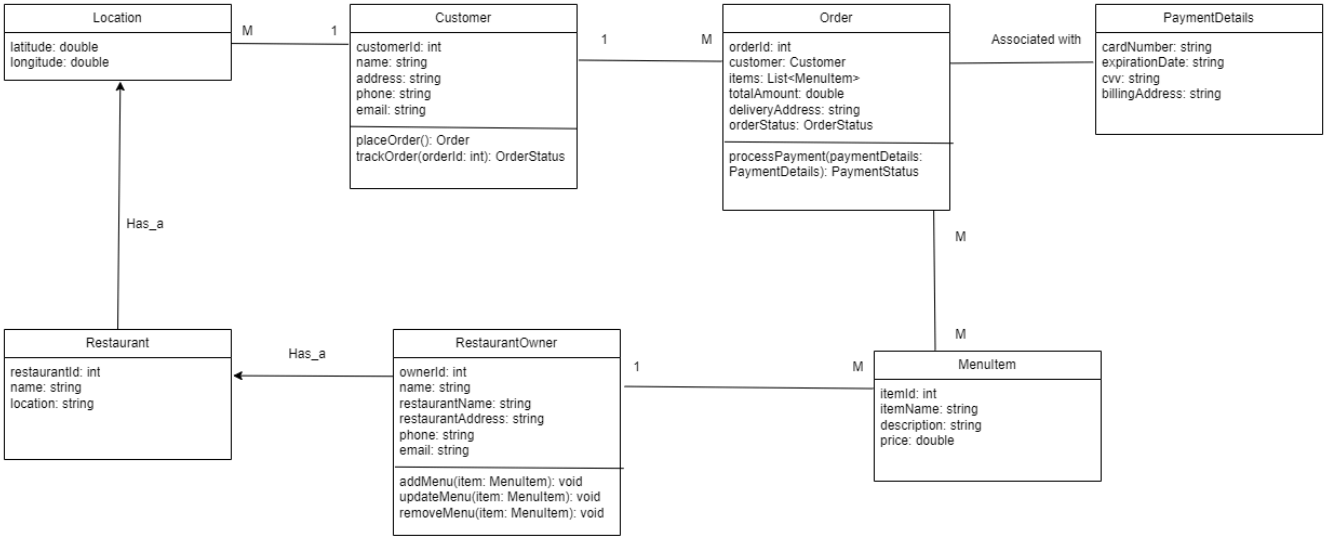

Figure-14 : Logical View

This UML class diagram represents the logical view of the Easy Food Tracker, including the main functions and components described. Each major function has its corresponding module, and components are encapsulated within the respective modules. The diagram illustrates the relationships and dependencies among different modules and functions in the system.

Customer:

- Functions: Place order, track orders, search restaurants and foods.
- Components: Search module, Order module.

Restaurant Owner:

- Functions:Upload menu, receive order, receive payment, deliver food
- Components: Upload menu module, receive payment module, deliver food module

Payment:

- Functions: Online payment system
- Components:Call payment API.

4.4 Sequence diagram

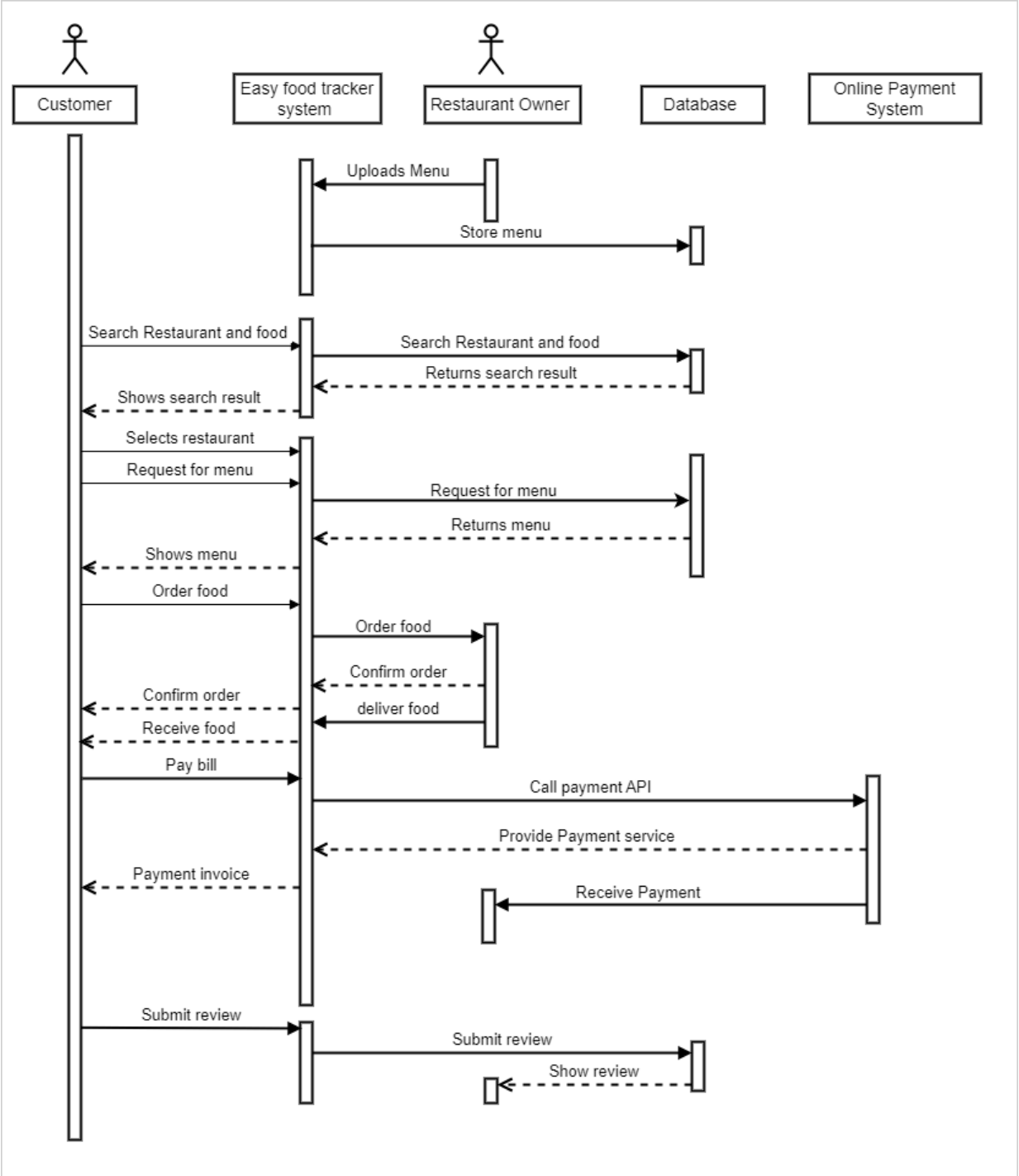

Figure-15: Sequence Diagram

Sequence Diagram Description

1.Customer Places Order

- The sequence begins with the customer initiating an order by interacting with the system's user interface.
- The system receives the order request from the customer and processes it.
- This involves selecting food items from the menu, specifying delivery details, and adding payment information.

2.System Notifies Restaurant Owner

- Upon receiving the order, the system notifies the respective restaurant owner or manager about the new order.
- The notification prompts the restaurant owner to view and process the incoming order.

3.Restaurant Owner Processes Order

- The restaurant owner accesses the system's interface to view the details of the incoming order.
- They confirm the order, prepare the food items, and mark the order as ready for delivery.

4.System Communicates with Online Payment System

- As the order is confirmed, the system initiates a payment transaction with the online payment system.
- The payment system processes the payment securely and confirms the transaction status back to the system.

5.Customer Receives Confirmation

- Once the payment is successfully processed, the system notifies the customer about the payment
- The customer receives an order confirmation message along with payment details.

6.Order Delivered

- The delivery person arrives at the customer's specified location and completes the delivery.
- The system updates the order status to "delivered," and the customer receives a notification confirming successful delivery.

This sequence diagram outlines the interactions and flow of events within the "Easy Food Tracker" system, illustrating how customers place orders, restaurant owners process them, and payments are handled.

4.5 Deployment View

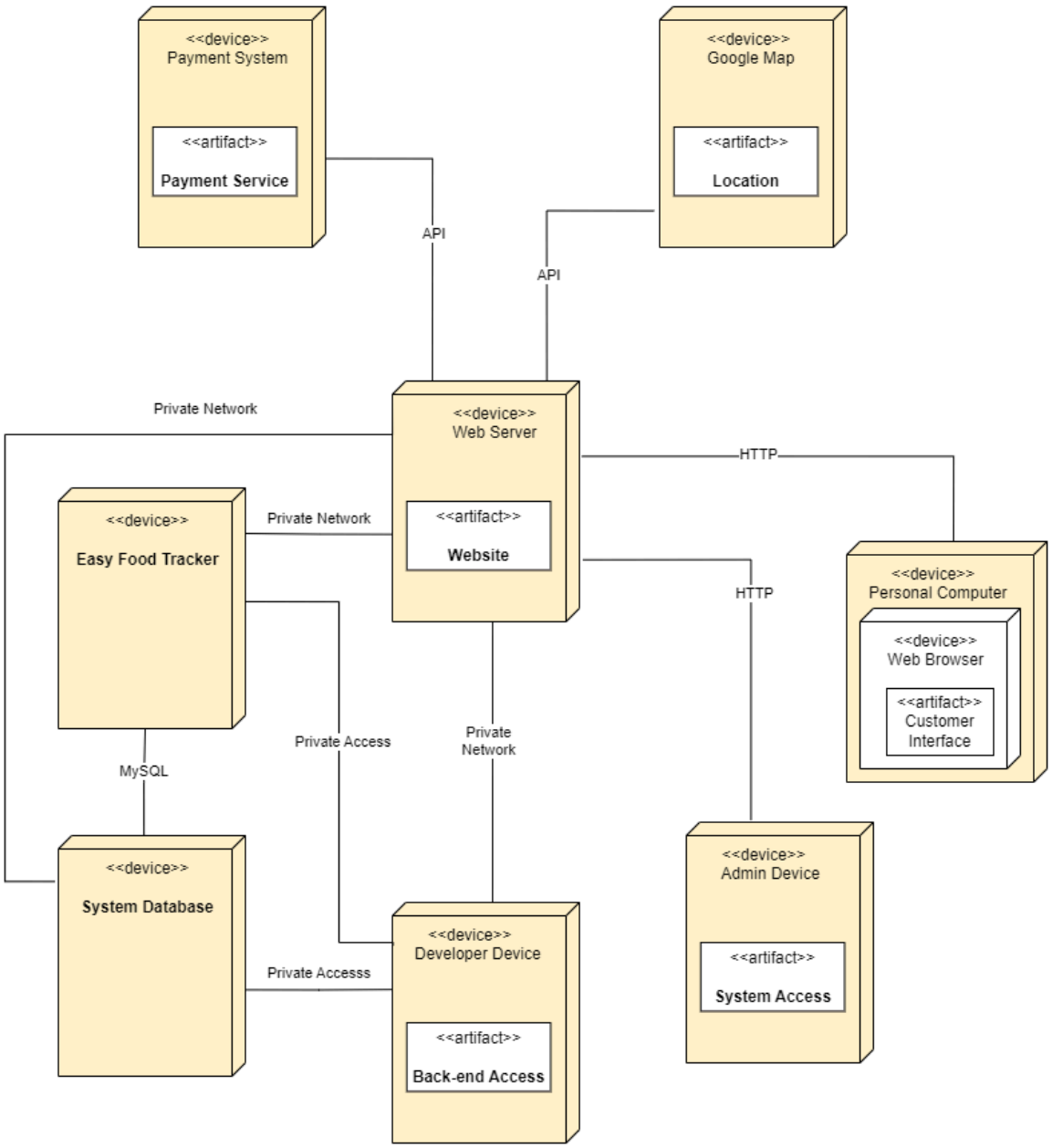

Figure-16: Deployment View

1. Server Infrastructure:

Components:

- Web Server: Hosts the frontend components and handles user interactions.
- Application Server: Executes the backend logic and business processes.
- Database Server: Manages the storage and retrieval of data.

2. Client Devices:

Components:

- Web Browser: Renders the frontend interfaces for users.
- Mobile Devices: Devices used by customer and restaurant owner to interact with the system.

3. Communication Protocols:

Components:

- HTTP/HTTPS: Used for communication between the web server and clients.
- API: Used for communication between the system and third party app like google map and online payment system

4. Database System:

Components:

- Relational Database Management System (RDBMS): Manages and stores data in a structured manner.

4.6 Implementation View

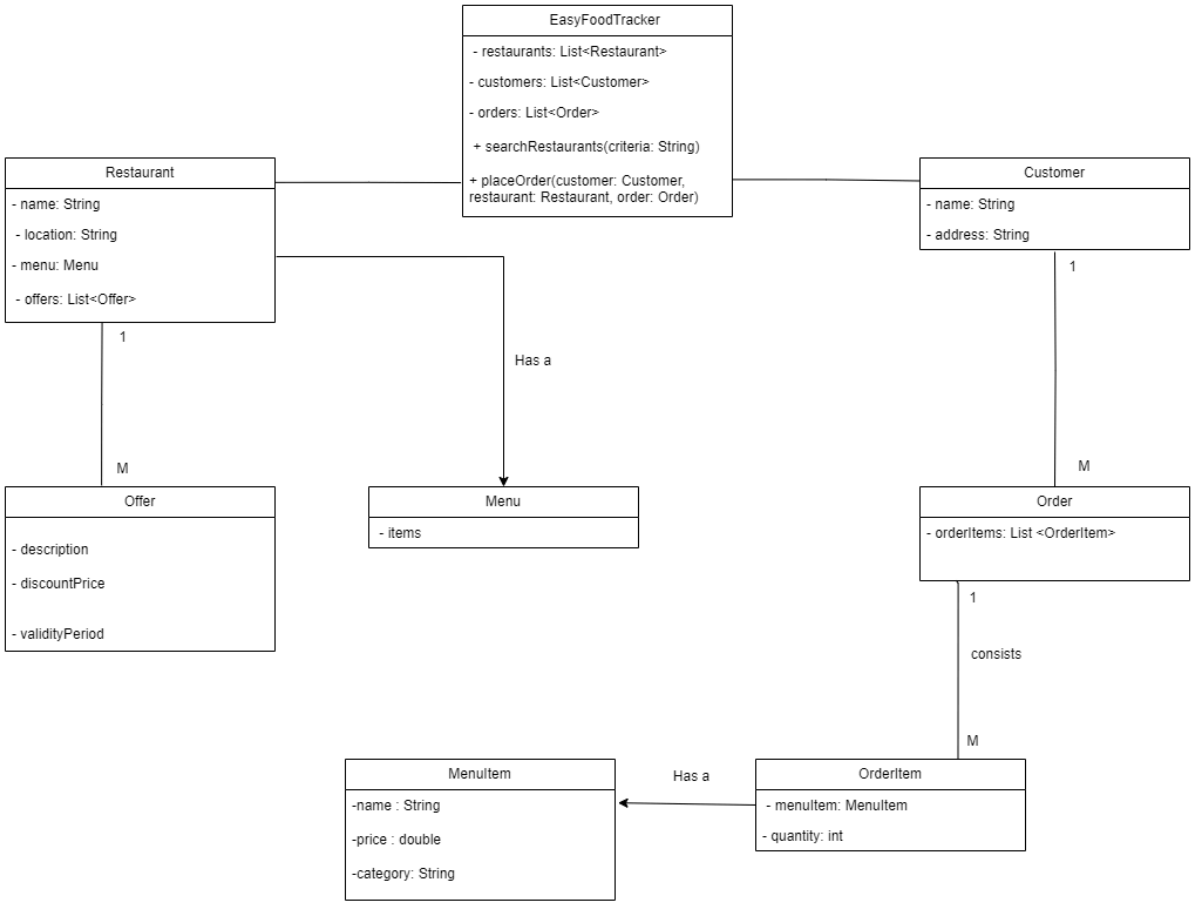

Figure-17: Implementation view

5. Architectural Goals:

Scalability:The system architecture should be scalable to accommodate a growing number of users, restaurants, and data without compromising performance.

Performance:Achieve high performance to ensure quick response times for users when accessing restaurant information, placing orders, and performing other activities.

Security:Implement robust security measures to protect user data, transactions, and ensure secure access control for both customers and restaurant owners.

Usability: Design an architecture that promotes user-friendly interfaces, simple navigation, and a positive overall user experience for both customers and restaurant owners.

Reliability:Develop a reliable system architecture that minimises downtime, ensuring users can access the platform consistently.

Compatibility: Ensure compatibility with various devices, browsers, and operating systems to maximise accessibility for a diverse user base.

Maintainability: Design the system architecture to be easily maintainable, allowing for updates, bug fixes, and improvements over time.

Integration: Facilitate integration with external services or APIs that may enhance the functionality of the Easy Food Tracker system.

5.1 Architectural Constraints:

Technology Stack: The system architecture must work within the specified technology stack, limiting the selection of programming languages, frameworks, and database systems.

Budget Constraints: The architectural design must adhere to budget limitations, influencing choices related to infrastructure, development tools, and third-party services.

Regulatory Compliance: The system must comply with relevant legal and regulatory requirements in the regions where it operates, affecting architectural decisions related to data storage, privacy, and security.

Timeline:The development timeline imposes constraints on the architectural design, requiring efficient and effective development and deployment processes.

Data Privacy: Adherence to data privacy regulations and policies must be maintained, influencing decisions related to data storage, encryption, and access controls.

Appendix:

The appendix section can include additional information that supplements the main document, such as detailed technical specifications, diagrams (e.g., system architecture diagrams, database schemas), and any other relevant documentation that provides a more in-depth understanding of the Easy Food Tracker system.

Specific items that may be included in the appendix:

Entity-Relationship Diagrams (ERD)
Use Case Diagrams
System Flowcharts
Database Schema
API Documentation
Mockups or Wireframes of User Interfaces
Any additional technical documentation or reference materials.

5.2 Easy Food Tracker Design Process

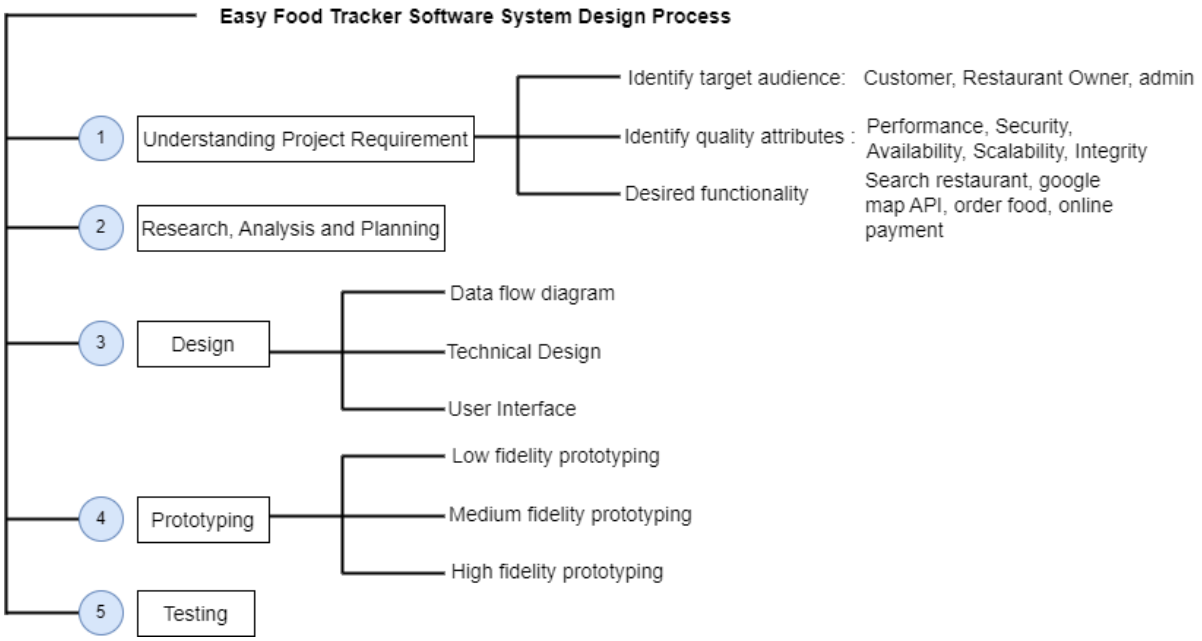

Figure-18: Easy Food Tracker Design Process

1. Requirements Gathering and Analysis

- **Identify target audience:** This involves understanding who will be using the system, such as tenants, house owners, and administrators.
- **Identify Quality Attributes:** This involves defining the desired qualities of the system, such as performance, security, maintainability, availability, and scalability.
- **Desired functionalities:** This involves specifying the features and functions that the system should have, such as house search, search food and restaurants, integrating with Google Maps API, and integrating with online payment system API.

2. Research, Analysis, and Planning

The research, analysis, and planning phase aims to dissect gathered requirements and subsequently formulate a comprehensive design plan.

3. Design

- **Data Flow Diagram**: This involves creating a visual representation of how data will flow through the system.
- **Technical Design:** Choose programming languages, frameworks, and databases based on requirements and expertise.
- **User Interface:** This involves designing the user interface of the system, which should be easy to use and navigate for all users.

4. **Prototyping**

- **Low Fidelity prototyping:** This involves creating a basic prototype of the system to get feedback from users.
- **Medium Fidelity prototyping:** This involves creating a more detailed prototype of the system to get further feedback from users.
- **High Fidelity prototyping:** This involves creating a close-to-final prototype of the system to get final feedback from users.

5. **Testing**

   This involves testing the system to ensure that it meets all of the requirements and that it is free of bugs.

5.3 Architecture Patterns

Architectural patterns, or styles, are predefined solutions to recurring design problems in software architecture. They offer abstract templates for organising a system's structure and components, incorporating best practices and design principles. By providing high-level design blueprints, these patterns facilitate the creation of scalable, maintainable, and robust software systems. They serve as reusable guidelines for architects and developers, streamlining the decision-making process in system design and promoting efficiency and consistency across projects.

5.3.1 Model-View-Controller (MVC) pattern

For core architecture we will use MVC pattern for integrity of the system. In future we may need to change only UI or business logic, so to avoid impacts of change on others module MVC pattern is best suited.

Model

Description: The Model component of the Easy Food Tracker system encapsulates the data and business logic related to food orders, menus, customer information, and other domain-specific entities.

Responsibilities

- Manages data storage, retrieval, and manipulation operations.
- Enforces business rules and validation logic.
- Notifies registered observers (such as Views) of changes to the data.

Example Functions

- Managing food items in the menu.
- Processing orders and updating order statuses.
- Managing customer accounts and preferences.

View

Description: The View component of the Easy Food Tracker system represents the user interface elements and is responsible for presenting data to the user in a readable and interactive format.

Responsibilities:

- Displays information to the user based on the data provided by the Model.
- Renders user interface elements such as menus, order forms, and delivery tracking screens.
- May include presentation logic for formatting and styling content.

Example Functions:

- Displaying available food items and prices.
- Presenting order confirmation screens and payment options.
- Showing order tracking details to customers.

Controller:

Description: The Controller component of the Easy Food Tracker system acts as an intermediary between the Model and the View, handling user input, processing actions, and updating the Model accordingly.

Responsibilities:

- Receives user input from the View and translates it into actions on the Model.
- Orchestrates the flow of control and business logic within the application.
- Updates the View with changes in the Model's data state.

Example Functions:

- Processing orders submitted by customers.
- Handling payment transactions and updating order statuses.
- Managing authentication and authorization for users.

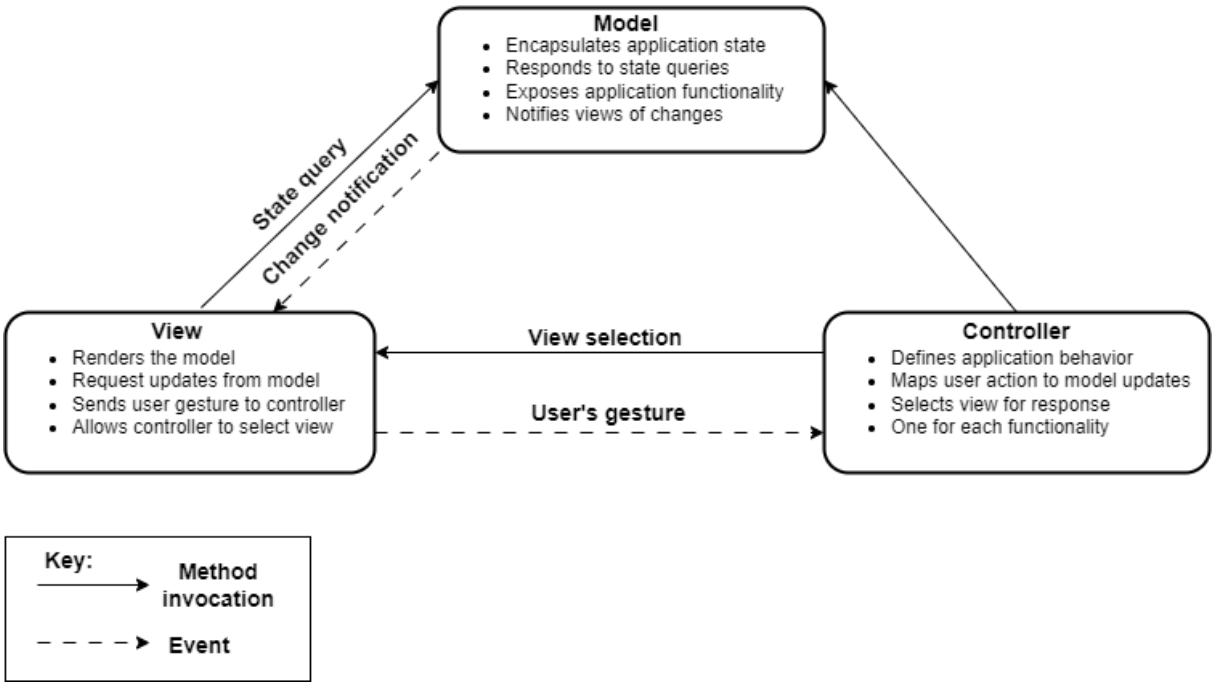

Figure- 19: MVC Pattern

5.3.2 Layered Pattern

The layered architectural pattern, also known as the layered architecture or n-tier architecture, is a specific type of architectural pattern that organises a software system into layers. Each layer has a specific responsibility, and the layers are stacked on top of each other. The layered pattern promotes modularity, separation of concerns, and ease of maintenance. The Layered Pattern involves organising the system into different layers, where each layer performs a specific set of functions.In the context of our application:

1. Presentation Layer

- Manages the user interface for both customers and restaurant owners.
- Contains interfaces for user interaction, restaurant management, and other related functionalities.

2. Application Layer

- Handles the business logic of the application.
- Includes components for order processing, user management, and notification handling.
- Orchestrates the flow of data and requests between the presentation and data layers.

3. Data Layer

- Deals with database interactions for storing and retrieving essential data.
- Manages specific data entities such as restaurant information, user details, and order data.

In the context of our "Easy Food Tracker" application, the layered architecture can be illustrated as follows:

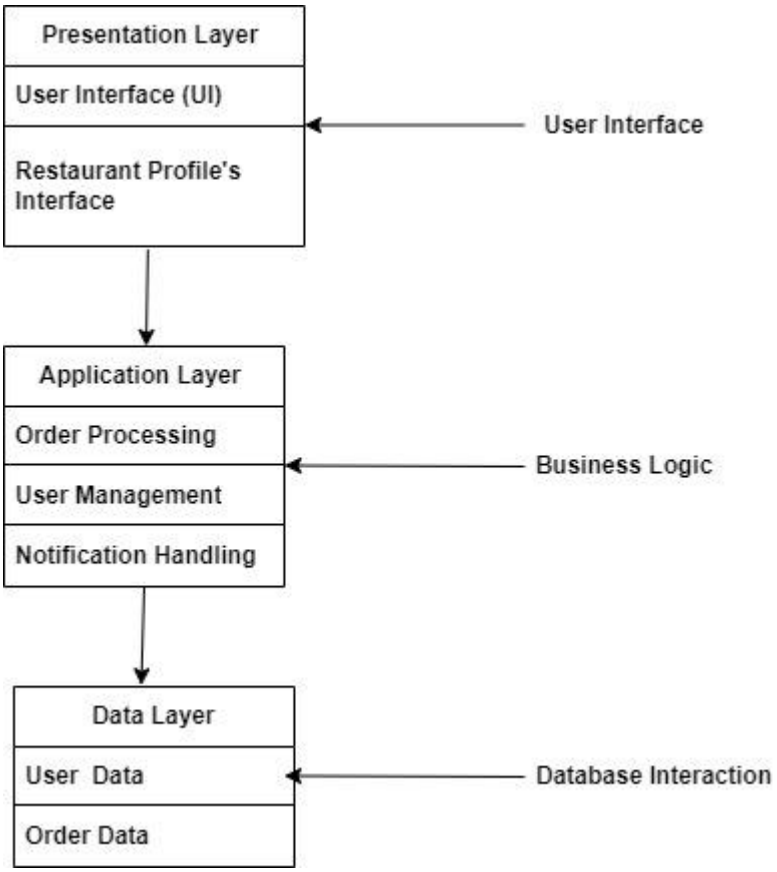

Figure-20: Layered Pattern

5.3.3 Broker Pattern

The Broker pattern is a design pattern that facilitates communication between components in a distributed system. It introduces a mediator component called the "broker" that centralizes communication and coordination between different parts of the system. In the context of the Easy Food Tracker system, the Broker pattern can be applied to handle the ordering process between customers and restaurants efficiently.
Components

Customer: Represents users who place orders through the Easy Food Tracker system.
Customer-Side Proxy: Acts as a proxy for the customer, providing an interface for interacting with the broker.
Broker: Serves as the mediator between customers and restaurants, coordinating order requests, and responses.
Bridge: Facilitates communication between the broker and the restaurant-side proxy.
Restaurant-Side Proxy: Acts as a proxy for the restaurant, providing an interface for interacting with the broker.
Restaurant: Represents the entities responsible for fulfilling food orders.

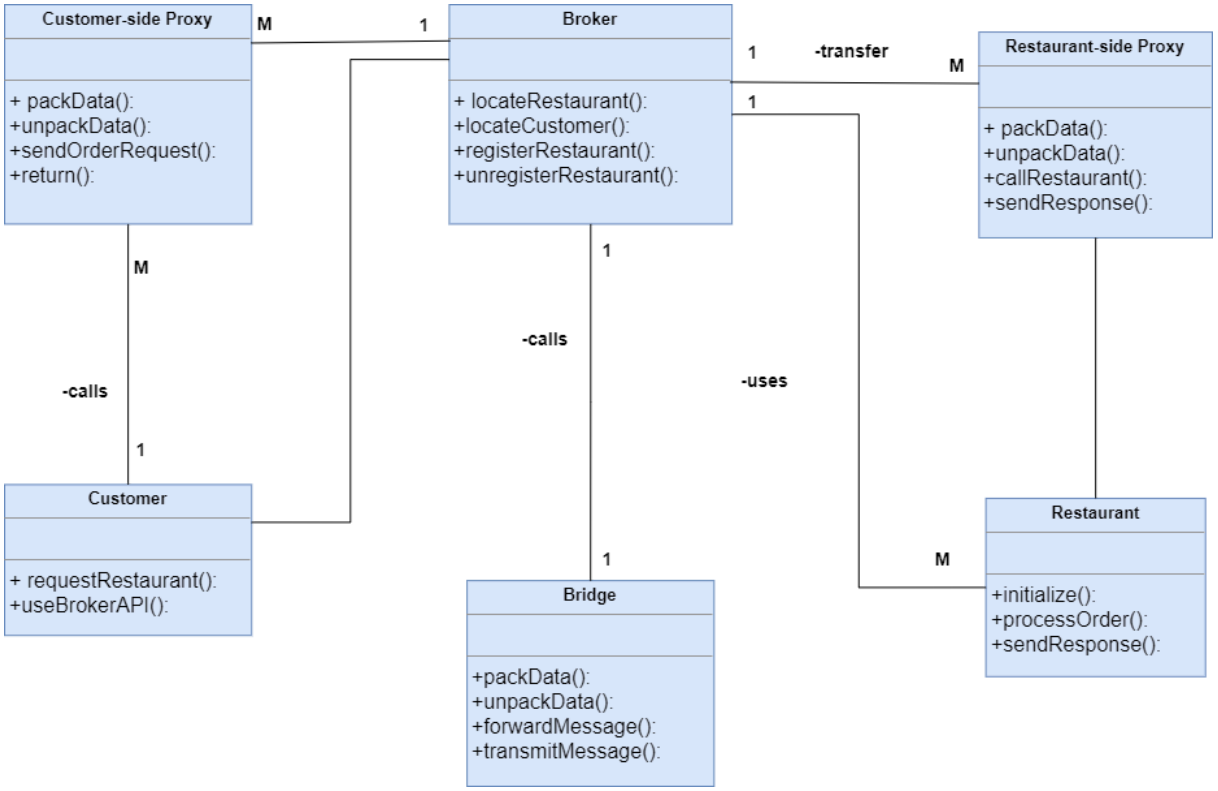

Figure-21: Broker Pattern

5.3.4 Client Server Pattern

The Client-Server pattern is a software architecture pattern that divides an application into two separate parts: clients and servers. In this pattern, clients request services or resources from servers, which provide the requested functionalities. The communication between clients and servers typically occurs over a network, such as the internet. The clients initiate requests, and the servers respond to these requests by providing the required data or performing the requested operations. The Client-Server pattern is widely used in distributed systems, where multiple clients interact with centralised servers to access shared resources or services. We will use for searching, access control, upload menu functionality in this project

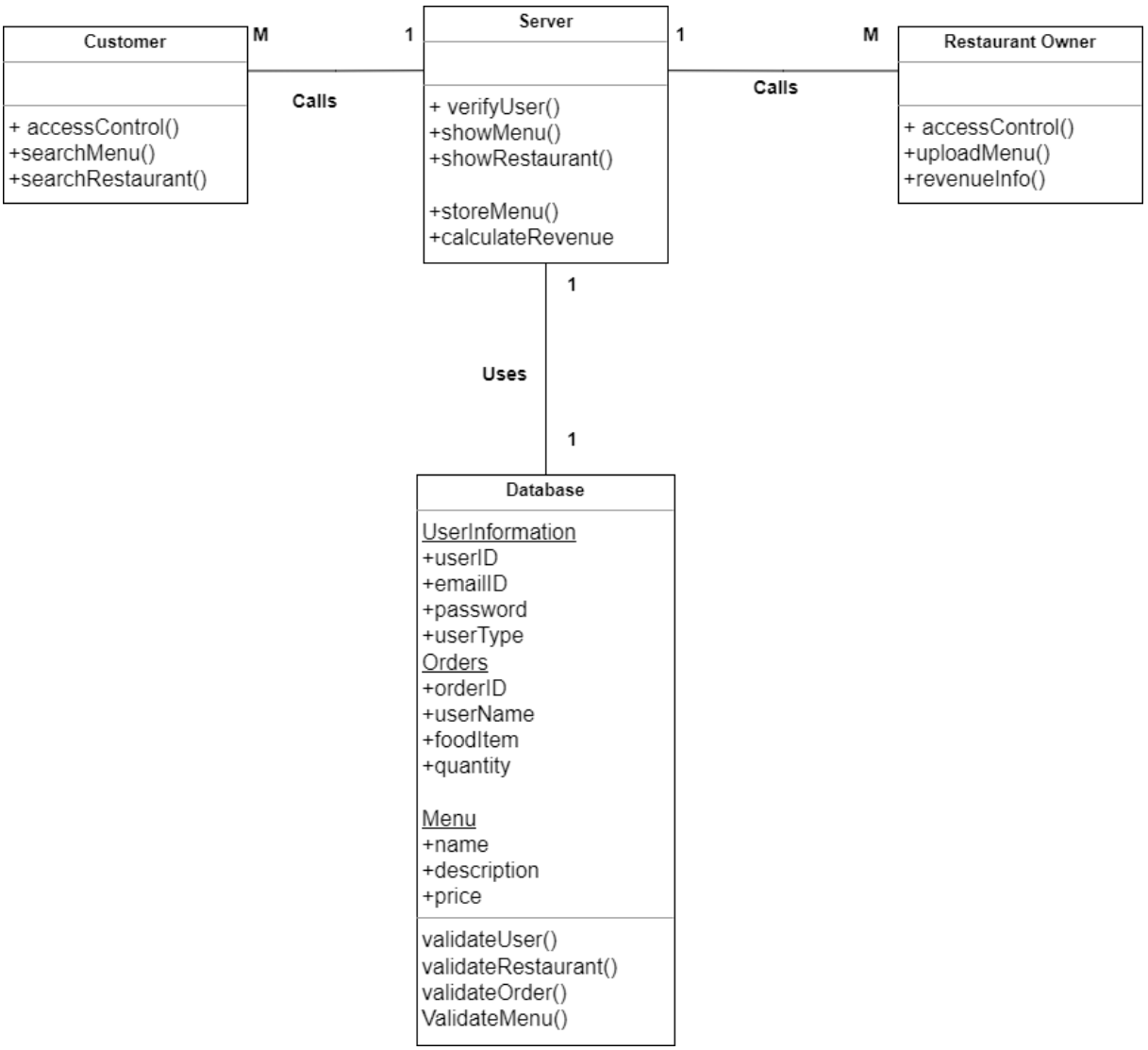

Figure-22: Client Server Pattern

Components

Client

We have two types of clients in this system, customer and Restaurant.

Customer: Methods of customer class are:

- **Access Control:** Allows customers to log in to the system securely using their credentials.
- **Search Food:** Enables customers to search for specific food items available in the system.
- **Search Restaurant:** Facilitates customers in finding restaurants based on their preferences and location.

Restaurant: Methods of restaurant are

- **Access Control:** Provides authentication for restaurant owners to access their accounts securely.
- **Upload Menu:** Allows restaurant owners to upload and manage their menus, including adding, editing, and removing menu items.
- **Revenue Info:** Provides information about the restaurant's revenue, including sales reports and analytics.

Server

Represents the central server component responsible for handling client requests and providing the necessary services.

Methods

- **Verify User:** Validates user credentials during the login process to ensure secure access to the system.
- **Show Menu:** Retrieves and displays the menu items available in the system to customers.
- **Show Restaurant:** Displays information about restaurants, including their location, menu, and contact details.
- **Store Menu:** Stores menu items uploaded by restaurant owners in the database.
- **Calculate Revenue:** Calculates and generates revenue reports based on order data stored in the database.

Database

- **User Information Table:** Stores user data, including user IDs, email addresses, passwords, and user types (customer or restaurant).
- **Orders Table:** Stores information about orders placed by customers, including order IDs, customer usernames, food items, and quantities.
- **Menu Table:** Stores menu items uploaded by restaurant owners, including item names, descriptions, and prices.

5.3.5 SOA Pattern (Service Oriented Pattern)

This pattern is best suited for integrating third party software so this pattern will be used on implementing online payment API and google map API.

Service-Oriented Architecture (SOA) allows for the creation of loosely coupled services that communicate with each other over a network. Each service represents a specific business function and can be developed, deployed, and scaled independently. In your case, you would have separate services for food ordering, payment processing, and possibly others.

Here's a simplified UML diagram representing the architectural components for mentioned functionality:

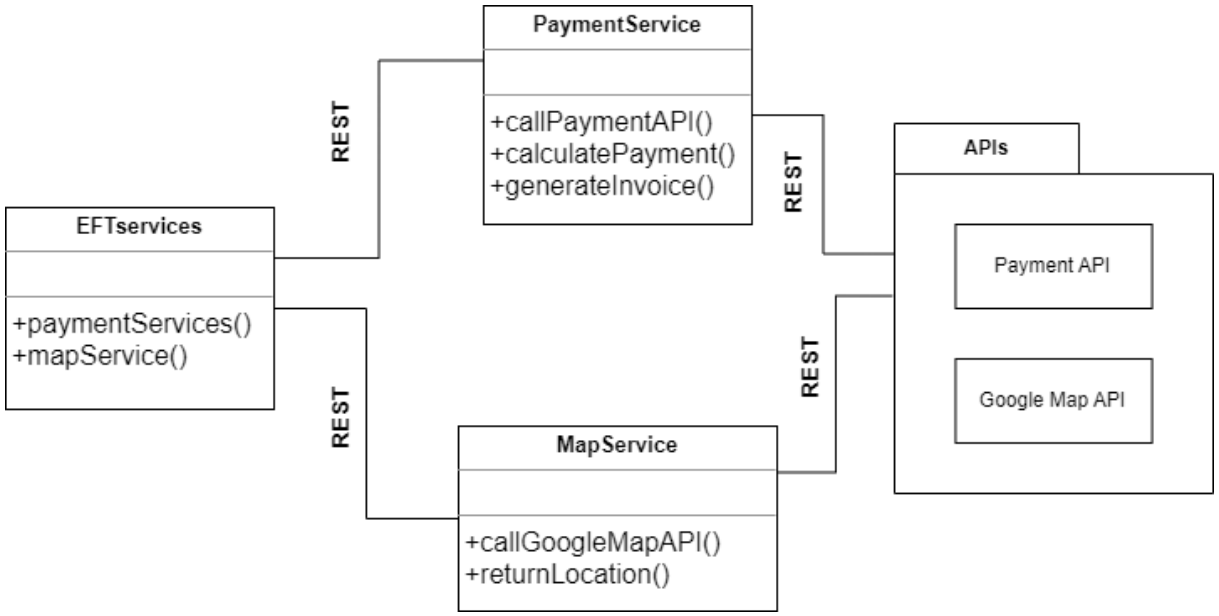

Figure-23: SOA pattern

In this diagram:

- **Food Ordering Service:** This service handles all functionalities related to food ordering such as managing menus, processing orders, etc.
- **Payment Service:** This service handles all functionalities related to payment processing including initiating payment requests, verifying transactions, etc.
- **Third Party APIs:** These are the APIs provided by external payment service providers like bKash, Nagad, Rocket, etc. Your system interacts with these APIs to facilitate the payment process.

Each service communicates with others via well-defined interfaces, often using RESTful APIs over HTTP(S). This allows for scalability, flexibility, and easy integration with other systems.