# IMPLEMENTING A WEB-BASED COMPUTERIZED RESTAURANT SYSTEM

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2013

By
Chin Loong Tan
School of Computer Science

# Contents

Word Count: 38220

# List of Tables

# List of Figures

# List of Listings



# List of Abbreviations

| Abbreviation | Description |
|---|---|
| **WCRS** | Web-based Computerized Restaurant System (see §1.2) |
| **UI** | User Interface |
| **HTML** | Hyper Text Mark-up Language |
| **CSS** | Cascading Style Sheet |
| **OO** | Object-oriented |
| **XP** | Extreme Programming (see §2.3.3) |
| **HTA** | Hierarchical Task Analysis (see §2.4.1) |
| **CRUD** | Create-Read-Update-Delete |
| **DAL** | Data Access Layer (see §4.2.1.3) |
| **BLL** | Business Logic Layer (see §4.2.1.3) |
| **PL** | Presentation Layer (see §4.2.1.3) |
| **ORM** | Object Relational Mapping (see §4.3.3.5) |
| **VS** | Microsoft Visual Studio (see §5.2.3) |
| **CI** | Continuous Integration (see §5.2.5) |
| **TFS** | Microsoft Team Foundation Services (see §5.2.5) |
| **EF** | Entity Framework (see §5.3.1.1) |
| **MVC** | Model-View-Controller pattern (see §5.3.3.1) |
| **MVVM** | Model-View-ViewModel pattern (see §5.3.3.2) |
| **Ajax** | Asynchronous JavaScript and XML (see §5.3.3.3) |

# Abstract

Managing restaurant operations is more challenging than it appears. A restaurant generally relies on paper-based system for manual information flow. However, such system soon meets its limitations. This is mainly because individuals in the restaurant have limited capability to handle massive information flow when the restaurant is at peak capacity. Consequently, many restaurants have adopted computerized restaurant systems to allow efficient operation management.

This project seeks to research, develop and experimentally implement and validate a computerized restaurant system to replace error prone and monotonous paper-based systems. The project proposed a *Web-based Computerized Restaurant System* (WCRS), to handle restaurant operations such as order handling, payment processing and inventory control. The two main research sub-domains investigated during the project are *Human-Computer Interaction* (HCI) and *Software Engineering* (SE); as well as the history behind restaurant management and information systems. The project demonstrated SE methodologies from the initial requirement gathering phase to the software testing and validation phase. Some noteworthy practises include establishing software architecture that could promote *separation of concern* and *reusability*, designing essential data structures and algorithms for restaurant data processing, applying presentation separation patterns such as *Model-View-Controller* and *Model-View-ViewModel* to decouple software components, and adopting web technology for real-time communication. The project also created intuitive and mobile friendly user interfaces by utilizing *Hierarchical Task Analysis* (for user interface and task-modelling) and adopting *Responsive Web Design* (for dynamic content presentation), both of which are directly aligned to the HCI methodologies.

A sequence of software prototypes were developed after extensive researches, designs, implementations and testing phases were conducted sequentially. The final prototype satisfied most of the high priority functional and non-functional requirements. Many subsequent features were integrated into the prototypes as the project evolved; these covered the most important restaurant operations. They were each tested and validated in order to demonstrate their capabilities to fulfil the project's objectives. All these processes are managed by agile methodology and involved *Continuous Integration* for software source control and test automation.

# Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

# Copyright

i. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the dissertation, for example graphs and tables ("Reproductions"), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/display.aspx?DocID=487), in any relevant Dissertation restriction declarations deposited in the University Library, The University Library's regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University's Guidance for the Presentation of Dissertations.

# Acknowledgement

The author would like to thanks to his supervisor, Dr. Richard Neville for providing continuous: mentoring, general support and helpful advice throughout the project. He has inspired the author in continuous learning and self-improvement during the preparation of this paper.

The author also wants to express gratitude to the School of Computer Science for providing rich academic resources to support the learning process. The author also feels graceful for receiving support of English language improvement from the University Language Centre (ULC).

Finally, the author would also likes to thank family and friends for their unwavering support and continuous encouragement.

Without these helps, the author would not have completed the report.

# Chapter 1. Introduction

This chapter will provides an overview of the project and report. It first introduces the reader to the project context and identifies the associated problems of managing a restaurant's operations. It then discusses the motivations that drove the project to conduct the researches and implement a software solution. Following this, it outlines the objectives that the project intended to achieve. Finally, it describes an overview of the report and the writing styles used in authoring the report.

## 1.1 Project Context

This project sets to design, build and test a *web-based computerized restaurant system*. Generally, a *computerized restaurant system* aims to solve restaurant problems with Information Technology (IT). A *computerized restaurant system* may be familiar to the reader considering that most restaurants are equipped with a basic cash drawer to process payment. In fact, the terminal used to process payment in restaurant is the origin of such system. This dates back to 1974, when William Brobeck and Associates built microprocessor-controlled cash register systems for McDonald's Restaurants [1] . In this system, tapping on associated item keys and numeric keys would place orders for a customer, it would then continue to calculate the bill when the operator by pressed the total button. This was followed by the invention of the first graphical point-of-sale (POS) system with touch screen support by Gene Mosher in 1978 [2]. "*We've eliminated the need for keys*," he said and stated that menu can be changed frequently without programming [3]. Over decades, the *computerized restaurant system* had evolved to cover more operational aspects in a restaurant. Some systems provide full coverage in supporting operations such as: inventory control, customer relationship management, table reservation and staff shift planning.

One of the driving forces behind the innovation of such a system is the attempt to replace the error prone and monotonous paper-based system. Commonly, the workflow of the system would start from waiters gathering orders from the customer on an order sheet, then passing this to kitchen chefs for meal preparation and finally collecting payment from the customer. This process can promote certain risks however, especially during peak period, they are not limited to the loss of order sheet [4], incorrect sequence of meal preparation [5], and added cost due to mistaken orders. Eventually, they may lead to low productivity and

customer dissatisfaction. Realizing these problems can affect business performance, so the restaurant owner quickly seeks a remedy by adopting IT into their business model.

## 1.2 Project Motivation

IT has become important tools to support business operations. Especially in the restaurant business, IT is playing increasingly important roles in resources administration, managing services, and assisting strategic decision making [6, 7]. Several analysis and research works have also suggested that competitive use of IT in a restaurant has significant advantages [4-6, 8-11]. In term of operational benefits, it can improve process efficiency, reduce possible human errors, and maximize use of resources [4, 5, 8-11]. Additionally, it also supports long term business goals, including achieving cost-effectiveness, maximizing profits, and the potential to penetrate wider markets [6, 7].

Motivated by the IT benefits, the project intended to utilize IT further to improve restaurant operation. To achieve this, the project will investigate the principles and techniques of the *Computer Science* (CS) domain, particularly *Software Engineering* (SE) and *Human-Computer Interaction* (HCI), to build a prototype of *computerized restaurant system*. The prototype are intended to be deployed as web application to support collaboration of various users, thus it will be referred to as a Web-based Computerized Restaurant System (WRCS) subsequently in this report. As this project progress, it seeks to answer eight interesting research questions:

**#1.** Could the computerized prototype system fully replace the paper-based system in restaurant?

**#2.** How could the project ensure that the development process is being properly managed?

**#3.** How does the project address the requirements of each stakeholder?

**#4.** What are the possible software design methodologies that are able to support the desired features and performance?

**#5.** How could the project align with the concept of "*separation of concerns*" to each stakeholders operation?

**#6.** What are the data structures and algorithms that could realize (and process) persistent of critical business information?

**#7.** What interface design technique could ease the tasks that the stakeholders require the system to perform?

**#8.** How would the prototype system support and utilize mobile device as medium to access its services?

From a CS perspective, these questions are worth investigating because their answers demonstrate the importance of SE and HCI principles to support the development of an IT system and creative usage of web technology to solve actual business problems – such as these.

## 1.3 Project Objectives

The general goal of this project is to develop and experimentally validate a *web-based computerized restaurant system*. This is supported by researching sufficient knowledge in the SE and HCI domains, and then applying this to the development of the system. To achieve this, the project will be underpinned by various stakeholder requirements – which link to the requirement engineering research domain in SE. The project also investigates practical software design methods (a sub-domain of SE) in order to build a high quality system. In addition, it also investigates the possibility of build a real-time information system to allow effective communication – again linking to the previous research domain of web development. Lastly, the project will also explore hierarchy task analysis and responsive web design [12] to specify, model, and develop appropriate graphical user interface (UI) behaviours – these are embedded in the HCI research domain.

The following objectives are defined in order to accomplish the project goal:

**i) Developing an efficient multi-tier system architecture**

The system required an architecture design that supported *separation of concern* and highly reusable implementation. It should allow full or part of the functionalities to be accessible by a range of devices. Hence, it should have presentation layer to facilitate an appropriate user interface, a business logic layer that hold common business functions and a data layer for managing database transaction. In-depth analysis and good design practise will be required to realize this implementation.

**ii) Developing efficient information sharing methodologies**

Sharing information should happen instantly since a delay in fulfilling order certainly reduces customer satisfaction. The project will need substantial investigation on how

real time communication should occur to allow effective collaboration among staff. Besides, data transfer for cross layer communication could also affect system performance and should be addressed by an optimized solution.

iii) **Designing a simple and intuitive user interface**

In any system, users will need to perform several tasks to achieve a high-level goal. A user interface should guide user through the tasks and help them to attain final goal. Since operations in a restaurant involve numerous tasks, extensive analysis should be performed into designing a UI that is simple and intuitive and addresses the users' goals (functional requirements) effectively.

iv) **Developing an efficient mobile friendly user interface**

The system should be easily accessed by different type of devices; so that it is portable and reusable. Considering that each mobile device may have a different screen resolution and size, the UI of the system should provide a responsive mechanism to offset these limitations with a dynamic UI layout and content resizing.

v) **Ensuring quality of the system through adequate software testing**

A significant amount of testing should be in-place to ensure that the prototype system is free from errors and bugs. In addition, the prototype's performance should be evaluated to analyse the effectiveness of the proposed methodology.

## 1.4 Report Overview

This report documented the concepts and solutions behind the development of the prototype software. It is broken down into seven major chapters (including this). This chapter has introduced the project in terms of its context, motivations and objectives. The remainder of the report is structured as following:

- Chapter 2 provides a background and outlines the literature review conducted related with computerized restaurant system and its early attempts. It then describes approaches that could improve this type of system.
- Chapter 3 explains the process of requirement gathering and project management techniques. The chapter formulates a set of requirements that the project has to satisfy in order to full its objectives, using of feature list, tabulated requirements, context model and use case model. Several techniques used to manage and monitor the project activities are also documented.

- Chapter 4 describes the process of transforming the requirements into conceptual solutions. The chapter introduces a high-level architecture vision of software and various diagramming techniques during system modelling. It also includes approaches that help modelling presentation and behaviours of the user interface.
- Chapter 5 focuses on the adoption of technologies to realize the design ideas developed previously. The chapter includes code usage and the algorithm from the implementation of each the software components. Finally, walkthroughs of the actual usage of the implemented system are shown.
- Chapter 6 is concerned with ensuring the implemented prototype software work as intended. It outlines testing techniques and tools used to validate and verify the software behaviours.
- Chapter 7 evaluates the approaches taken during this project and the achievement of the project. It also details possible future enhancements and a reflection of the skills attained during this project.

## 1.5 Writing Style

When the author references other sections of the report, section number are preceded by "§" and the first part of the subsequent number denotes the chapter so that "§4.3" is the third section Chapter 4. Figures, tables and listings follow the same convention; hence, Figure 4.6 is the sixth figure in Chapter 4.

The project uses square brackets, [] to encompass the reference number of the referencing materials. The list of references can be found after the Conclusion chapter, and right before the Appendix section of the report.

The Appendix sections include addition details and diagrams that are relevant to the main report. Each appendix section is prefixed by an Alphabetic character (e.g. Appendix A) to differentiate itself from the main report cross-referencing.

# Chapter 2. Background

This chapter describes the background of the system and looks at the project's nature in a wider context. It begins by understanding the characteristics of *computerized restaurant system* and looking at early attempts at such a system. The chapter then discusses how web technology fits into the development of such system. Next, the chapter investigates software process models that best meet the interests of the project. The last section explores the UI design techniques that could enhance user experience and accessibility of mobile devices.

## 2.1 Computerized Restaurant System

The term, *computerized restaurant system*, which is utilised throughout this project could be obscure to the reader. The general concept for this term is an integrated IT system that supervises, manages and facilitates the planning operations in restaurant. It is not odd that such a system is often associated with a *point-of-sales* (POS) system, a terminal that is use to process sales transactions – e.g. when the meal bill is paid. As stated in §1.1, this was derived from a simple electronic cash drawer which was utilised to collect payments, then it evolved to the basic POS system to the assist order phase and payment process. During 1990s, much investment in IT development focussed on integrating POS with back-office systems such as accounting and payroll systems [13]. Technology advances have allowed POS system, which previously use multiple software packages for different operational purposes, to evolve to fully integrated solution that automate restaurant operations [14]. The all-in-one system, including front-desk service control to back-office planning, is actually a computerized restaurant system. Some drivers behind such evolution, highlighted by [15] are:

- network connected system allows instantaneous connection to services and information;
- real time communication increasingly important to meet customer satisfaction;
- data warehouse and data mining emerge as important tools for decision making; and
- rapid technology changes have challenged the IT capabilities of restaurant stakeholders.

This clearly indicated a close relationship to IT advancement, which in turn introduced more possibilities to restaurant operation. For instances, networking technology allows remote data access in IT systems compared to what was previously restricted access on an embedded database previously. The restaurant industry quickly gains benefits from the phenomenon and relies IT solutions to remain competitive in the industry. Motivated by this scenario, some early efforts [4, 5, 8-11, 16, 17] attempted to address various concerns of restaurant operations. The following section will discuss the attempted works and their focus areas.

### 2.1.1 Early Attempts at Computerized Restaurant Systems

Early attempts at *computerized restaurant systems* aimed to improve the workflow of food ordering and kitchen preparation, and [4] proposed a *Process Management System* (PMR) that expand POS system to share order information in real-time. The system addressed the customer order management with timely tracking and validation. It demonstrated potential to reduce fraudulent orders and improve meal preparation efficiency. In addition, there are several research studies that focus on encouraging user interaction in a restaurant system, such as *Multi-touchable E-Restaurant Management System* [9] and *Mojo iCuisine* [11]. The proposed solutions allow self-ordering of food items by interacting with touch-screen interfaces. Both solutions consist of a touchable digital menu, which can be dynamically updated. Besides enhancing the dining experience, this approach also features flexibility over menu engineering and real time customer feedback.

Another important area of IT adoption in restaurant operation is inventory control. For example, the *RFID-based Sushi Management System* [10] presents an attempt to utilize *radio-frequency identification* (RFID) in conveyor-belt sushi restaurant to enhance operational efficiency, particularly in the area of inventory control, responsive replenishment, and food safety control. The developed system demonstrates promising potential in improving the quality of service in the food industry. Conversely, [8] presents a solution that is also adopted RFID technology but instead focuses on another operational aspect: customer relationship management. The study focuses on developing an intelligent menu recommender based on a customer RFID membership card. The approach of adapting a digital menu to customer preferences demonstrated practical customer-centric services and customer satisfaction improvement.

Finally, adopting mobile devices as part of the restaurant system has gained much attention lately. As noted by [16], mobile services are "*available at any time and any place*.". They demonstrate the great potential of more portable and accessible functions in a restaurant system. This is aligned to [5, 16, 17] mobile solutions for food ordering in restaurant. The main technologies used to realise their solution are web and wireless connectivity. Web technology provides loosely coupled and platform-independent ways of accessing application services [17] while wireless technology lifted the restrictions of close range operations. This approach is still applicable despite the recent evolution of mobile devices, from the *personal digital assistant* (PDA) to the smart phone.

All the studies above present various possibilities for improving restaurant operations with the aid of IT. However, there is still room for improvement. One of the limitations in their attempts is that the systems' UI often targeted particular platforms (embedded, desktop or mobile device). Thus, it might limit choice of medium to access the systems' services. Besides, their UI are not associated with the user tasks; hence, training might be required to operate such a system. Finally, it is also possible to incorporate every operational concern under a single solution in order to promote better collaboration among restaurant staff. The project aims to address these limitations and improvements by proposing a prototype software that adopts better UI design techniques and web technologies. Thus, an insight into web application development will be covered in the next section.

## 2.2 Web Application Development

World Wide Web (WWW) application, or *web application*, is any software application that is executed on the web [18]. Originally, the web functioning as an information medium [18, 19] and most of its content remained static. *Web application* evolved though, from statistic textual content with limited interactivity, to rich interfaces with dynamic content and responsive interaction, known as *Web 2.0* [20]. The role of the web has transformed from simple information publication to distributed enterprise-scale workflow systems.

*Web application* has proven that web technology could help in software development. Three basic elements of *WWW* that are found useful to software application development are highlighted by [20] as the following:

- *Uniform Resource Locators* (URLs), is a naming scheme to identify computer location, the requested resource in the file system and a protocol to communicate

with the resources [20]. The requested resource is not limited to file document, instead, developers may also use an URL to access a particular software service. Modern software applications enable communication across server boundaries by pin pointing the remote resources with the URL.

- *Hypertext Markup Language* (HTML), refers to a formatted document containing content, styling and structure to present the content of web pages [20]. The recent evolution to *HTML5* [1]promised even greater flexibility in visual presentation and responsive interaction. The potential of HTML is further enhanced by a combination of client side scripting (e.g. JavaScript) and dynamic content generation with server based programming language (e.g. ASP). In software development, designers could work on HTML presentation while back-end developers could focus on server side algorithms.

- *Hypertext Transfer Protocol* (HTTP), is a main communication protocol of the Web [20]. The protocol formulates the request data operation in GET method and sends data operation in POST operations. This allows modern *web application* to utilize a single URL for information request and data manipulation by specifying the desired operation in a request packet. The widely adopted protocol improves inter-operability of software application in fetching dynamic content. There are also increasing attempts to embed a web browser within a native software application to present information.

As discussed in §2.1, web technology emerges as one of the popular choices to develop a *computerized restaurant system*. One of the major advantages of *web application* is its accessibility. This connects to Gellersen & Gaedke's statement,

> *"Applications that use HTML-based front ends benefit from the pervasive distribution of Web browsers for universal, cross-platform access,"* [18].

In other words, *web application* could allow users to access remote resources (e.g. data and services) that are distributed across an enterprise network or the internet [5]. Besides, the nature of ubiquitous clients and centralized maintenance in *web application* has also enabled instantaneous deployment of software updates at minimal cost [5]. Compared to the

[1] A new standard for HTML with improve support for multimedia.

conventional patch and update approach, updating *web applications* could happen in real-time. The update also would have minimum impact on the client side as most changes are processed on server side. Both specified advantages make web technology an excellent candidate for inclusion in WCRS – in which remote data access and information sharing, is crucial to operations efficiency.

### 2.2.1 Designing Web Applications

The concerns of designing and developing *web application* are generally similar from a SE perspective as specified by [20]. The user will interact with the user interface, often a browser, to view and manipulate with the data managed at the server. The tight coupling between web page logics and contents result in poor maintainability and reusability. This is until the developer realized that *Model-View-Controller* (MVC) pattern, a well-known software pattern applied in SE, could be applied just as well to many *web applications* [20]. Models are classes containing data and business logics, the Views are web pages with formatting instructions of data, and the Controllers will facilitate communication between Views and Models for data presentation and manipulation. This approach achieves the SE principle, *separation of concerns*, by decoupling presentation logic from business logic. The MVC framework has recently become the dominant development framework and some object-oriented (OO) programming languages (e.g. *J2EE* [2] and *.NET* Framework[3]) would have their own MVC frameworks. The details of MVC pattern will be further discussed in §5.3.3.1.

*Web application* development could be different from general software application development, if following aspects were considered:

- It is concerned with creativity and interactivity of interface presentation [19];
- It is often content-oriented and required techniques to structure content [19];
- It needs to cater for a diverse environment as is exposed to various wider range of access device. [19]; and
- Its distributed architecture promotes unpredictable remote transactions.

[2] Java Platform Enterprise Edition, a computing platform maintained by Oracle.
[3] A software development framework developed by Microsoft.

The general approach used to develop web-based application is mainly ad hoc [19, 20] . It involves continuing patching of documents on a running web server. Such unmanaged development process lacks quality control and maintainability. This has leads to research of more disciplined approaches, which involve employing *Software Development Process* discussed in the next section.

## 2.3 Software Development Process

*Software Development Process* commonly comprises sequence of work activities, actions, and tasks that are undergone to create the final product. In the context of software development, the final product is a software application, a plug-in component, or a software service solution. However, software development processes are complex and unmanaged process could easily lead to catastrophic failure in delivering a usable system. While there are many factors that contribute to its complexity, the two main reasons described by [21] and [22] are:

1) Intellectual and creative processes rely on people's decisions and judgement; and
2) The environment may vary hence producing rapidly changing software requirements or strictly defined criteria.

Consequently, careful planning of development activities is required and this results in the adoption of software development process model. A s*oftware process model* is an abstract representation of interrelated activities in software development [22, 23]. It describes the general approaches in structuring activities and some techniques to produce deliverables. Selecting a suitable model for WRCS would cut down the development time and increase the quality of the output. Following sections describes several widely applied software process model in the software industry.

### 2.3.1 Waterfall Model

*Waterfall model* is a traditional software process model introduced by Royce [24]. It is a rigid and linear document driven methodology. This model is known as the *waterfall model* because it proceeded from one phase to another in a cascading order as shown in Figure 2.1. Before each phase can begin, each of the phases has a definite set of deliverables that must be approved by project sponsor, after the stakeholders have elicited them. However, the process of producing and approving these deliverables will incur significant cost. The

*waterfall model* often receives criticism on its inability to accommodate changes because the project freezes system specification upon deliverables sign-off. In a dynamic business environment, it is often difficult for user to state all requirements explicitly. The *waterfall model* lacks the ability to accommodate natural uncertainty and the changing need of users.

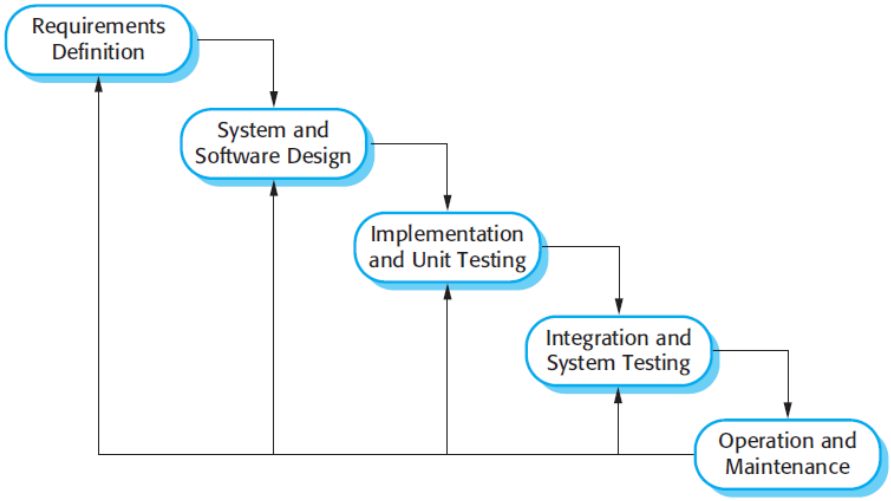



Figure 2.1: Overview of the waterfall model [22].

Another serious disadvantage of the *waterfall model* is that testing is often left to the end of the project. Errors and feedbacks obtained in later stages will require additional effort to resolve. Eventually, this will lead to a software product that not fit for user need. An enhanced variant of the *waterfall model* known as the *V-model* has improved to this issue. Figure 2.2 illustrates the quality assurance actions associated with deliverables of earlier phases in the *V-model*. Verification and validation approaches applied to earlier engineering work could significantly reduce errors found in later stages. However, the *V-model* does not explicitly describe actions taken in order to deal with errors found during testing. Nevertheless, *waterfall model* does show its strength when used in project where requirements are well understood and stable during development [22]. Documents produced during each phase provide traceability to address safety and legal issues when such concerns are critical to the user.

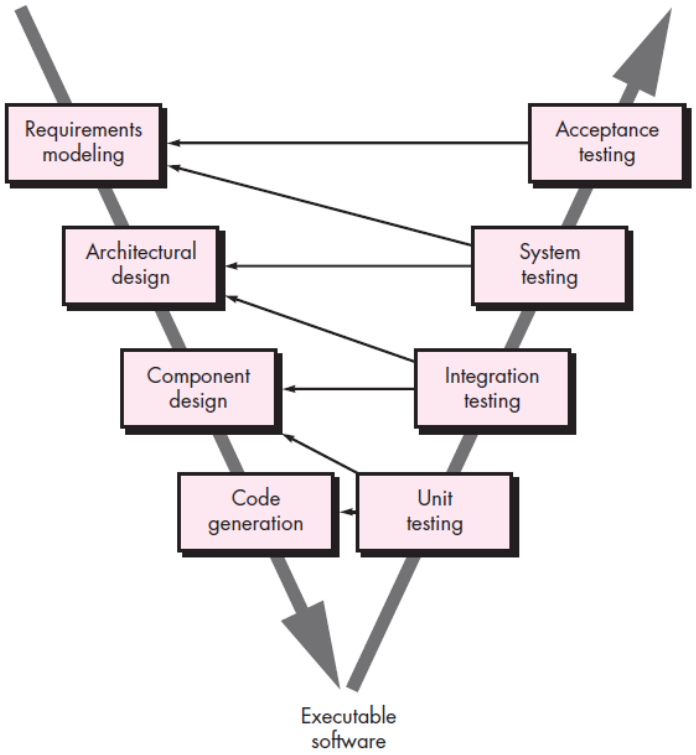



Figure 2.2: Overview of the V-model [23].

### 2.3.2 Evolutionary Model

*Evolutionary model* encapsulates two fundamental approaches: incremental and interactive; when addressing changing requirements. It organizes processes in a manner that enables the development of increasingly complete versions of software based on customer feedbacks through a series of iteration. The two fundamental types of *evolutionary model* that will be covered are *prototyping* and *spiral model***.**

The idea of *prototyping* is to enable users to interact and experiment with early prototypes which encapsulate a set of mutual understanding requirements. In [22], *prototypes* are described as an initial version of a software system used to demonstrate concepts, explore design options, in-depth problems and their possible solutions. Commonly, there are two types of prototype [22, 25] :

- Common prototypes aim to explore customer requirements through building an incrementally usable system. Prototypes with a minimum set of basic requirements are built and presented for the customer's evaluation. The prototypes evolves by the implementation of customer proposed features and changes until it's functionalities finally agreed by customer; and

- Throwaway prototypes aim to gather information and generate ideas on how system should be built. Commonly during project start up, the user may not fully understand their need and the developer may not share understanding on certain features. To clarify these uncertainties, a design prototype [25] which contain just enough details is built for evaluation. Once issues have been clarified, developers could then move on to an actual design and implementation.

Allowing requirements to be implemented rapidly is the key advantage to *prototyping*. However, this may lead to stakeholder confusion by mistreating what they see as final version of the system. Stakeholders should be well aware that some prototypes only serve as tools to gather requirements and may vary from the final product.

The *spiral model* is an evolutionary process model that combines the iterative nature of prototyping while retaining the systematic approaches of *waterfall model* [23]. Unlike the *waterfall model* in which it is hard to backtrack to previous phases once deliverables freeze, *spiral model* can be adopted throughout system development phases as shown in Figure 2.3. Explicit recognition of risk in the *spiral model* is the main difference compared with other process model [22]. However, understanding and mitigating the risks potentially reduces things that can go wrong. Iteration over entire phases of software process would be costly; hence, the *spiral model* is more suitable for large-scale projects, which contains high risk and requires well-structured approaches.

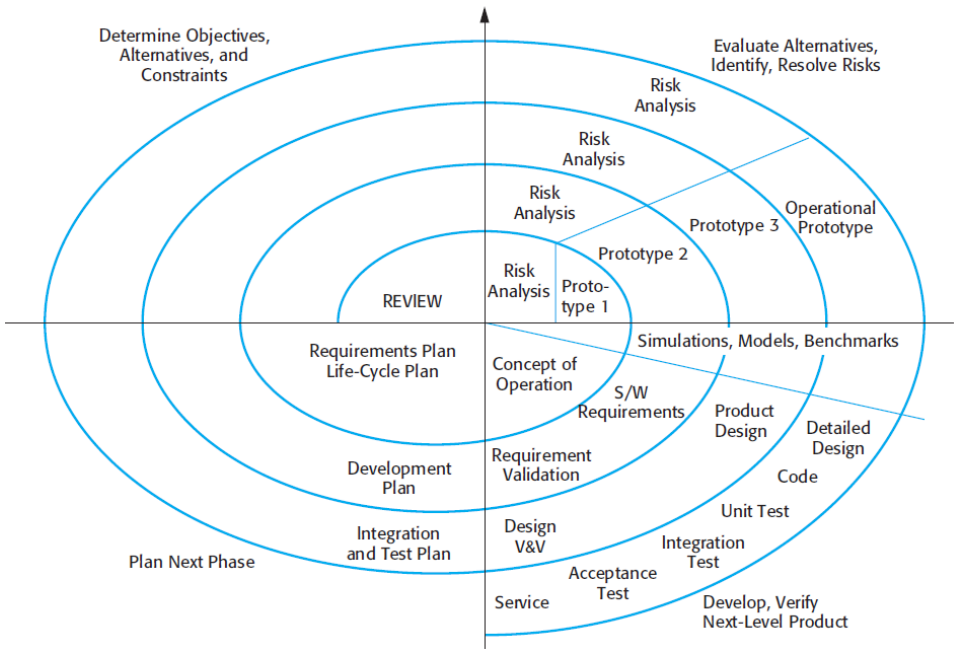



Figure 2.3: Overview of the spiral model [22].

### 2.3.3 Agile Development

Agile development processes have emerged to be the dominant software process model in recent years. Agile processes focuses on people, communication, working software, and responding to change as opposed to plan-driven models that have high process bureaucracy. These are best explained with *Agile Manifesto*[4]. Design and implementation are the central activities in agile development processes [22]. It would also be possible to incorporate requirements elicitation and testing into these activities, for instance, applying *test-driven development* (TDD). In TDD, the developer first writes test cases before writing actual implementations. This serves as the preliminary steps to clarify requirements and understanding for problem domains. Developers then code the actual implementations and execute tests to verify the implementations.

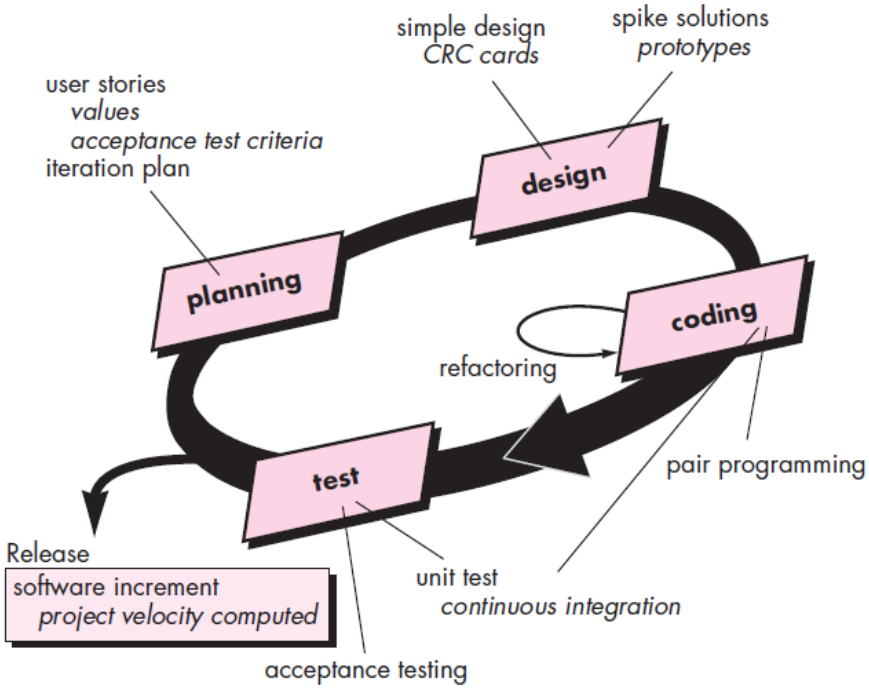



Figure 2.4: Overview of the Extreme Programming (XP) process [23].

*Extreme programming* (XP) has been widely known approach since the introduction of agile development concept. Figure 2.4 shows the XP processes and its practise during each phase. XP captures requirements in the form of customer stories or scenarios to determine the features required. In XP, continuous customer engagement in development is important for

[4] Declaration of four key values in iterative software development process, see Appendix A.

feedback and acceptance testing. XP favours small and frequent releases of software version like any other agile methods. Thus, design should only meet the current needs and expect refactoring when future improvement is required. XP recommended pair programming among developers because it can enable real time problem solving and quality assurance on solution applied [23]. XP is a lightweight process and fits well for small size projects. However, in a large-scale project where physical interaction among team members is difficult, it could be challenging for XP principles.

### 2.3.4 Comparison among Software Process Model

Table 2.1 presents a comparison of the three models discussed above based on several concerns that may affect WRCS development activities. These concerns, together with their explanations, are listed below:

- Requirement elicitation, presents approaches to gather requirements for system;
- Change management, reflects how changes will be handled throughout project;
- Validation, explains when testing will be done during project;
- Delivery discuss how quickly and often the software features will be delivered; and
- Design modelling covers the depth of design processes during modelling activity.

Based on the comparison, agile development clearly exhibited features that meet WCRS needs. WCRS will require segregation of user tasks and roles to model intuitive UI. Apparently, user stories of agile development fit better with these requirements. In addition, agile development has factored change management in the model. Its ability to cope with changes reduces the risk of delivering products that does not meet the objectives. The earlier the system is tested, the less effort will be spent on the error that may arise in end of the project. TDD of agile practise embraces this idea and encourages testing done before development. Connected to this, frequent delivery also implies that new enhancement have actually been verified in smaller scale. It reduces complexity by testing only parts that have been changed. Upfront design often leads to "*design paralysis*[5]*"* [26] when the developer tries to adopt concerns and considerations that may not be materialized in the future of project. This is why agile development prefers modelling just enough detail to support

[5] An anti-pattern that involves excessive up-front analysis and design but no actual action taken.

current need and refactor as required. Finally, lightweight agile process such as XP fits well into small-scale development, as in WRCS, which involves only single developer.

Table 2.1: Comparison among Software Process Model.

| Concerns | Waterfall Model | Evolutionary Model | Agile Development |
|---|---|---|---|
| **Requirement Elicitation** | Formal system specification | Requirement and prototyping | User stories or scenario |
| **Change Management** | Change is minimum or ignored. | Accept changes and will introduce changes at future incremental version | Accept changes and re-prioritize with current objective for future incremental version |
| **Validation** | Testing left to the end project | Testing done at the end of each iteration | Testing done in parallel with development in each iteration |
| **Delivery** | Slow and delivers as whole system at the end of project | Delivers increasingly complete software when requirements and design decisions are defined | Always delivers incremental working software with prioritized features |
| **Design modelling** | Excessive and lengthy | Limited but may grow quickly if too much emphasis on upfront design decision | Minimum with just enough details to meet current need |

## 2.4 User Interface Design

As discussed in §1.3, UI design, a sub domain of HCI, remains one of the main topics of the project. The UI concerns of this project are mainly designing UI that comfortably map user task to the interaction action, and more mobile friendly website design. The following sections will introduce two design approaches, *Task-based UI design*, and *mobile friendly website design*.

### 2.4.1 Introducing Task-Based UI Design

UI design has been increasing challenging for developers due to the rise of complexity and need for consistency [12, 27]. One possible factor that has contributed complexity is the users' mental model mismatch with the conceptual design of UI designers [28]. Users tend to focus on the task in hand rather than struggling to learn the features and procedures

necessary to activate services. Besides, it is also challenging to maintain consistency when more features are added to the UI [12]. Users potentially distracted from finding desired command if many possible selections are provided. Thus, new kind of user interactions that are self-explanatory and has a minimum learning curve is required.

Most of the current UI design focuses on the viewpoint of domain objects – which hold underlying data that will be presented in the UI [29, 30]. Users have to learn the navigational system, the naming of interaction objects, and follow a series of navigation on the domain-oriented menu to achieve particular goal [29]. For example, in a restaurant system, collecting orders for customer will travel following path: "table"; "order"; "create order"; "menu'; food categories"; and finally "add food item". The original intention of users has switched from simply submitting an order to interacting with several domain objects. There are additional layers of transformation from user intention to following the structure of system objects [29]. A *Task-based UI* employs a different approach by presenting available commands that meet user goals. Using the previous example, taking order function will present a more task-oriented command such as "take order" and arrange the interface to allow quick selection of food items.

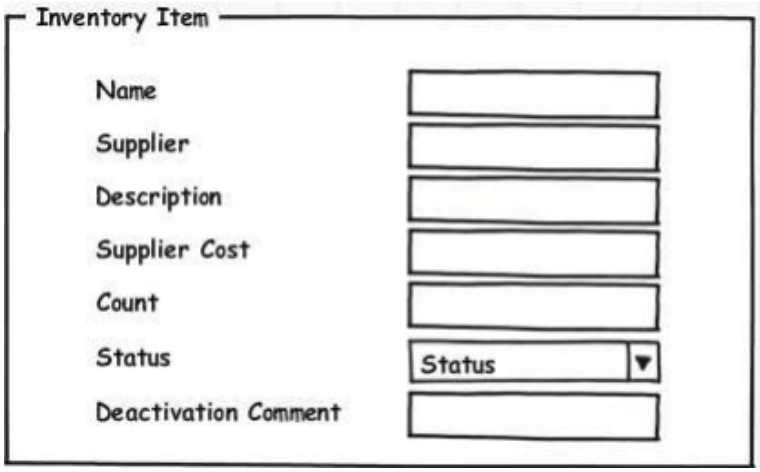



Figure 2.5: CRUD based UI design [31].

Another good example of *Task-based UI* approach is shown in the document by [31]. The author utilized a scenario where a user was trying to deactivate an inventory item in a typical software application. Figure 2.5 shows a CRUD based UI design where the user needs to navigate to exact instance of the item and lookup fields for changing status and deactivation comment. This required the user to browse through all the fields that not related to the original goal. Figure 2.6 shows a better approach by presenting the list of items in grid and

offer quick command to deactivate them, then a dialog to record the comment. Obviously, the latter approach is clearer to the user and presents only what the process needs.

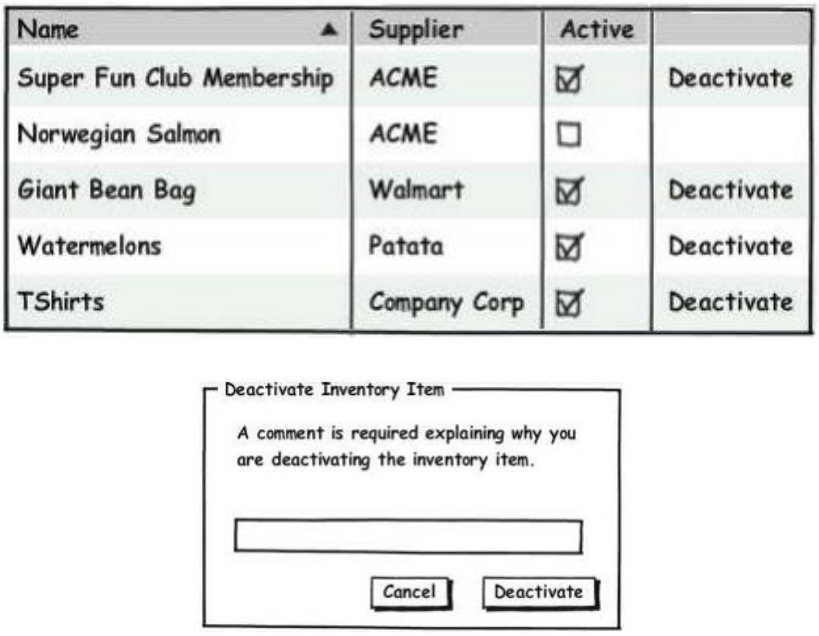



Figure 2.6: Task-Based UI design [31].

The experiment results [29] suggested that *task-based UIs* provide better performance in achieving user goals. It is also less cognitively demanding as it allows direct mapping of interface commands to user tasks [30].

Task analysis is an essential technique to model user tasks and system behaviour in *Task-based UI design* [32]. It helps developers to understand possible user actions and subsequently transfer them into UI designs. A few methods to perform task analysis and their objectives are adopted from [33] as below:

Table 2.2: List of Task Modelling Techniques and their objectives [33].

| **Methods** | **Objectives** |
|---|---|
| **Hierarchy Task Analysis (HTA)** | To inform designers about potential usability problems |
| **Goals, Operators, Methods, and Selection rules (GOMS)** | To evaluate human performance |
| **Task Knowledge Structure (TKS) or ConcurTaskTrees (CTT)** | To support design by providing a detailed task model describing task hierarchy, objects used, and knowledge structures, |
| **Adept approach** | To generate a prototype of a user interface |

This project will focus on HTA since it best meets interest of the project. HTA is the preferred method considering that its objectives meet the project's need in terms of resolving usability issues of a restaurant system. HTA is defined as:

> "*A hierarchical task analysis provides an understanding of the tasks users need to perform to achieve certain goals,"* [34].

In HTA, the high-level task (associated with user goal) is first identified, then it is subsequently broken down into more specific tasks to help users achieve their goals. An example of HTA model is shown in Figure 2.7. It depicts a high-level task as of making tea, then breaking it down to a series of preparation tasks to fulfil this goal. HTA is helpful in UI design because it allows designers to explore different possible paths that lead to completion of the task. Eventually, optimization of user interaction can be achieved.[34].

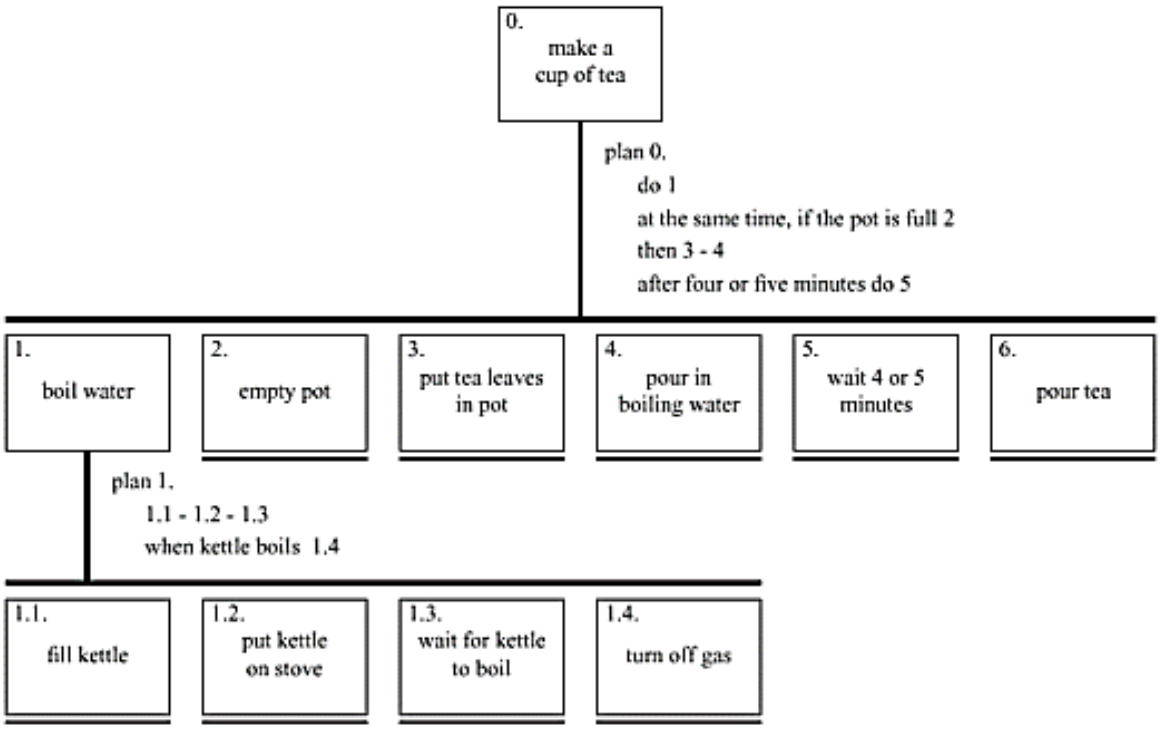



Figure 2.7: Hierarchical task analysis of tea making [35].

### 2.4.2 Designing Mobile Friendly Websites

The next concern of UI design is designing for mobile friendly websites. A *mobile friendly website* refers to a web UI design that has taken into consideration the limitation of mobile device, for example, screen resolution. Many websites failed to achieve this despite statistics [36, 37] clearly indicated mobile device has penetrated large populations and more people will use mobile devices as a primary medium to access online content. The common usability issue of these web applications is they do not scale well in mobile device [38].

Generally, there are two strategies to enhance web application support for mobile devices [38]. The first strategy is to separate mobile web sites from default web sites. If a web server detects access coming from a mobile client, it will redirect the client to the mobile pages (URL usually start with "m."). This approach has suffered from several criticisms and one of which is the additional effort to maintain several sites with duplicated content. Besides, redirecting is expensive because it introduces additional server request and overhead [40].

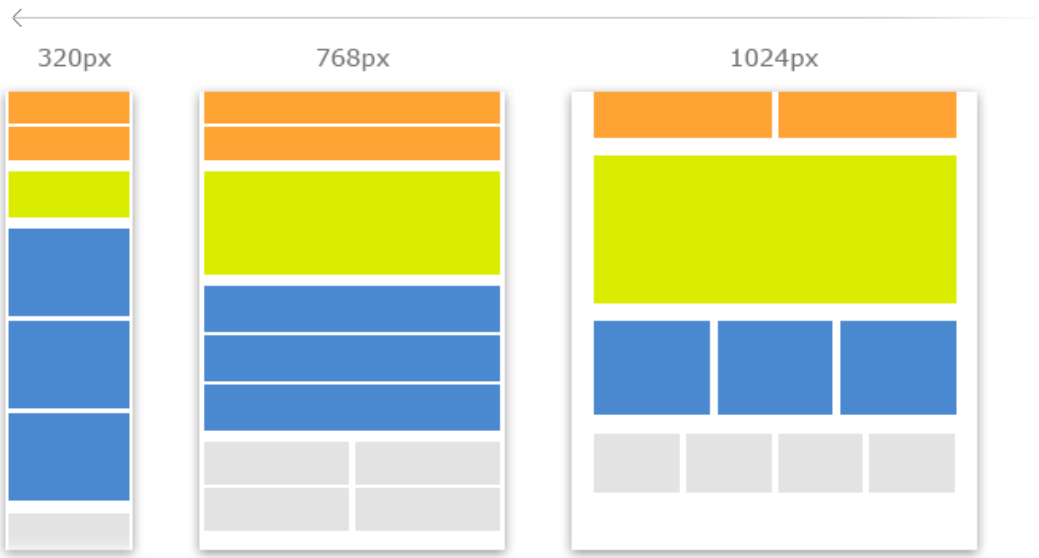



Figure 2.8: Example of responsive web design [41].

The next approach to this problem is *Responsive Web Design*. *Responsive Web Design* has gained momentum since the introduction of HTML5. With the help of the Cascading Style Sheet (CSS) – a HTML formatting framework, it is possible to design a web UI that will adapt to various screen resolution sizes. It scales down or hides certain UI contents to improve presentation on mobile browsers as shown in Figure 2.8. While this approach reduces significant efforts of maintaining redundant sites, it also increases the burden of

mobile web browser. As specified by [38], less visible content does not imply less content has been downloaded.

Recent web application developments lean towards the responsive design approach. This mainly because it has received increasing support from JavaScript web frameworks such as *Bootstrap* [42], *JQuery UI* [43] and *Less Framework* [44]. These frameworks adopt a fluent grid concept to layout contents on different screen resolutions. In fact, this project favours this approach because it requires smaller overheads to maintain separate sites and possible to achieve maximum reusability.

## 2.5 Concluding Remarks

This chapter has covered the background of the research domain and cast it in a wider context. It considered sub-topics that align to the project interest. By covering these topics, the reader should able to understand better the concepts and terminologies that will be used in later sections. However, these topics introduce their concepts from high-level viewpoint. The project will further investigate some of their subdomains, including techniques and tools, in other chapters.

The next chapter documents requirement gathering and project management techniques, marking the start of software development life cycle.

# Chapter 3. Requirement

This chapter investigates the necessary requirements of the project based on an understanding and analysis of the needs of system users. It first introduces the process of *Requirement Engineering* in WCRS. The techniques utilised to gather and analyse requirements are discussed which then leads to documents on functional and non-functional requirements. These requirements are then translated into informative models to assist decision-making and problem understanding in the *Requirement Modelling* phase. Finally, the chapter reviews several techniques to manage software development activities that could help to realise these requirements.

## 3.1 Requirement Engineering

The first step of the project is to understand user (and indeed stakeholders) requirements for building WCRS. Requirements of a system can be defined as descriptions of what services it could provide and the constraints on its operation [22]. These requirements directly address user needs in term of using the system to achieve their business operation – or process. In WCRS, business operations are rather obvious to those in the restaurant business – as specified in §1.1.

The process of understanding requirements, known as *Requirement Engineering*, involves a broad spectrum of processes, tasks and techniques to collect, analyse, document, and verify the requirements [22, 23]. *Requirements Engineering* is capable of bridging the gap between analysis and design of system as the output of such processes could serve as input to modelling design of systems. It starts with gathering user requirements, then performs analysis on the gathered requirement, and finally documents them in an appropriate format.

Following sections will cover the understanding and application of techniques that would help to formulate user requirement.

### 3.1.1 Requirement Elicitation

*Requirement elicitation* are concerns with identifying problems to be solved, what the user (or stakeholders) are trying to accomplish with the system, and how the system addresses the business need [23]. The process begins by understanding and analysing the restaurant

business problems (see Appendix B). Business analysis often reviews that people within an organization would have different needs (or opinions) and views concerned with the overall requirements of the system. They are stakeholders who either directly interact with or are indirectly affected by the system requirements. Hence, the focuses of *requirement elicitation* in WCRS are to analyse the stakeholders' roles and how the operations that they performing – affects the system; and hence must be specified in the requirements.

#### 3.1.1.1 Stakeholder Analysis

Stakeholders are of primary importance to any project due to enormous project resource that has been invested to know exactly what the user wants [45]. If stakeholders are approached earlier in the project, it is easier to communicate their requirement and work out their high priority concerns. The initial step to discover stakeholders requirements would be via *Stakeholder Analysis* [45]. *Stakeholder Analysis* views a system as "*a complex set of interacting elements which working together to satisfy needs or objectives*" [45]. The idea is to discover how, when and where stakeholders are involved in the process. As for WCRS, the different levels of stakeholders' involvement in the system can be viewed from a stakeholder analysis depicted in Figure 3.1.

The analysis process starts by identifying the first group of people who form part of the system (i.e. in a hierarchical manner) – namely the operators. In WCRS, they are referred to Waiter, Cashier, Host and Chef. They undertake specific tasks to achieve their operational goals and the system cannot operate without them. Following this, *stakeholder analysis* continues to discover the next group of stakeholders (in the hierarchy), usually beneficiary of the first group. The manager may be viewed as a direct functional beneficiary – who relies on staff to operate the restaurant. In additions, the manager needs to purchase ingredients for restaurant at the appropriate time, amounts, and price; hence, there is necessity to establish long-term relationships with supplier for stable material supply. Finally, the customer who is paying for services provided by restaurant is the another functional beneficiary of the system. In reality, it is possible for a single stakeholder to perform combined roles of other stakeholders. For instance, the manager sometime take over the role of cashier and host in certain restaurant.

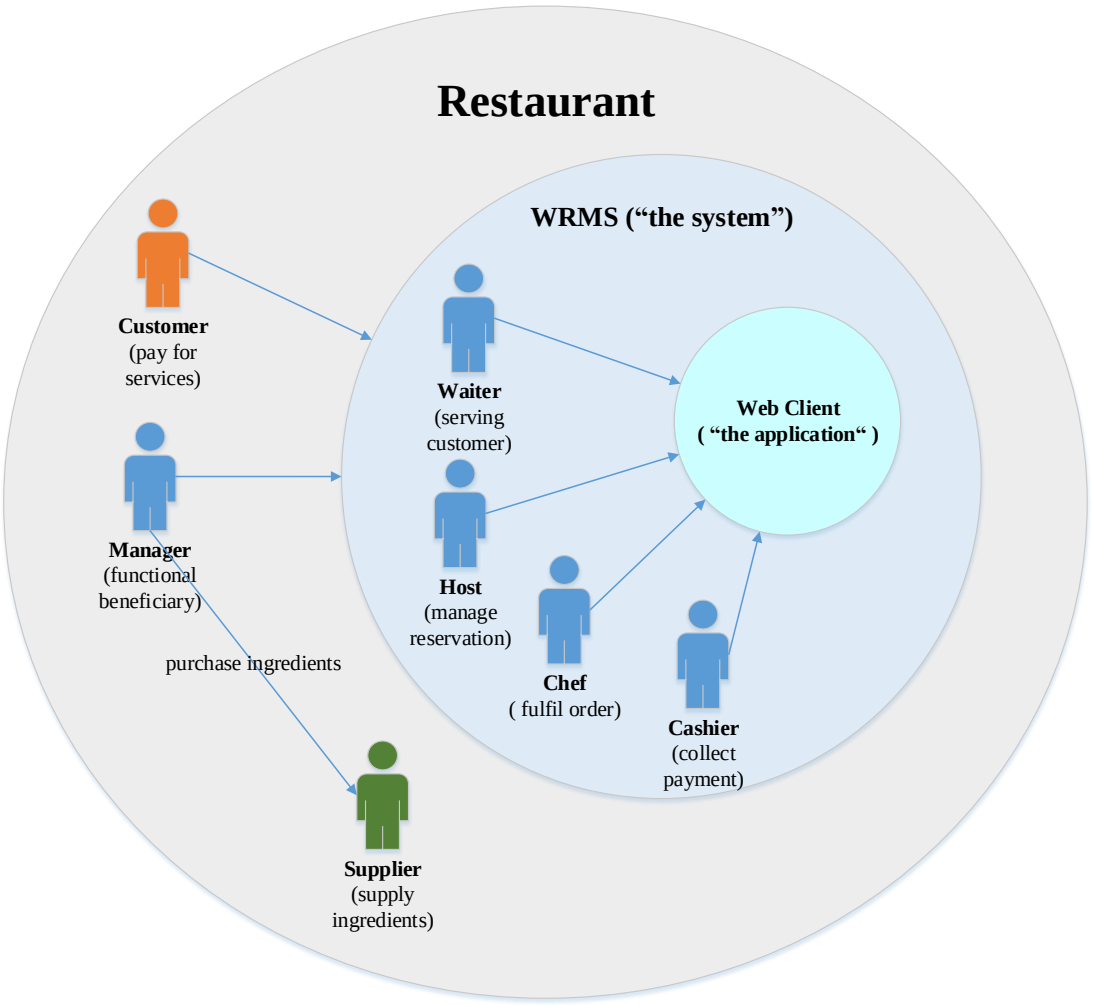



Figure 3.1: Shareholder Analysis of WCRS.

### 3.1.1.2 Identifying Stakeholder Operations

After the initial analysis of stakeholders as shown in Figure 3.1, the next step is to understand the responsibilities of stakeholders. Table 3.1 depicts an overview of stakeholder involved in the restaurants operations. It states the responsibilities and operations of each stakeholder. Although it shows clear segregation of Waiter, Cashier and Host roles, their positions often overlap in reality and can be referred to merely Waiter in general. However, precisely identify the operations involved in each distinct roles is a prerequisite for detailed task analysis in design phases. This table will serve as foundation for consideration of required system features to support stakeholder responsibility. The functions that support their operations usually are requirements of the system, except Customer and Supplier who do not directly interact with the system – they are considered external actors.

Table 3.1: Stakeholder and Roles in Restaurant Operation.

| Roles | Responsibilities | Operation Goals |
|---|---|---|
| **Waiter/Waitress** | Responsible for servicing customers at restaurant dining hall. They gather order information from customers and serve cooked food dishes to customer table. | i) Present menu to customer for order;<br>ii) Collect order information from customer;<br>iii) Submit orders to kitchen; and<br>iv) Answer customer enquiries about order status. |
| **Cashier** | Prepare bill and collect payment from customer. They ensure transactions occur and recorded correctly. | i) Calculate amount to be paid for order;<br>ii) Prepare bill for customer; and<br>iii) Collect payment and record transaction |
| **Host/Hostess (Maitre D')** | Ensure fine dining experience and smooth customer communication. They manage table allocation and reservation list. | i) Make reservation for table on request of customer<br>ii) Manage table allocation based on reservation list |
| **Chef** | Prepare food dishes based on customer order. They also create recipe and track usage of kitchen materials. | i) View order information;<br>ii) Update order status after cook dishes;<br>iii) Design and create recipe;<br>iv) Keep track of inventory status; and<br>v) Assist manager in ordering ingredient. |
| **Manager** | Oversee restaurant operations. They ensure sufficient resources such as food materials and staff to operate the restaurant. They also plan menu and promotion for restaurant. | i) Manage staff information;<br>ii) Order material from external suppliers;<br>iii) Generate and view report of restaurant operation; and<br>iv) Maintain menu information. |
| **Supplier** | Supply food materials to the restaurant. They receive purchase orders and deliver them periodically. | i) Receive purchase order for material from restaurant; and<br>ii) Supply ingredients to the restaurant. |

| | | |
|---|---|---|
| **Customer** | Visit restaurant for food and dining experience. They are main source of income for restaurant. | i) Ask for menu information;<br>ii) Order food item based on provided menu; and<br>iii) Make reservation for table. |

### 3.1.2 Requirement Analysis

It is important to gain insight in to what kind of system should be implemented and the level of change that may affect organization before determining which requirements are appropriate for a given system [25]. Hence, the steps taken after gathering the initial requirement involve performing an analysis on information obtained. Some of the basic techniques that could be applied during this process as discussed by [22] and are described in the following:

- Classification and organization of requirements, involves grouping related requirements and organizes into logical clusters or modules. Using model of system architecture is a common way to discover possible modules (sub-system) and associate related requirements to them.
- Prioritization and negotiations, involves prioritization requirements resolve conflicting requirements through negotiation with stakeholder. The concern is to achieve a set of agreed requirements that considered views of stakeholder involved.

Additionally, [25] stated that analysis process may also involves business perspective such as business process automation, business process improvement and business process reengineering. Business process may not too be the focus of this report, but possible improvement and automation should be considered as system requirements.

#### 3.1.2.1 Requirement Classification and Organization

As specified above, organizing requirements involves grouping requirement into related logical clusters to identify their relationship and dependency. Requirements may be vague at this stage because they are stated in the form of stakeholder's operational goals. Nevertheless, they could be translated into sets of required system functions. As for the complex software system, grouping related system functions often leads to a modularity view. Adopting modularity views in software architecture allows developers to have clear segregation of each subsystem concern and their relationships.

The result of grouping requirements based on system functions is shown in Table 3.2. Each grouping is given a module naming. Additionally, it relates the operators (stakeholders) operation goals specified in Table 3.1 to system functions. The last concern in the table is the dependencies of each module, serving as important criteria when prioritizing the requirements in the next section.

Table 3.2: Requirements Grouping for WCRS.

| Logical Grouping | System Functions | Operators | Dependencies |
|---|---|---|---|
| **Recipe Module** | i) Create and manage of recipe collection<br>ii) Consider required amount of materials to cook recipe<br>iii) Calculate recipe costing | Chef, Manager | Inventory Module |
| **Menu Module** | i) Create and manage menu plans for customer selection<br>ii) View menu item | Chef, Waiter, Manager | Recipe Module |
| **Order Module** | i) Submit order for customer<br>ii) Visualize of order items for kitchen interaction<br>iii) Manage order status | Waiter, Chef, Cashier | Menu, Table Module |
| **Payment Module** | i) Compute payment amount for order<br>ii) Print bills for payable order<br>iii) Collect payment and record transaction details | Cashier | Order, Table, Menu Module |
| **Table Module** | i) Create and manage reservation list for table<br>ii) Provide updated table status | Host | - |
| **Management Module** | i) Record and maintain employee information<br>ii) Control and maintain employee privileges level<br>iii) Product report for sales and orders<br>iv) Produce report for material usages | Manager | Order, Payment Module |
| **Inventory Module** | i) Record and trace material inventory level<br>ii) Plan material resupply and purchasing | Chef, Manager | - |

#### 3.1.2.2 Prioritizing Requirement

Prioritizing requirement involves ranking requirements by weighting the characteristic of requirements in terms of user needs and dependencies. High priority requirements should be addressed first because other requirements often depend on them. These requirements would also fulfil basic operation goals of stakeholders. Prioritizing requirement is an activity in XP according to the principle of incremental planning [22]. Requirements can be changed depending on the time available and their relative priority.

Based on module dependencies shown in Table 3.2, Recipe, Menu and Order modules are depended by most modules. Hence, they should have higher priority compare with other modules. However, each requirement maybe further broken down into sub-requirements for enhanced usability or improved workflow. They may not be essential to basic operations and thus would subject to lower priority. In brief, priority of requirements is concerned with balancing dependencies and user needs.

Table 3.3 depicts a method of prioritizing requirements in the format of features lists. It provides initial abstraction on the important requirements, followed by optional features. The list will require its priority to be further elaborated during requirement specification to achieve optimum ranking. This is described in the functional and non-functional requirement sections.

Table 3.3: Features List of WCRS.

| **Important Features** |
| --- |
| • Creation and management of recipe collection;<br>• Creation and management of menu;<br>• Submission and management of order;<br>• Mean to interact with pending orders in kitchen;<br>• Order processing and notification order status on cooking completion;<br>• Payment computation for order;<br>• Generation of bill and associated VAT;<br>• Mean to collect and store payment transaction details;<br>• Material inventory level monitoring;<br>• Recording and maintaining employee information;<br>• Employee login and privileges level control; and<br>• Reporting of sales and orders |

| **Optional Features** |
| --- |
| • Mean to attach and store recipe photo;<br>• Creation of composite menu item;<br>• Adjustment of Menu available time;<br>• Mean to attach remark to order;<br>• Mechanism for cancelation and changing order;<br>• Processing order for dine-in and take away order;<br>• Mean to view and search order history;<br>• Reporting on materials usage;<br>• Creation and management of reservation; and<br>• Materials resupply planning. |

### 3.1.3 Requirement Specification

*Requirement specification* aims to define requirements in clear and unambiguous language based on requirement identified during *requirement elicitation* and *requirement analysis*. The requirements are evolved over time and become more accurately reflect the needs of the stakeholders. The next steps are to document these requirements and treat them as ultimate references of domain knowledge and business rules. Documenting requirements can be challenging, and it is aligned to the fact that the software industry appeared to use the term "*requirement*" inconsistently [22]. Requirement could be a high-level abstract description of system provided services, or comprehensive formal definition of system functional units. Given the vast difference of stakeholder's perspectives and interests about the best way of specifying requirements, a further investigation into approaches of documenting requirements is necessary.

Requirement documents – known as *software requirement specifications* (SRS), contain important statements describing the software product to be built. The level of details may vary depending on the type of developing system. Safety critical and complex systems often require detailed description of constraints or essential domain knowledge. On the other hand, requirements for commercial software are often changing and become out-of-date quickly. It appears that *use case* and *story card* from agile development methods are more flexible in capturing business requirements. Based on [46] suggestion, the agile approaches in documenting requirement are:

1. **Focus on software, not documentation**. Create it only if it is essential to the work effort;
2. **Keep it simple.** Create the most minimalist version of each artefact and use simple tools such as index cards
3. **Proceed iteratively.** Start by identifying a high-level model and gather the details as the work proceeds; and
4. **Work as Team.** Close collaboration could improve communication thus reduce need for documentation.

Following on from this, the next steps are looking at two major categories of requirements: *functional requirements* and *non-functional requirements*.

#### 3.1.3.1 Functional Requirement (FR)

*Functional requirements* describe the features that the system has to provide. It often describes the expected features the user can utilize to perform their task. It may cover certain business processes and procedures that the software must follow and perform to achieve user goals – sometimes termed business logic. Thus, *FRs* are often associated with desired behaviour characteristics of developing software to produce the expected result. The primary *FRs* of the WCRS are to support or assist restaurant workers in the process or management of the food-ordering workflow. However, the granularity of the requirements may grow quickly as more understanding of requirements derived from Table 3.2 and Table 3.3. Functional requirements of WCRS are listed in Appendix C. Some of the scenarios of the requirements are defined clearer in use cases §3.2.2.

#### 3.1.3.2 Non-Functional Requirement (NFR)

*Non-functional requirements* refer to characteristics and constraints with which the system must comply. It concerned with quality aspects of software system and good user experience such as performance, security, availability. Unlike FR, NFR does not usually related to functionality that yields operation results directly. However, NFR do affect the experience or result quality when the user using the system. This is pointed out by [22] that:

"*NFR may affect the overall architecture of a system rather than the individual components.*"

The author further claimed that NFR might also generate FR given that the example of implementing Security features could introduce new system services. There are several concerns with respect to NFR in WCRS, especially usability – or usability analysis. Users should feel comfortably navigating the system using mobile devices. The user interface should provide clear indication of navigation path, and the menu, colour and layout should look consistent to the users. Security is also a major consideration when developing the system. Access control should be implemented to prevent destructive actions. The list of NFR is tabulated in Appendix D.

## 3.2 Requirement Modelling

In order to understand the system to be built, developers create model(s) to identify important ideas and decisions. There are two types of models that could be created during the software engineering process:

i) *Requirement models* – known as analysis models, represent requirements in terms of informational, functional and behaviour [23]. Requirement modelling explores analysis models to help developers understand the requirements in an intuitive way. These are important tools to prompt user feedback and "*provide a means for assessing quality once the software is built*" [23].

ii) *Design models* represent software characteristics that inform developers about how the software should be built.

The following sections will cover requirement models for WCRS while design models are discussed in the §4.3.

### 3.2.1 Context Model

During the initial phases of modelling requirements, developers must first understand the context where the system operates in a specific environment. The *context model* captures this perspective to allow decisions on project scope to be made by the stakeholders. The *context model* also shows system dependencies to its interacting environment. The external dependencies could be another automated system, manual processes, and functional peripheral or actors. They might produce data for system usage or consume the output of the system.

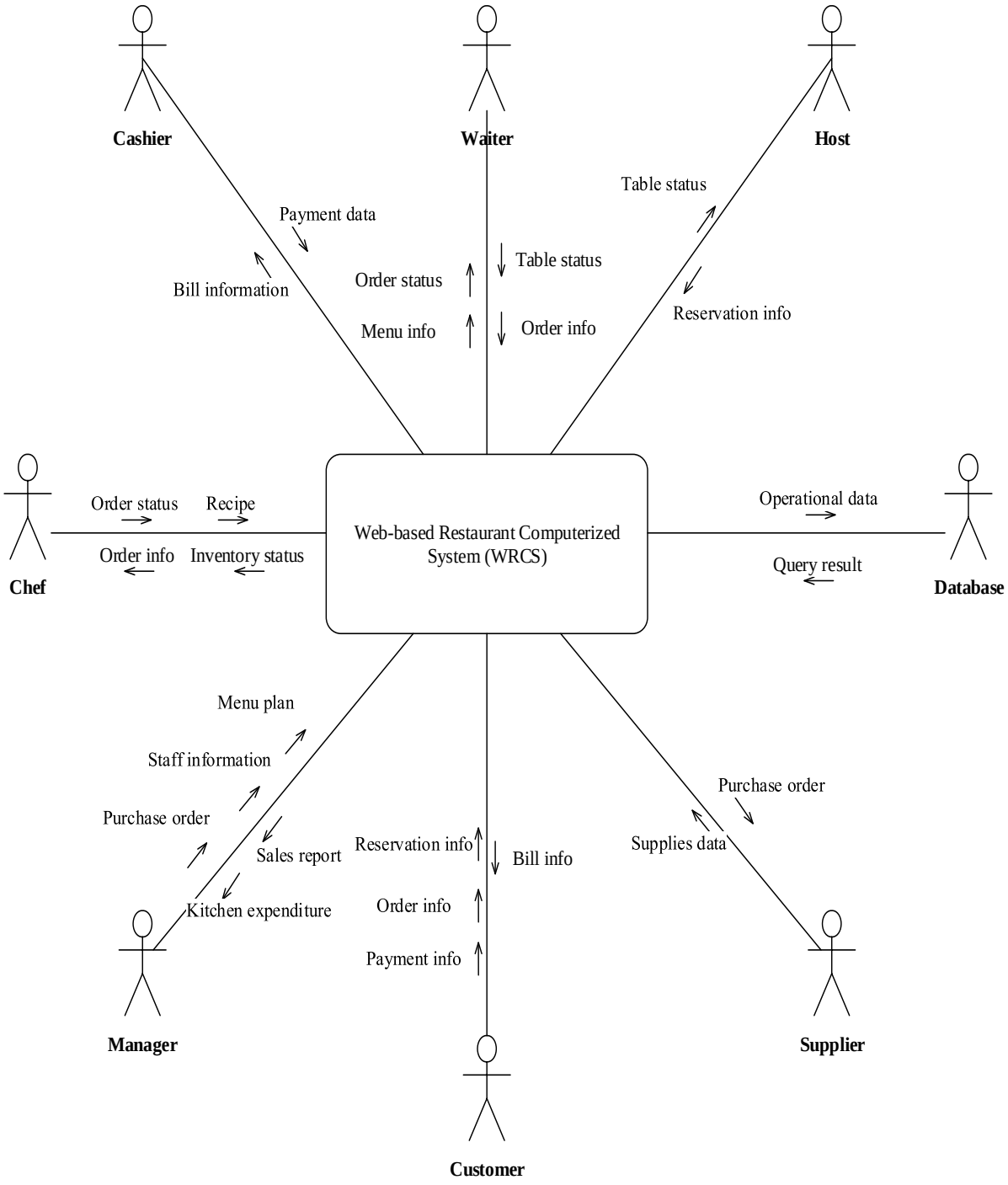



Figure 3.2: Context Diagram for WCRS.

*Context Diagram* (CD) which depicts overview of system working environment [47] is one of the techniques utilised to model the system context. It shows system interaction with external entities and is used to identify information and control flows among these interactions [48]. The two fundamental components [47] in CD are *actor* and *message*. *Actor* represents the external entities with which the system interacts. It refers to a particular user's

role who uses the system to perform task or external systems that are required by system to provide functionalities. *Message* encapsulates information flow and controls as part of connection between system and actors. Each connection is labelled with information or particular functions that flow between actors and system. These connections provide insight into possible events that the system must response if the message is a particular type of command. For instance, the system will send notification to kitchen when Waiter submitted order information. A typical message will contain two essential properties: *data content* and *arrival pattern*. *Data content* depicts the information that the message carry while *arrival pattern* describes the nature of message occurrence (e.g. periodically or asynchronous) and possible events that trigger its occurrence [47].

Figure 3.2 shows the CD for WCRS. The list of actors is identical to the stakeholders described in section 3.1.2.1 – as they are the primary operators of the system. The CD also included external actors that the system was interacting with, such as Database. The message flows show typical information used in a restaurant environment. An example detailed description of the message is listed in Table 3.4 and the remainder listed in Appendix E.

Table 3.4: Example description of Order Information message.

| **Message : Order Information** | |
|---|---|
| **Data Content** | Contains order item associated with selected recipe information and their quantities. |
| **Arrival Pattern** | Occurs asynchronously when waiter submit order based on customer request. |

Employing CD allows a clear understanding of system dependencies and system scope. Besides, explicit labelling information flow among system and external actors provides initial concept of data structure and process sequence. These ideas will then contribute to design analysis and system architecture.

### 3.2.2 Use Case Model

*Use case* modelling is one of the commonly applied modelling techniques in requirement modelling. Its primary use is to capture interactions between users with the system. Interactions that occur within a system could be user interactions such as input gesture, communication with external systems, or collaboration between components of the system [22]. Knowing users' preferred ways to interact with the system also allows developers to capture precise requirements and build a more usable system.

*Use cases* are simple descriptions of system features from the point of view of users. *Use cases* also capture scenarios of what the user could perform with the system and the expected response from system. Nevertheless, a *use case* is often used to capture *functional requirements* of the system [49] and generally are inappropriate for *non-functional requirement* [23]. *Use case modelling* involves two major artefacts: *use case diagram* and *use case description.*

*Use case diagram* is a simple representation of what functions the system allows actors to perform. It provides a high-level view of the relationship between actors and functionalities. Figure 3.3 illustrated the *use case diagram* for WCRS. Each *use case* is represented as an oval shape and each actor is represented as stick figure. An actor could provide input and receive output from associated use cases and these associations are depicted by line. The diagram shows all the actors that interact directly with the system features. This diagram inherited most of the information from Table 3.2.

Table 3.5: Example Use Case description for Manage recipe collection.

| **Use Case: Manage recipe collection** | |
|---|---|
| **Description** | User intended to create, update or delete recipes |
| **Actors** | Chef |
| **Scenario** | The use case starts when user elects to manage recipe collections from command menu. The system will present a collection of recipes. The user then can elect create update or delete actions. Create and Update recipe will prompt user to input the details. Delete action will remove the recipes from database. After selected action is completed, the user confirms to persist the result. |

*Use case diagram* shows only a small amount of detail about the functionalities and their flows. Therefore, *use case descriptions* are required to provide in-depth information to understand what is involved. [22] stated that they could be simple textual descriptions, a tabular and structured description, or a sequence diagram. Table 3.5 shows a simple tabular format that was employed to describe the use case of manage recipe collection. It abstracts the interaction details in the scenario and intentionally avoids premature decisions such as UI element design. The rest of the *use case descriptions* can be found in Appendix F.

In summary, modelling interactions in WCRS are fundamental steps to explore essential features required to achieve user objectives. This is particularly useful for grouping related features belonging to an actor role.

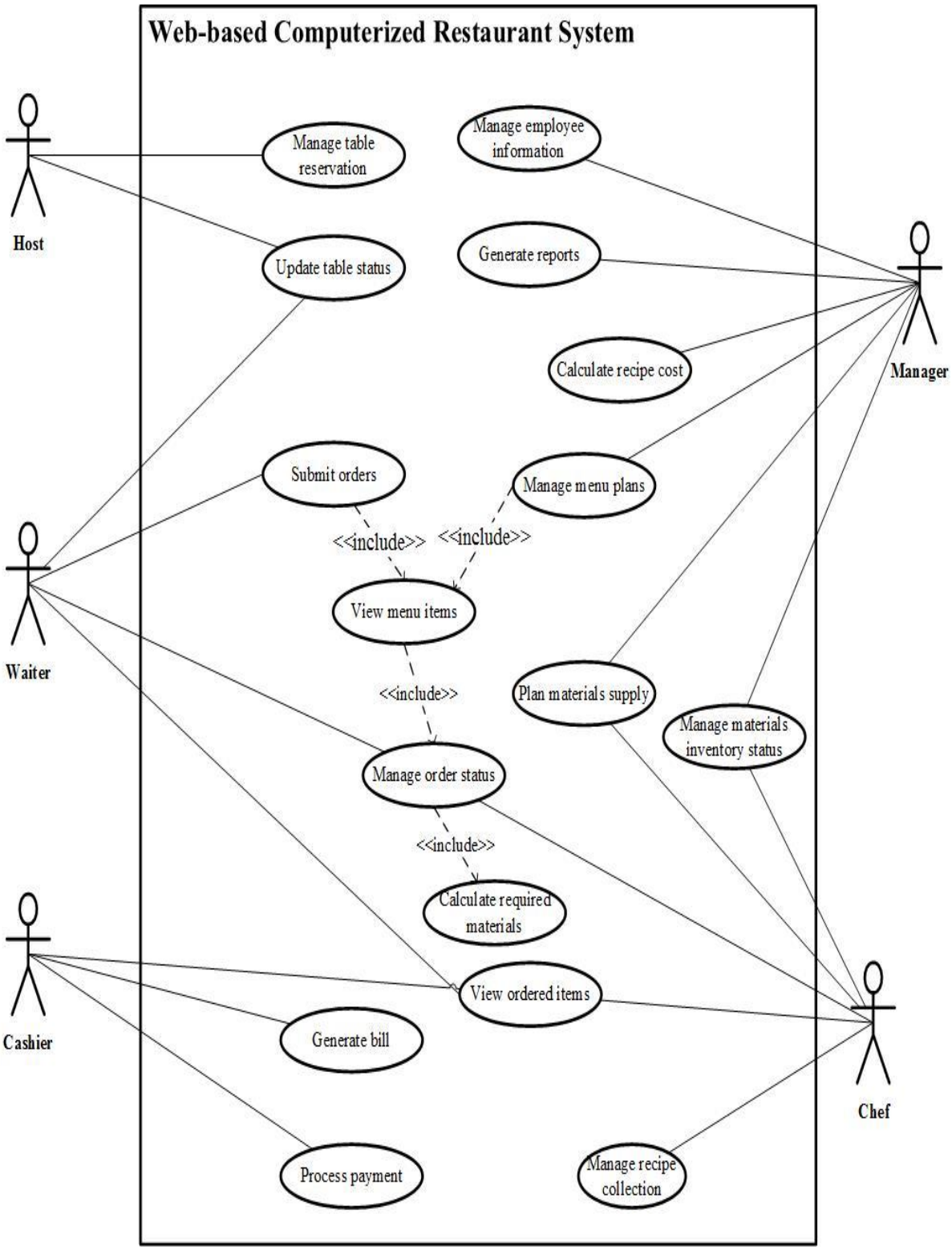



Figure 3.3: Use Case Diagram for WCRS.

## 3.3 Managing Software Development Activities

Based on the requirements discussed in previous sections, the project needs to formulate a project plan or sets of development activities derived from software engineering methodologies. As discussed in §2.3.4, the XP process (agile method) will be adopted as the software process model for the project. The important concern of adopting software process model is not strictly follow every principles and steps – but to use it as guiding principles. This section intended to cover some methods that could be used to improve traceability of project activities to requirements. It involves structuring tasks to be performed (i.e. software process model) and consideration of significant milestones.

### 3.3.1 Small Iteration or Releases

The development activities could be easily organized based on the system modules and features (refer §3.1.2.1). Each modules depicts a major release of the software features that are then verified against a sub-set of the requirements. It is important to note there are various dependencies between these modules; some lightweight tasks such as defining interfaces for other modules took precedence to enable development smoothness. At the end of each iteration, there are possible chances to re-evaluate requirements and adjust the plans – once each is: evaluated, tested, and validated against requirements. Each version of release prototype will be traceable to a FR or NFR (see Appendix M). This ensures high priority works are focused on first.

### 3.3.2 Project management: Gantt chart

A *Gantt chart* presents a series of tasks and their associated period in multiple bars spawning across the entire lifecycle of the project. Each task should have a clear timeframe and due dates as well as proper indications of its dependency. The chart includes a list of significant milestones. A milestone is defined as a "*significant event in the course of a project that is used to give visibility of progress in terms of achievement of predefined milestone goals*" [50]. It is used to measure and access the project success in meeting the deadline of original planning. The full chart of the project can be viewed at Appendix G.

### 3.3.3 Project management: Kanban

A *Kanban* [51] are useful project management techniques to provide timely feedback on the project progress – which is far more dynamic and interactive that a *Gantt chart*. It is a table which nominally drawn on a white board with several columns indicating the status of tasks before it is considered completed. Development activities often subject to various changes, for instances, a new techniques discovered during research would likely introduce new tasks. Besides, multiple related tasks may be performed simultaneously and sequential listing of these activities could be difficult. Hence, A *Kanban* is used to track high granularity tasks that can be derived from *Gantt chart* activities.

Figure 3.4 shows how tasks are managed using a *Kanban* during the project – the picture taken from the author's project logbook. The sticky notes with task information are moved to the left as they progress through: *draft*, *check* and *done* phases. They are removed from the project logbook and achieved when no longer required.

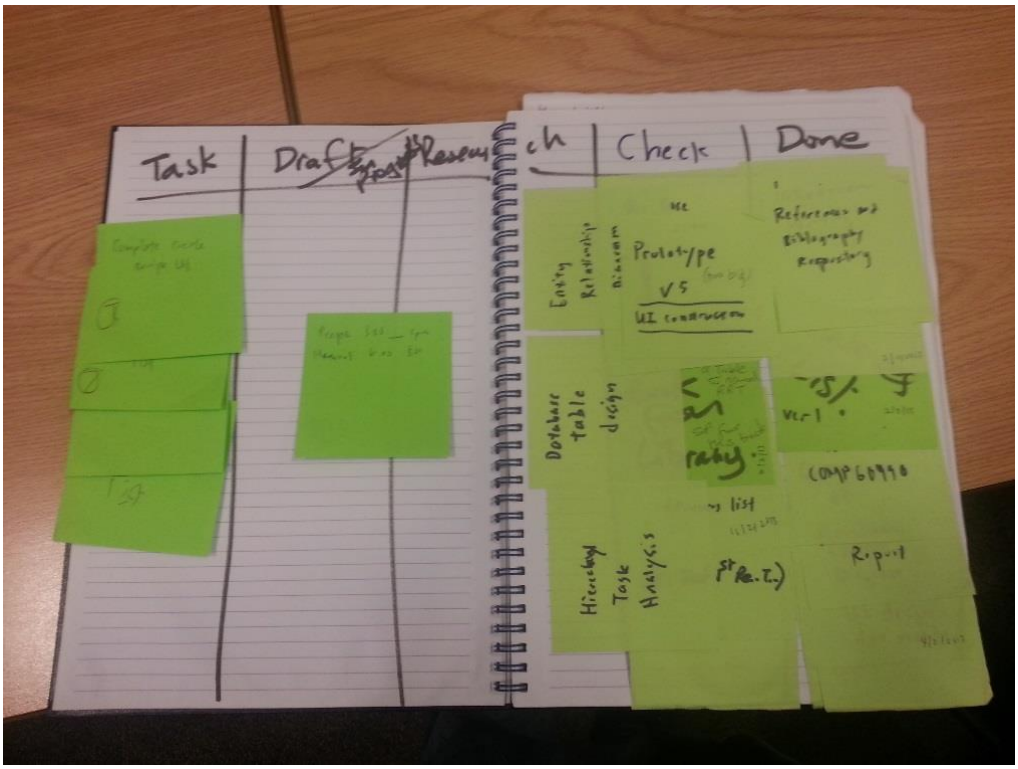



Figure 3.4: Kanban in project logbook.

## 3.4 Concluding Remarks

This chapter covered requirement: theory, understanding and software development methods that could be used to manage requirements – and as stated some were utilised. Based on the identified requirements, the author can estimate the scope of the system and formulate an expectation of the final product. The requirements once analysed show that the system required considerable effort and time to be a fully functional artefact and to meet all requirements. Thus, the high priority requirements were given focus and designed, implemented and tested first whereas low priority requirement – those that were not fulfilled due to time constraint will be discussed in the conclusion chapter. Following this, the chapter also discussed the theory behind managing software development activities and project planning which are key considerations to enable requirements to be compiled and then built. Usage of effective methods has helped the developer to achieve maximum output from the development activities.

The next chapter covers the design concepts and techniques of the project. It describes the process of solving requirements with logical and rational thoughts.

# Chapter 4. Design

Once the requirements of the project are established, the design phase will be followed. The design phase is intended to transform the requirements into conceptual solutions that could set a baseline for software implementation. This chapter intends to identify the design needs, investigate the relevant techniques and propose design solutions. It starts by identifying the design principles of agile development and taking them into heart of design activity. The project will then establish a high-level vision of the developing system through architectural design. It moves on to system modelling to create design models to understand characteristics and constraints of the system. Finally, the appropriate user interface design techniques are discussed.

## 4.1 Software Design

Software design comprises a set of principles, concepts and practises to build high quality system. It is intended to form a solution by appropriate consideration of requirements and technical issues [23]. Software design can be defined as:

> *"Software design is a systematic, intelligent process in which designers generate, evaluate, and specify concepts for a software system whose structure and function achieve clients' objectives or users' needs while satisfying a specified set of constraints,"* [52].

Further, [53] gives another view that describes the design space as focused on attaining the stakeholders goals by adapting inner environments (means) to the outer environments (tasks). The outer environments refers to requirements, goals and need; while inner requirements is the set of software, languages, components and tools used to build software (see Figure 4.1).

Software design may have broad spectrum of meanings and objectives based definitions above. Nevertheless, the primary goal of the design process in WCRS is to develop concepts and ideas that could answers our research questions while satisfying project requirements. There are four key considerations when performing design activities:

- Manage problem complexity through *separation of concerns*;
- Produce abstract representation of design decision;
- Provide unambiguous meaning to the concepts and terms used in the design models; and

- Establish guided paths to achieve specific end-user task.

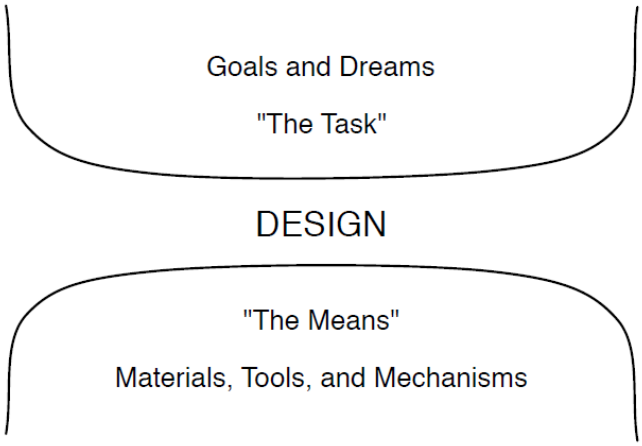



Figure 4.1: The continuing place of design, [53].

### 4.1.1 Design Process in Agile Development

Design process in traditional software development follows a series of planned design activities or steps, or phases which produces design models that become guidelines to the developers – for the implementation phase. However, this can conflict with agile methods, as if considering an interesting fact:

> *"...agile practitioners believe that design is not only highly iterative, but emergent, and models often lie. Thus, only coding, running tests, and refactoring the code reveal the truth about a design,"* [53].

In order to understand this conflict, [54] explained two styles of design in software development, namely: i) *planned design*; and ii) *evolutionary design*.

*Evolutionary design* means the design for a particular system grows as the system is being developed. However, evolutionary design is a disaster in common usages because [54]:

- Aggregate of ad-hoc tactical decisions lead to code base that hard to change;
- It leads to poor design when ability to make changes deteriorates; and
- Bugs become exponentially expensive to fix.

In contrast, *planned design* is closer to the engineering metaphor – where designer produces blueprint that consists of fundamental rules and structures to build the software. This design

style had been applied widely in early software development, yet it has several drawbacks [54]:

- Impossible to think through all the issues and new questions that emerge during programming;
- Technology changes rapidly and initial design concept may not match with the latest tools and materials; and
- It deals poorly with unforeseen changes of requirements.

The conflict of *planned design* and *evolutionary design* is widely known as "*code is my design*" versus "*big design upfront*" [53]. Nevertheless, both design styles may not be ideal to software development due to their problems mentioned above. This was until the emerge of Agile methods such as XP – that enabled *evolutionary design* by adopting set of principles to address changes and effects of changes in design. [54] pointed out that the core of enabling principles of XP are testing and *continuous integration*. Testing ensures that design decisions and changes are safety verified while *continuous integration* is essential to keep team in sync and not worry about new changes will break existing system. In addition to this, [46] described that models reach their point of maximal value when they are barely good enough, as shown in Figure 4.2. This again argues that design does not necessary needs every details been planned.

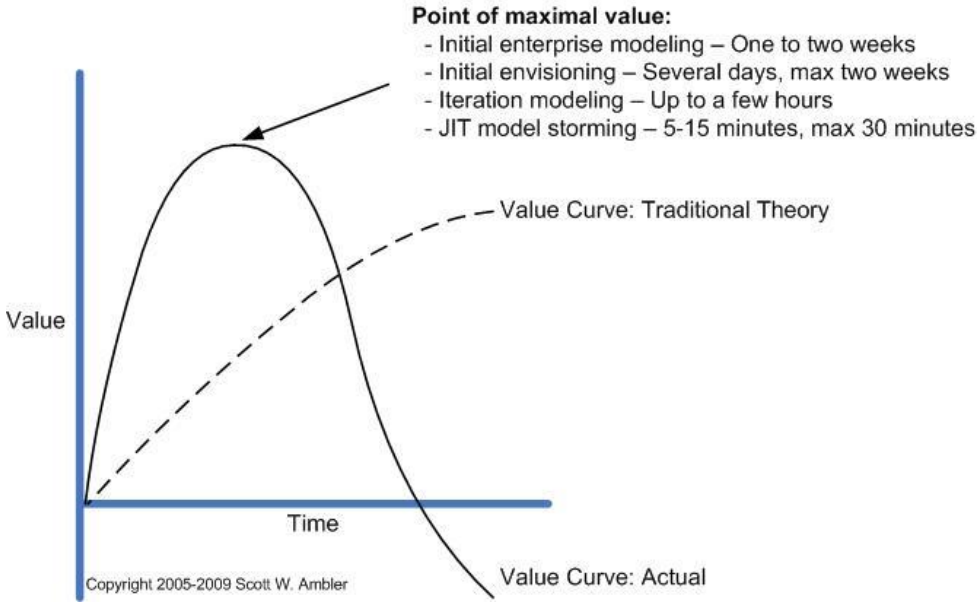



Figure 4.2: The value of modelling [46].

Design process in WCRS attempts to embrace changes with agile methods while encourage important design decisions that could be done with planned modelling. It needs to be aligned to several principles of *Agile Modelling* introduced by [46]. They are described as following:

- *Model with a purpose.* It is important to know the purposes and for which stakeholder a particular model was intended when creating a model. The model is intended to keep the developer focus on solving particular problems rather than figuring out whether it is sufficiently detailed and accurate.
- *Incremental changes.* Models do not need and are unlikely to meet all needs when they are first implemented. Modelling should start from high-level details and evolve over time in an incremental manner. The model should contain just enough details to solve current goals and may require refactoring.
- *Assume simplicity***.** Modelling should assume the simplest solutions are the best solutions and hence avoid unnecessary efforts spent on features that users do not need or optimizations that are not necessary – or focus on issues that may take long to materialize and are not essential. This is highlighted in the quotations: "*Do the Simplest Thing that Could Possibly Work*" and "*You Aren't Going to Need It*" which are manifestation of *Simple Design* principle in XP [54].
- *Working software is your primary goal***.** The main objective in software development is to produce high quality software. Every documentation, artefact or model that does not directly contribute to this goal should be questioned and avoided if it cannot be justified.

## 4.2 Architecture Design

The design process in WCRS started with an initial architecture design. The architecture of software system is described by TOGAF as:

> *"1) A formal description of a system, or a detailed plan of the system at component level to guide its implementation; and 2) The structure of components, their inter-relationships, and the principles and guidelines governing their design and evolution over time,"* [55].

It acts as the important entry point to the design process in software development. [46] described this step as "*architecture envisioning*" and considers it particularly important in scaling software development as it gradually become large and complex. As the project

progresses through the various decision-makings, it needs to be well-established architecture that acts as a baseline to such activities. Thus, it is concern about the evolution of the system as specified by the second meaning in TOGAF.

Architecture design is also connected to design goal of achieving *separation of concern* by the decomposition of functional elements into different subgroups. The mechanism of decomposition (or division) will affect every functional modules including the structural definition and their interaction. Thus, the architecture should be planned carefully to ensure that the system and its sub-modules align to the project objectives. This is again linked to the first meaning in the TOGAF definition.

The next sections will explain two key design decisions in architecture design: software architecture and physical architecture (deployment).

### 4.2.1 Software Architecture

As one of the major concerns in software engineering, the term *software architecture* could be understood through various definitions. [56] defined it as:

> *"The process of defining a structured solution that meets all of the technical and operational requirements, while optimizing common quality attributes such as performance, security and manageability,"* [56].

In addition, [57] defined *software architecture* by highlighting its important elements, listed as:

> *"1) Highest level breakdown of a system into part; 2) Decisions that are hard to change; 3) There are multiple architecture in a system; and 4) What is architecturally significant is one that can change over a system's lifetime.,"* [57].

These definitions mean that the *software architecture* phase represents the important design decision phase required to build a stable and scalable system. This is crucial for a successful system because it helps to address the quality and risk factors aligned to the project. Moreover, design concerns such as selection of algorithm(s), business logics and data structures often overlap with architecture decisions. Thus, the vision of architecture needs to be established to provide direction of development style.

This project is developing a web application; hence, the following sections will explore several *software architecture* styles for web application.

#### 4.2.1.1 Client-Server Architecture

The most common architecture style for a distributed system such as web application is the conventional *client-server architecture*. It divides the system into server components that offer services and client components that provide user interface to consume the services through connected networks. The server could serve request from multiple clients. This architecture style is also known as *2-tier architecture* [56].

Clients are often represented by range of applications with GUI to capture user inputs and transform them into requests to the server. The server contains data storage to collect, modify and distribute data. Client could be either a thin client if application processing logic is located at server side or a fat client if it is embedded with client application (see Figure 4.3). However, this architecture style lacks scalability because both the client and the server have limited resources. Reusing the application logic for different modules is also increasingly complex as systems expand due to their tight coupling between either data tier or presentation logic. In addition, it is difficult to maintain because it is hard to distribute new changes to system users.

#### 4.2.1.2 N-tier and Layered Architectural Style

The *N-tier* (multi-tier) or most popular *3-tier* architecture style addresses conventional *2-tier* architecture issues by placing the application logic at an additional tier (see Figure 4.4). *Layered architecture* shares the common goal with multi-tier architecture style – separation of functionalities into segments to improve scalability and maintainability. *N-tier* architecture is often used together with *layered architecture* style and they addressing following concerns:

i) Presentation logic (user interface);
ii) Business and application logic (domain process); and
iii) Data accessing logic (database communication).

The presentation tier is at the top most level of the system and exposes visual representation to allow users to interact with the system. It gathers the user intention in the form of commands and inputs and forwards them to the business tier. The business tier is responsible for processing data between presentation and data tier. It contains important application logic performs calculations, evaluation and makes logical decisions. The result of the logic execution will either become responses to the presentation tier request or forwarding to data tier for persistent. At the bottom, the data tier handles create, update, read, and delete

(CRUD) operations for the data sources. It contains various information to form appropriate queries for data retrieval. In an OO environment, it also in charge of handle mapping between information from data source to domain entity.

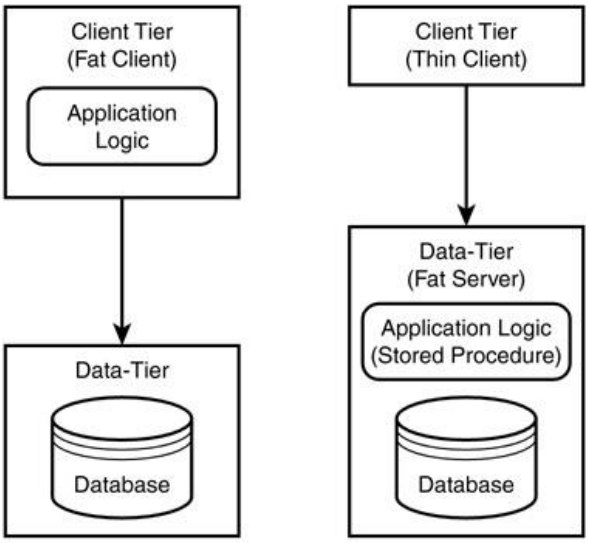



Figure 4.3: 2-Tier Architecture (Client/Server), [58].

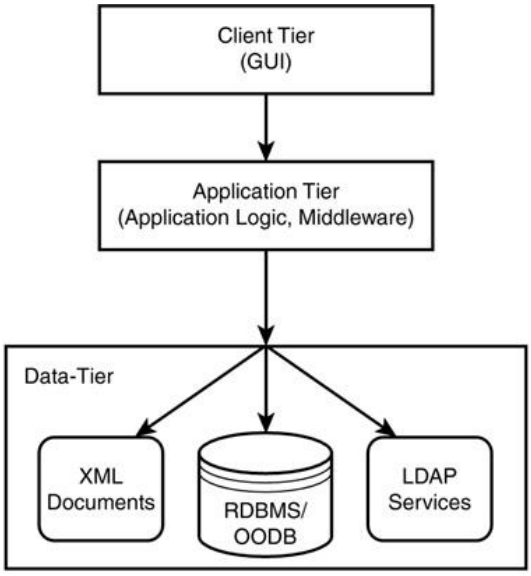



Figure 4.4: 3-Tier Architecture [58].

Based on the explanation by [56], the *layered architecture* style focusses on grouping related functionality into distinct layers and stacked vertically. *Layered architecture* model system structures in layers and each layer will have a collection of functions with similar concerns. Each layer can grow independently due to its loose coupling to upper and bottom layer.

In theory, *multi-tiered* and *layered architecture* styles have a number of similarities. However, *layered architecture* has adopted OO design concepts into its core principles. Each

layer promotes abstraction and encapsulation by providing just enough details to the dependant. It should also have high cohesion and clearly defined responsibility for maximized reuse. In contrast, *multi-tiered architecture* styles are concerned with the deployment of each segments into distributed environment, as discussed in §4.2.2.

#### 4.2.1.3 Software Architecture of WCRS

WCRS will employ a *3-tier (layers) software architecture* that is commonly adopted by web applications as shown in Figure 4.5. As discussed in the previous section, *N-tier architecture* enables *separation of concerns* and simultaneously promotes scalability and loose coupling among interaction objects. The responsibility of each layers and its contents are described in the following paragraphs.

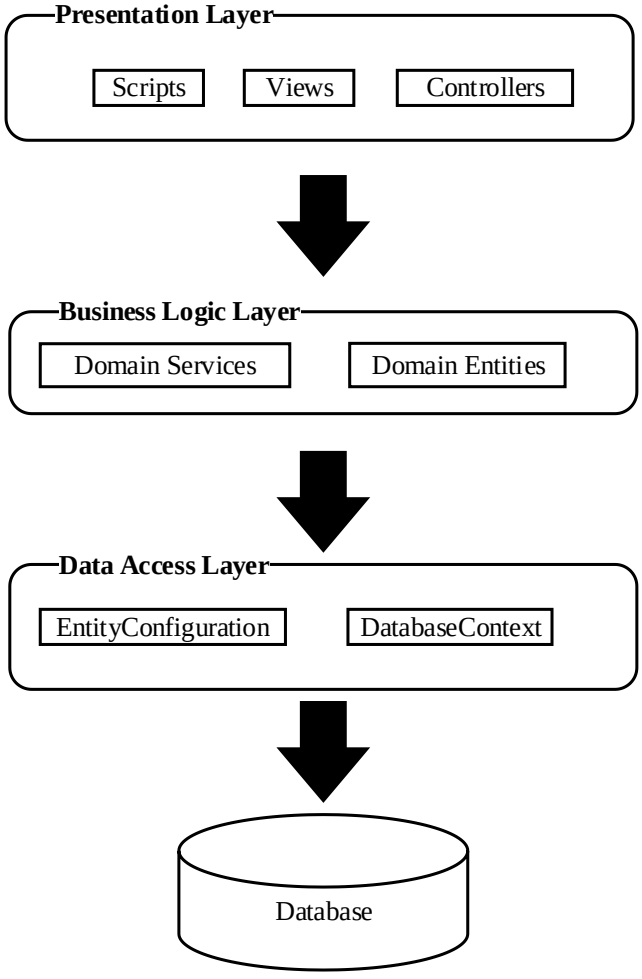



Figure 4.5: Overview of Multi-tier Software Architecture in WRCS.

Presentation in WCRS remains a major concern and aims to provide comfortable user experiences to users. As this is a web application, HTML, JavaScript and CSS will construct the UIs of WCRS. The *Presentation Layer* (PL) is responsible for mapping the information

collected from domain entity to the UI element in HTML. In addition to this, it handles certain UI transitions on the client side to allow the application more responsive to user gestures. It is also important to ensure the presentation logic decouples from the business process logic. Separated presentation patterns such as *Model-View-Controller* (MVC) and *Model-View-ViewModel* (MVVM) will be utilized to manage communication between the presentation components [56]. These are discussed in details and with the framework used to implement them in §5.3.3.

The *Business Logic Layer* (BLL) in WCRS is a middleware layer that provides the services of business functionality (e.g. manage order services) for PL. It encapsulates different computational logic, such as calculating order total amount or workflow logic, such as updating a menu plan within each domain services. It also contains important business entities that serve as basic data structures to carry and process information. This layer employs an OO design and analysis approach to address the structural and behavioural aspects of the application.

The *Data Access Layer* (DAL) mainly handles communication between the applications with the data sources. Its responsibilities include establishing necessary connections to the data source and construct necessary queries to retrieve and persist data. The database context will take care of the life cycle of database connections while entity configuration will adjust database schema on when the entity structures change. The design of this layer is mainly concerned with data structure and data storage methods.

### 4.2.2 Physical System Architecture

Once the *software architecture* is established, the next step is to consider a deployment scenario and the infrastructure of system. Figure 4.6 depicts the deployment diagram for WCRS. It illustrates the components of the system, their intercommunication, and the location of where they physically run. When deploying the system, most software packages are deployed into *Business tier* or the web server. However, the web browser of each client device will fetch some of HTML views and scripts when connected to the web server. This implies presentation components are partially deployed to *Presentation tier*. In addition to that, database schema in DAL will be used to construct the database instance in database server of *Data tier*.

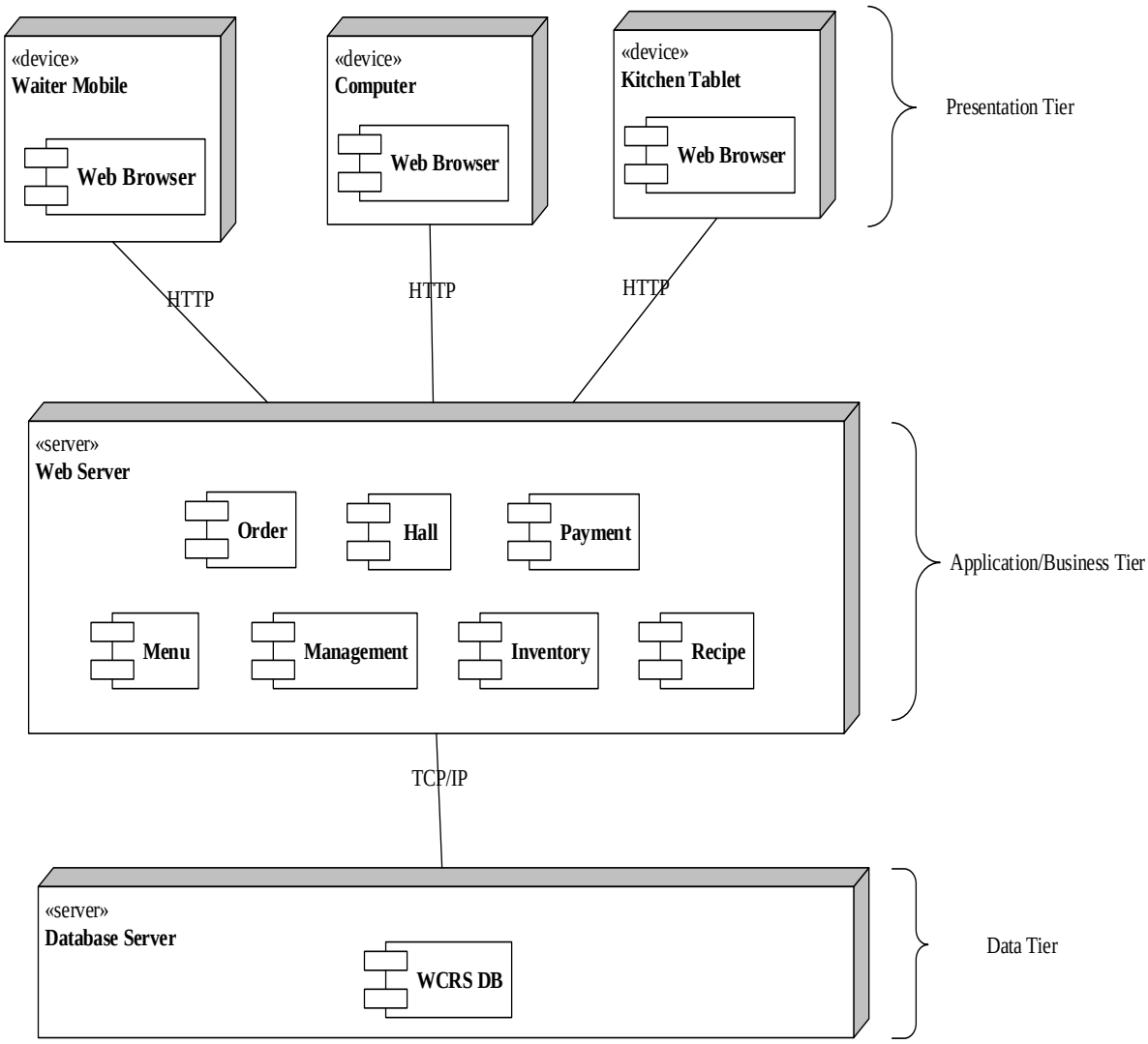



Figure 4.6: Deployment diagram of WCRS.

Figure 4.7 depicts an overview of the *physical architecture* that will be supported by prototype software inspired by [10]. The web application, deployed into the web server, will provide centralized access to its services based on the local area network deployed around the restaurant. Various types of device such as smartphone, desktop or laptop computer could be connected to this network. In addition, a database server will hold the operation data of the restaurant. Touchscreen panel or tablet with web browser will be used to enable order information sharing in the kitchen.

With both a vision of software and physical architecture established, the project has set a baseline for further design processes. The next steps will involve producing design models to address the design concepts of these architecture's components.

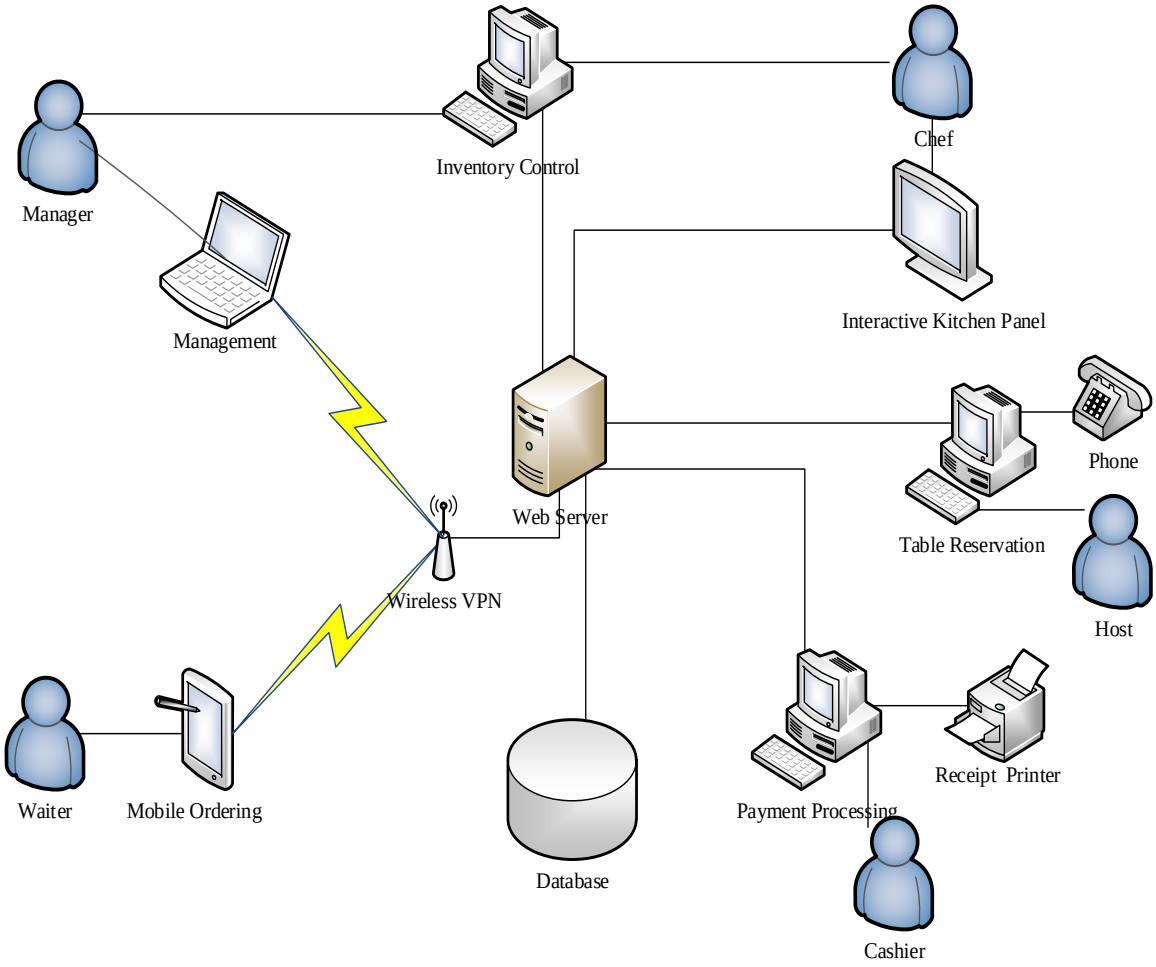



Figure 4.7: Overview of Physical System Architecture of WCRS.

## 4.3 System Modelling

*System modelling* is a process to construct abstracts representations of a system in several models, with each models encapsulate different views and analysis perspectives towards the system [22]. In fact, it primarily focuses on creating design models as discussed in §3.2. It works closely with *requirement modelling* by transforming requirement analysis result to design representation for building software [23]. These models encapsulated requirement understanding such as specification of software operational characteristics; software interface with other system elements; and constraints that software must meet [22, 23]. Often, these design models could easily translated from *functional* and *non-functional requirements* and via versa. They are used throughout development process and they are commonly used for [22]:

- Facilitating discussions about existing or proposed systems;
- Documenting an existing system; and
- Acting as a detailed system description, which could be used to generate system implementation.

Design models are often represented by different types of graphical notation with additional labels to describe their meaning. Depending on the development approach, different types of notation could be used to express a particular design. The most notable design notation used at present is UML:

> *"a unified modelling language that contains a robust notation for the modelling and development of object-oriented (OO) systems,"* [23].

Considering that WCRS is written in OO programming language, UML diagrams are de facto artefacts in the analysis and design process. However, this does not cover every representation or variant of design concepts (see Figure 4.8). Additional diagrams such as *Class responsibility-collaborator (CRC) Card* and *Entity Relation Diagram (ERD)* will be adopted to express uncovered design concepts.

To follow the *model with a purpose* principle, WCRS only employs models that are meaningful to the development process. While various types of models existed, modelling perspectives that are critical to WCRS are:

- *Behaviour model*, describes dynamic behaviour of a system and its response to events;
- *Structural model***,** describes organization of system structure and how its information is been processed; and
- *Data model***,** describes information that will be persisted and their relationships.

Each model could employ several possible modelling techniques and artefacts to represent its perspectives. The project goal does not exhaustively produce every artefacts of these models. It is rather to investigate how some modelling techniques could be applied to achieve the design objectives. The models also may not include every fine-grained level of details, yet these initial models will continue grow as developer refactors the design of the system.

### 4.3.1 Structural Model

A complex system consists of several sub-components and possibly some external systems. *Structural model* displays the organisation of these components and their relationships. It embodies important consideration about entities that will operate within the system. In

additions, [25] stated that *structural model* is to create a vocabulary that can be used by the analyst and the users. Things, ideas, or concepts discovered in the problem domain are represented as given object types in *structural model*, including relationships among such objects [59]. In fact, this process is also known as *domain modelling* in OO development. *Class responsibility-collaborator (CRC) cards* and *Class Diagram* are the two focuses of the WCRS structural modelling techniques.

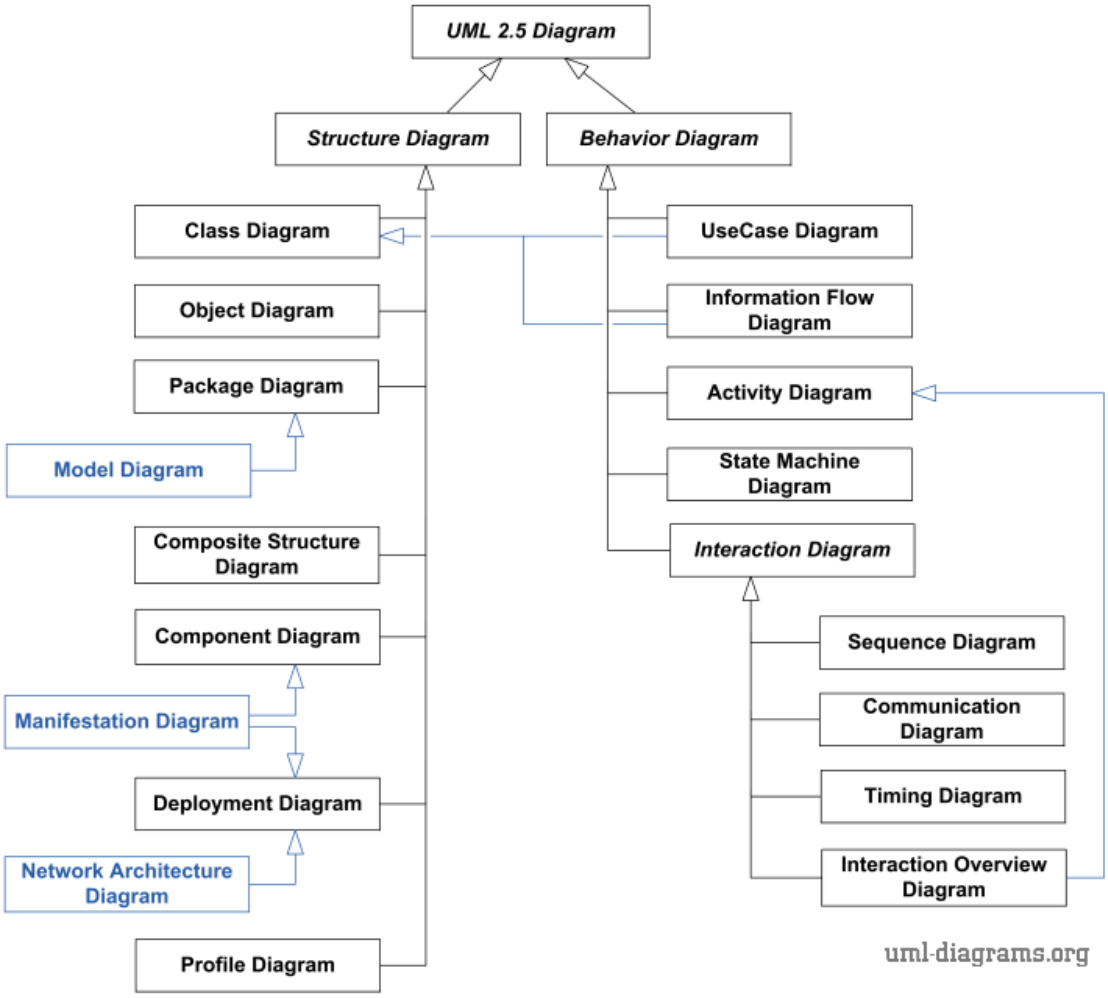



Figure 4.8: UML Diagram Overview [60].

*CRC card* is a collection of standard index cards that have three major sections as specified below [23, 59]:

1) *Class*, represents a template used to create specific collection of similar objects;
2) *Responsibility*, represents anything that a class knows or does as in form of attributes or operation; and
3) *Collaborators*, represents other interacting classes that are required for obtaining necessary information to fulfil responsibility.

Class name will present at the top, while responsibilities and collaborators will present at left and right panel respectively below it. CRC modelling is good option to explore structure and content of a particular class. Experimentally walking through use case with these cards could uncover design flaws such as missing properties, coupling and cohesion. Figure 4.9 show a *CRC card* for *Recipe* class. The class knows some information and they are often translated into property, while other responsibilities will be become operations. The other produced *CRC cards* can be found at Appendix H.

| **Recipe** | |
|---|---|
| Know title<br>Know description<br>Know price<br>Know serving yield<br>Know recipe category<br>Know menu group<br>Associate materials used for cooking<br>Calculate total cost | RecipeMaterial<br>MenuGroup<br>RecipeCategory |

Figure 4.9: CRC Card for Recipe class.

The established CRC model will eventually be translated into *Class Diagram*. The *Class Diagram* is a model that shows attributes and operations of system classes, as well as relationships among them. *Class Diagram* can have two levels of detail: i) *analysis class diagram* and ii) *design class diagram*. The *Analysis Class Diagram* is used to understand the relationship between domain objects while the *Design Class Diagram* contains in-depth descriptions of object's attribute and operations. In fact, *Analysis Class Diagram* is also known as *domain modelling* in OO development. Figure 4.10 shows the domain modelling of WCRS. It illustrated the possible elements that will be implemented as actual class. The more detailed *Design Class Diagram* can be found in Appendix I.

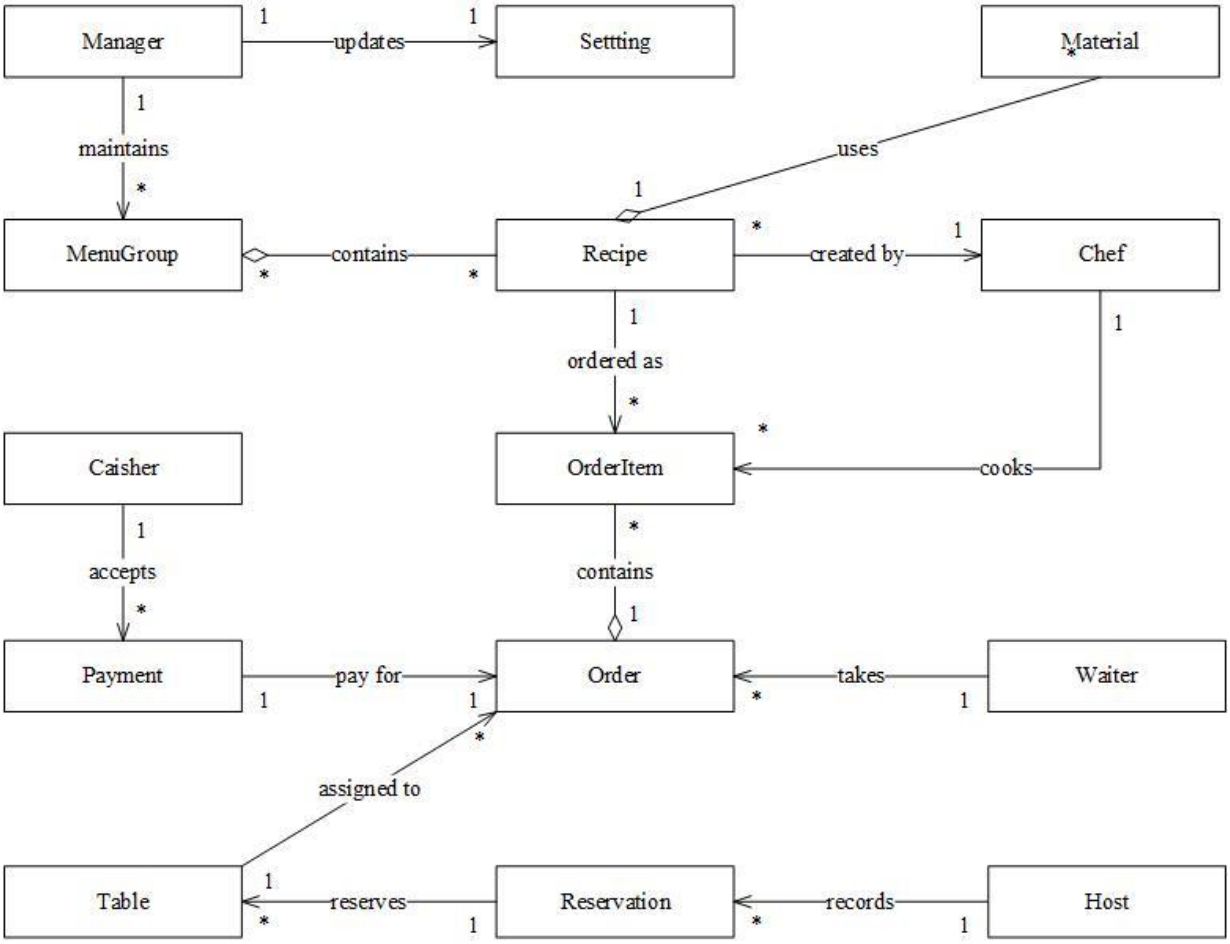



Figure 4.10: Example Domain Modelling for WCRS.

### 4.3.2 Behaviour Model

*Behaviour model* aims to model the dynamic behaviour during the execution of the system [22]. It can used to describe a use case at a specific time and event [23]. *Behaviour model* is particular useful to model a possible business process. The business process can be expressed in a series of continuous interacting objects in the system. [22] suggested that there are two types of behaviour modelling as shown in Table 4.1.

The table shows a comparison of two types of approaches in behaviour modelling. It describes the purpose of modelling when they should be used and the tools that could help during the modelling phase. As a system to support business operations, WCRS is expecting various explicit inputs from end users. For instance, when a waiter submits an order for a customer, the type of item and its quantity needs to be specified. Hence, *data driven modelling* is adopted because it fits the data processing nature of WCRS. In addition, since

the modelling process focuses on OO design, *Sequence Diagram* is used to model behaviour as opposed to DFD that is mostly used in a structured analysis design.

Table 4.1: Comparison of Behaviour Modelling types [22].

| | **Data driven modelling** | **Event driven modelling** |
|---|---|---|
| **Purpose** | Shows the sequence of actions involved in processing input data and generating an associated output | Shows how a system responds to external and internal events. |
| **When to use** | Useful when used to show entire sequence of actions that take place from an input being processed to the corresponding output | System has a finite number of states and that events (stimuli) may cause a transition from one state to another |
| **Tools** | Data-flow diagrams (DFD), Sequence diagrams | State diagrams |

*Sequence Diagram* is one of the UML diagram to model behaviour and interaction between objects, including actors. It demonstrates a sequence of interaction activities during a system flow. The interaction activities could be a reflection of an explicit sequence of messages that have passed between objects [25]. All the objects will be arranged in a parallel line and a vertical dotted line indicates its active timeline. [22] specified that not every detailed should be included in a *Sequence Diagram* unless it is used for code generation. The reason being it may lead to a lot of premature implementation design. Eventually, it could easily fall into entropy of "*big design upfront*".

Figure 4.11 depicts a *Sequence Diagram* to capture behaviour to submit a new order. It consists of high-level view abstraction on the potential classes and messages exchanges required to be carried out to submit an order use case scenario. Messages flows will inspires operations that need to be implemented for the potential classes. The remaining sequence diagrams can be found in Appendix J.

### 4.3.3 Data Model

*Data model* is concerned with exploring data-oriented structures [59]. It aims to define the structure of data objects, their relationships and related information that describe the objects and relationships [23]. *Data model* is important because every application, at a certain point, will need to persist its data to certain type of storage. The structure of data, if not carefully designed, will affect data retrieval performance. The *data model* design is closely related with the decisions of data structures. Before this process can even begin, the type of data and

storage strategy needs to be investigated and carefully selected to accompany the data processing need within the system.

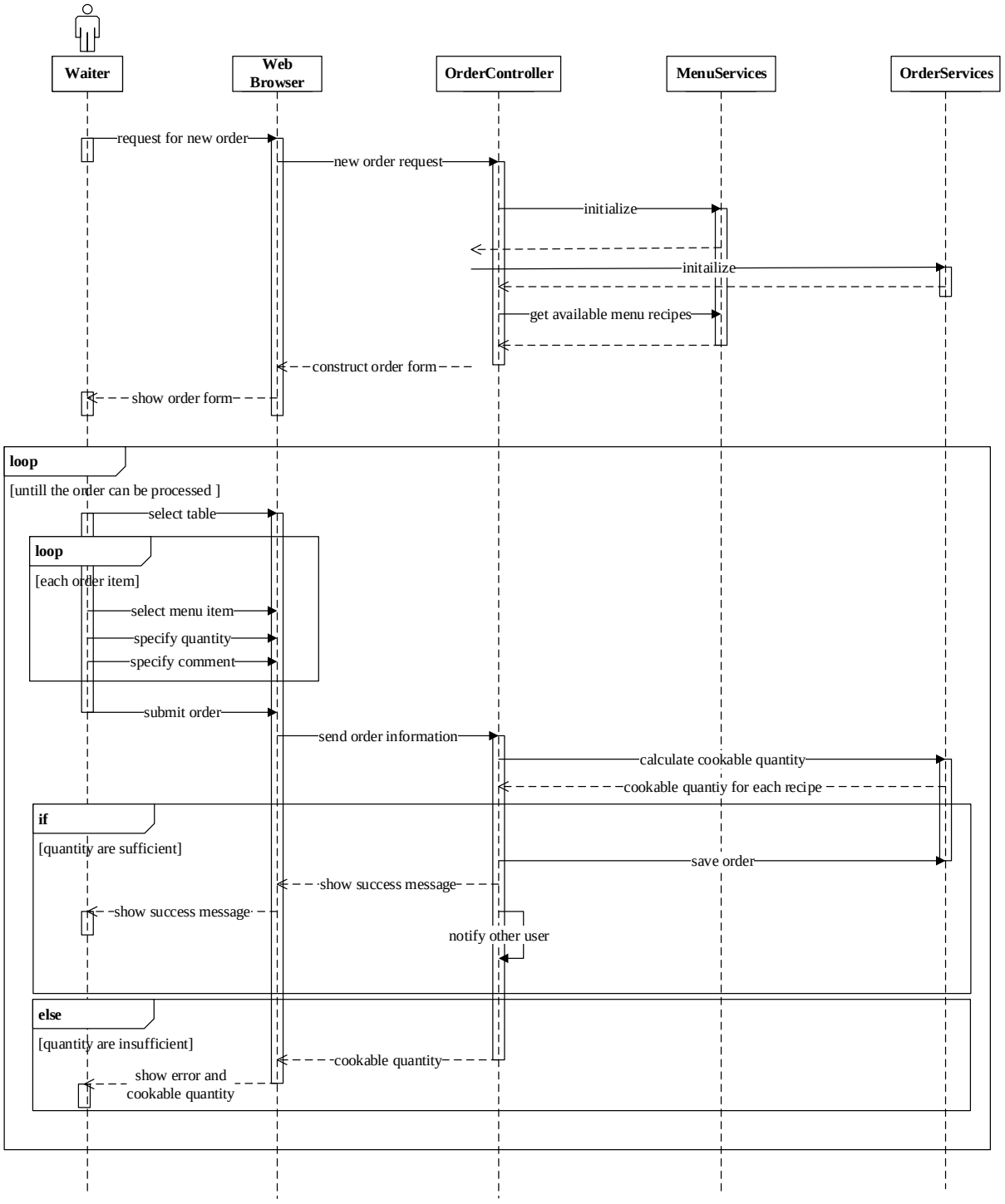



Figure 4.11: Sequence Diagram to model submit order behaviour.

#### 4.3.3.1 Handling Data in WCRS

The author has explained the characteristics and content of data that flow through the system in different sections in this paper. This issue was initially addressed in *Context Diagram* (see §3.2.1). The *Context Diagram* let us: the designer (and/or the developer), realise a view of input and output data. The views are further developed into more concrete ideas regarding inner content of data, through structural models. This section develops the concepts related to the explicit data structure concern in this project.

The first issue, if data structure is concerned and related to this project, that needs to be addressed is: persistent or non-persistent. Once the decision of persistent or non-persistent has been made, the next issue is whether this data is going to be stored externally or internally to the application. This implies an external data store that will need to be selected, designed and implemented. In fact, it involves designing different types of data structure. If different types of data structures need to be designed, we need to think about different type of physical files that need to be processed. Hence, the next decision would be the type of files: whether they are simple or complex. If the file is going to be processed implicitly, it could be a text file, or *comma-separated values* (CSV) file. However, if the file required more processing and involves complex read-write operation, it will be more appropriate to select a formal mark-up file as the data store. This could be a data structure of relational model or hierarchy model (e.g. Extensible Mark-up Language).

Table 4.2 shows the type of data that will be required by the WCRS. The table explain these data descriptions, their persistent requirement and processing nature. *Application Setting* is the top level setting that affects the entire behaviour of the system. For instance, the tax rate setting will affect every payment received by the system. It hardly changes but often read by application for decision-making and computational purpose; hence, it should be persisted and allowing user to change it. Next, the *Business Entity* refers to the object that holds crucial information representing a real life entity, e.g. order. This serves as the foundation building blocks to construct system functions. It needs to be stored and often involves heavy CRUD operations throughout application runtime. Finally, *View Data* is a model constructed to carry relevant information to UI for display purposes. It is often reconstructed using *Business Entity* objects and does not need to be recorded. Once we understood the necessary characteristics, the next step would be investigating a data store that could enable effective processing of them.

Table 4.2: Type of Data in WCRS.

| File Type | Description | Persistent or Non-Persistent | Processing nature |
|---|---|---|---|
| **Application Setting** | File contains necessary configuration that affect application behaviour | Persistent | Read intensively and Write infrequently |
| **Business Entity** | Model that represent a real life entity of the business domain | Persistent | Read and Write intensively |
| **View Data** | Model that constructed during the execution time to display necessary information to user | Non-Persistent | Read intensively |

#### 4.3.3.2 Data Storage

Data storage is the container resource for data objects. The data structures that will be persisted in the system need to be stored with one or more data storage methods. This section investigates two main options of data storage and describes the chosen methods for WCRS.

#### 4.3.3.3 Relational Database Management System (RDBMS)

RDBMS is data-processing software that employs relational model as its fundamental data structures. [61] describes the relational model as following:

> *"In the relational model, all data is logically structured within relations (tables). Each relation has a name and is made up of named attributes (columns) of data. Each tuple (row) contains one value per attribute,"* [61].

It presents the data in tabular form identical to spreadsheet format. The standard language used for data manipulation in RDBMS is Structured Query Language (SQL). SQL consists of two major components: Data Definition Language (DDL) for defining the data structure and controlling access to data; and Data Manipulation Language (DML) for retrieving and updating data [61]. SQL can be very powerful depending on the usage. Some applications leverage its capability to transfer part of the system computation logic to RDBMS through writing stored procedures with SQL [61].

RDBMS has become the dominant data storage methods for most software systems today, particularly web applications. There are many existing RDBMS solutions; all share similar sets of essential data processing functions varying slightly in the provided features. There are several mature commercial solutions such as Microsoft SQL Server [62] and Oracle Database [63]. Conversely, there are also free open source solutions such as MySQL [64]. These have been widely adopted by industry, particularly in small and medium business. The advantages of RDBMS are:

- Support for controlling concurrency and transactional access;
- Support for security management;
- Minimal data redundancy;
- Simple user interface to manage data schema;
- Ensuring data integrity through constraints; and
- Fast data retrieval through query optimization.

#### 4.3.3.4 Extensive Mark-up Language (XML)

XML is a meta-language that allows designers to define their own customized tags in a document. As a type of semi-structured data, XML is designed to be self-descriptive [65], readable by both humans and machines. It has a loose restriction of schema, thus allowing it to handle data structure that changes rapidly and unpredictably. This is especially true for information on the Web, where it requires certain degree of flexibility to accommodate ever-changing HTML design. The software industry today uses XML as the de facto standard for data communication [61]. It has evolved to be the primary medium of data exchange with external systems and among businesses.

Many technologies have built upon XML by a predefined structured format for XML with XML Schemas. These schemas lead to standardization of XML format when adopted widely by the industry. SOAP[6] and RSS[7] are some of examples of spin-off technologies based on XML. XML also has become popular thanks to a wide range of query languages available, including XPath[8] and XQuery[9]. These languages allow manipulation and retrieval of data to

[6] Simple Object Access Protocol, an XML-based protocol for applications to exchange data through web.
[7] Really Simple Syndication, a protocol to share site content and headlines with XML.
[8] A query language for finding elements and attributes in XML document.
[9] A query and process language of XML document.

become relatively simple. XML could be considered as the data storage if following advantages can be utilized:

- Ability to deal with frequent and unpredictable schema changes;
- Modelling hierarchy data structure effectively;
- Minimum data conversion if data is used directly from source; and
- Support for multiple platforms.

#### 4.3.3.5 Storage Method Chosen

The storage method chosen for both persistent files (*Application Setting* and *Business Entity*) is the RDBMS. The main rationale was the *Business Entity* – primary data structure of WCRS, required extensive cross-referencing. For instance, an order needs to know which recipes been selected and the recipes need to know what are the materials that need to be consumed. Putting this data into hierarchical models will result into redundant data everywhere. RDBMS is also has strong security measures (e.g. access control) and they are important for *Application Setting*, which contains critical data that would affect entire system behaviours.

In addition to that, another key factor that leads to this decision is the existence of *Object Relational Mapping* (ORM) solutions. ORM solution mainly help in converting the relational model into interconnected object graph, coined as the "*Object-relational Impedance Mismatch*" problem [66]. This significantly reduces the complexity of retrieval relational data into OO environment because complex SQL queries could be simplified into normal OO method calls (see §5.3.1.1 for usage).

Conversely, using XML as data storage could be relatively complex and verbose. The processes to retrieve the data in documents, map them to object and primitive data type in programming language, is difficult. Because the data type constraint is not enforced within XML, the designer will need to handle various kinds of data processing issues such as null value and format mismatch.

#### 4.3.3.6 Entity Relationship Diagram

*Entity Relationship Diagram* (ERD) is widely used data model to represent relational data model of database. It is "*a picture which shows the information that is created, stored, and used by a business system*" [25]. There are three major concepts in ERD. First, each data object in ERD is a named *entity*, often mapped to a *table* in RDBMS. Second, information

with an *entity* is represented by a set of *attributes* – which capture the data segment (e.g. recipe title) of a data object (e.g. recipe). Finally, association among *entities*, also known as *relationships*, depicts high level business rules of a system [25]. The ERD for WCRS is illustrated in Figure 4.12. This model is reflecting actual implementation in the database for data persistence.

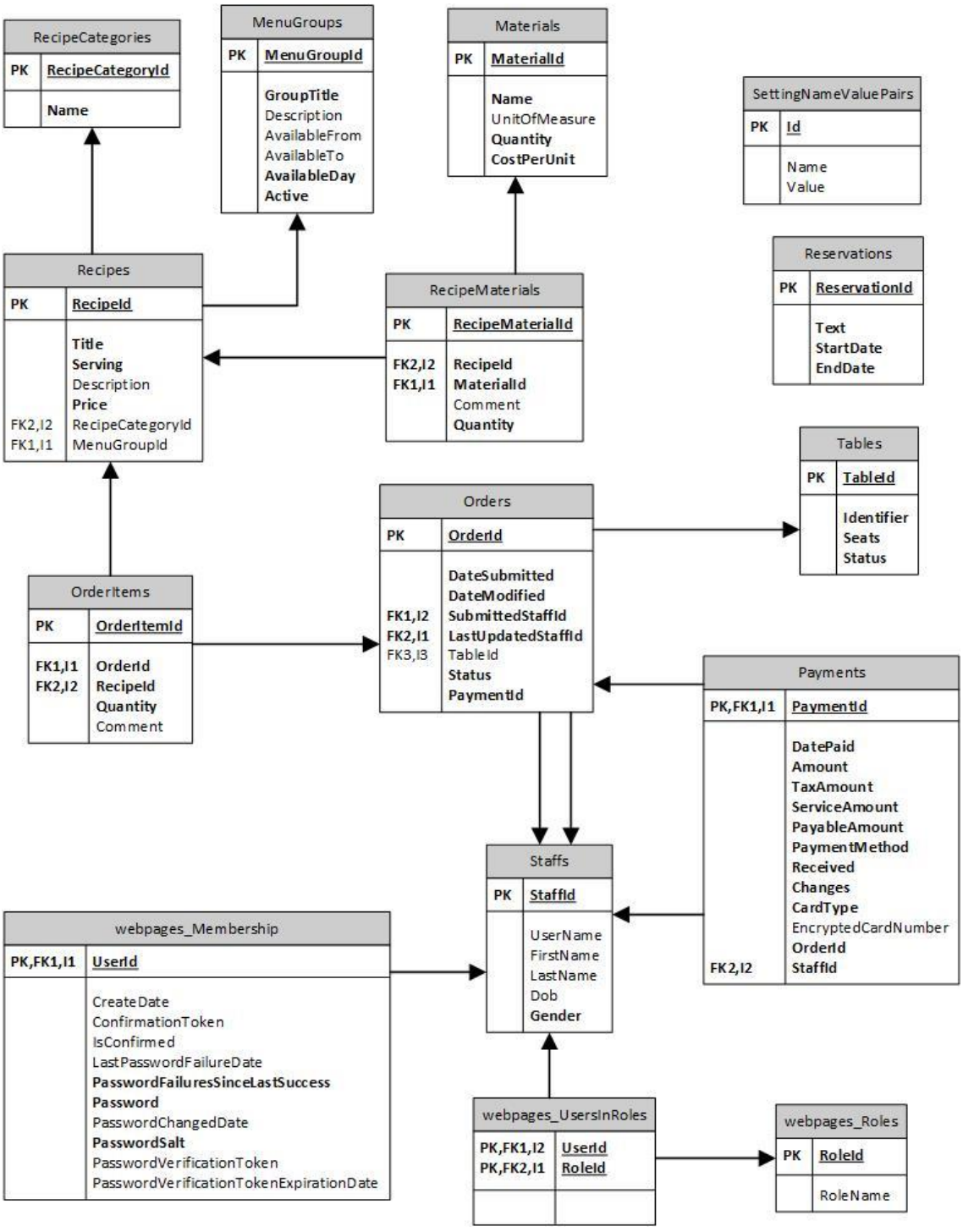



Figure 4.12 : Entity Relationship Diagrams of WCRS.

## 4.4 User Interface Design

This section develops the concepts related with modelling the graphical user interface (GUI) of WCRS. Two important concepts utilized in designing the UI in WCRS are *Hierarchical Task Analysis* and *Responsive Web Design*. The following sections will described how the design concepts contributed to the final design with high fidelity design utilizing wireframe diagrams. These solely serve as tools to communicate ideas and thus not every design is shown.

### 4.4.1 Hierarchical Task Analysis (HTA)

When users interact with the system, they are trying to perform actions that could meet their high-level goals or needs. These actions, more appropriately called *Tasks*, often can be broken down into series of small steps. UI of any system are means to assist users to go through these steps and provide comfortable experience during this process. Hence, the project employs HTA to understand several important tasks that the user is performing. This understanding will lead to design model with appropriate UI elements and behaviours that match user tasks.

To bring this more into context, this section will set an example for a submitting an order scenario. Figure 4.13 demonstrates the HTA diagram for submitting orders. It represents the necessary steps required to complete the submit order task. To demonstrate the concepts, the UI prototypes are shown at Figure 4.14 and the remaining at Appendix K. The task which each diagram represents is labelled explicitly at the end of its figure caption. Other non-trivia HTA diagrams can also be found at Appendix L.

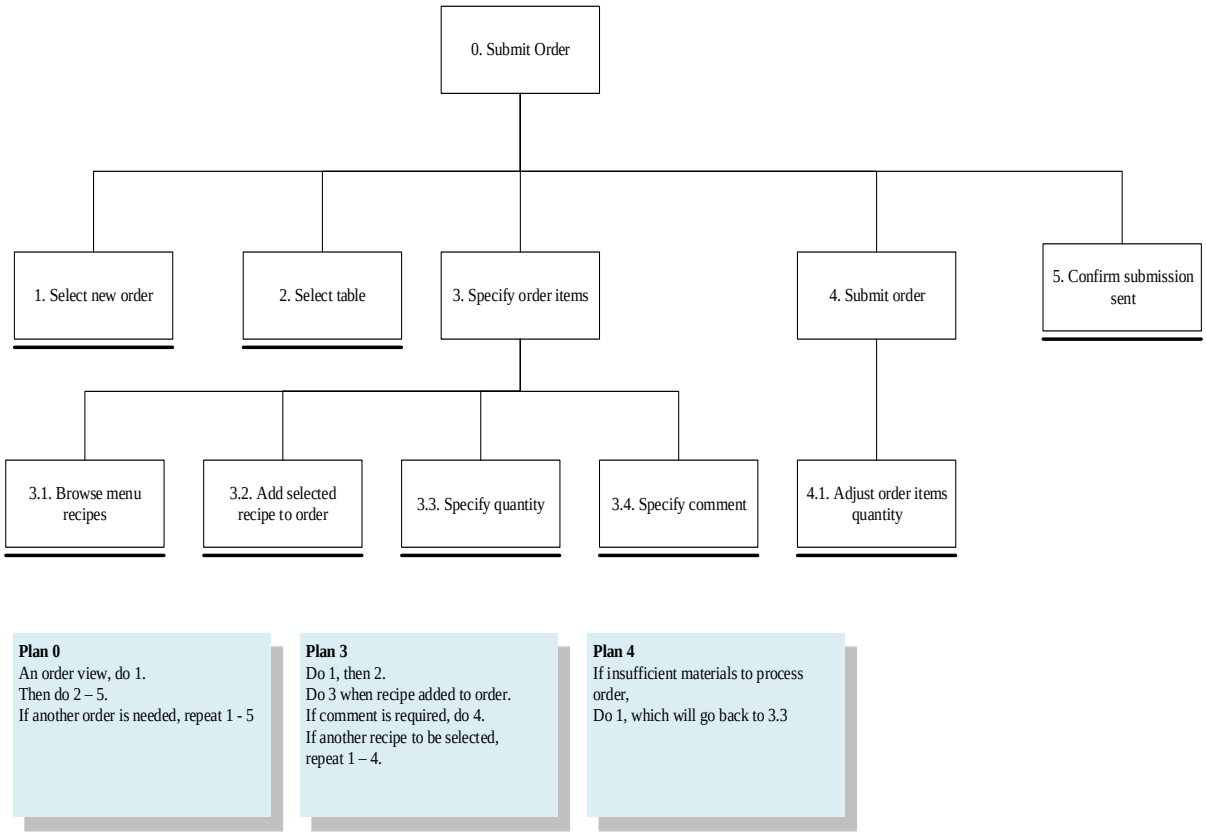



Figure 4.13: HTA diagram for Submit Order.

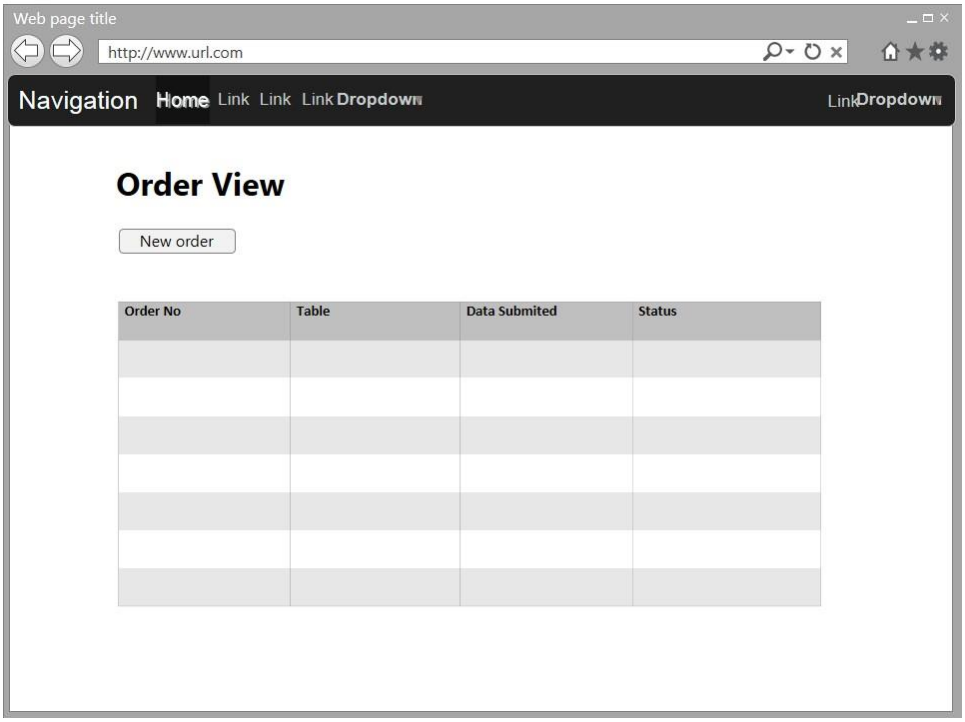



Figure 4.14: Design shows user select new order (HTA #1).

### 4.4.2 Responsive Web Design (RWD)

As a web application, WCRS mainly interact with its user through a web browser. Web browsers existed in almost every computing device, including mobiles and smart televisions. Hence, this implies the system is potentially accessed through any kind of device. One could argue that we could develop UIs that target each type of device. However, the user's roles may overlap and explicitly forcing them to switch to other devices to carry out the roles could be troublesome. It also places a restriction on the medium to be used to access system, resulting in an additional cost involved to acquire the right devices for the UI. The UI of WCRS therefore needs to adapt screen resolution of different devices through RWD. RWD's technical concepts initially discussed in §2.4.2 and this section explains how it addresses specific UI design issues in WCRS. This is mainly based on Letchford's [67] comprehensive list of UI issues and their solutions.

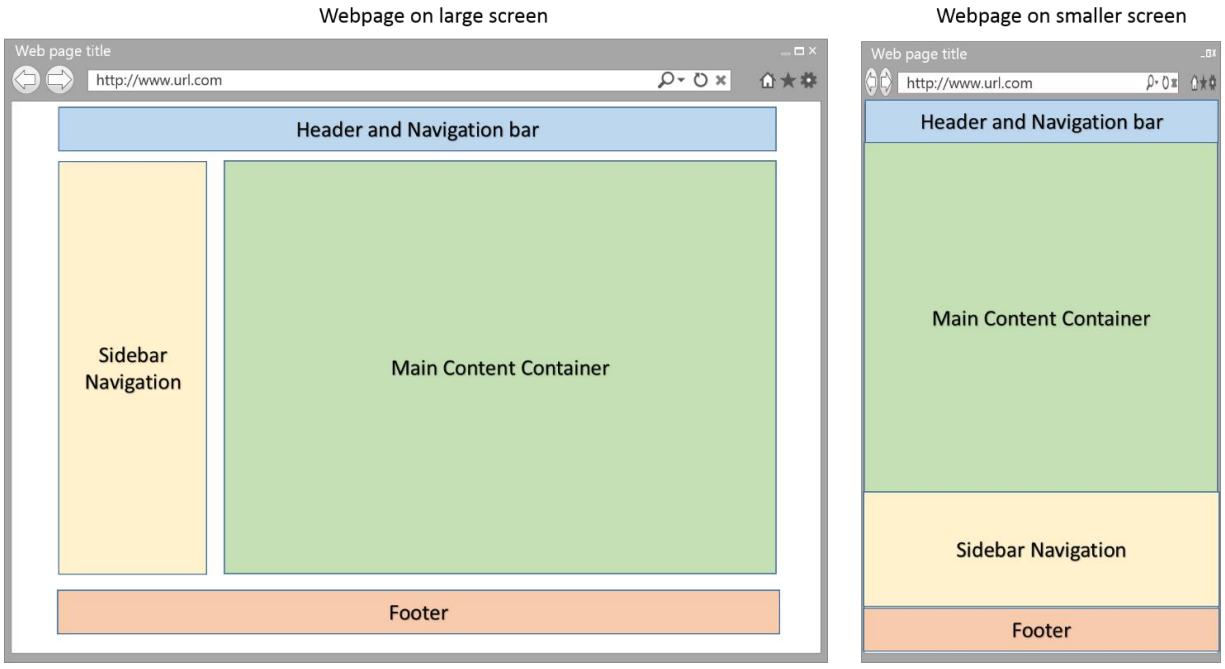



Figure 4.15: Layout changed when viewing on smaller screen.

One of the common issues in UI design is mismatched layout of content on different screen orientations. Landscape layout is wider and its content can exploit the width to present information horizontally. In contrast, portrait layout would have presentation issues with content that with a large width. Particularly on smaller devices, the user needs to browse the content with a horizontal scroll or zoomed out. This could significantly degrade the user experience. To bring this into context, Figure 4.15 presents the common layout of a web page. It usually contains header, main navigation, sidebar navigation, main content container

and footer. However, this layout is difficult to fit into a smaller device with portrait orientation. This will be worst when the main content container has components with large width. The solution to this problem is using flexible grid design that split available space into number of columns (commonly 12) as shown in Figure 4.16. The layout maintains a relevant proportion of width until it is viewed at a small resolution web browser. Its contents will then be stacked together, allowing browsing content easily on portrait orientation (see Figure 4.15).

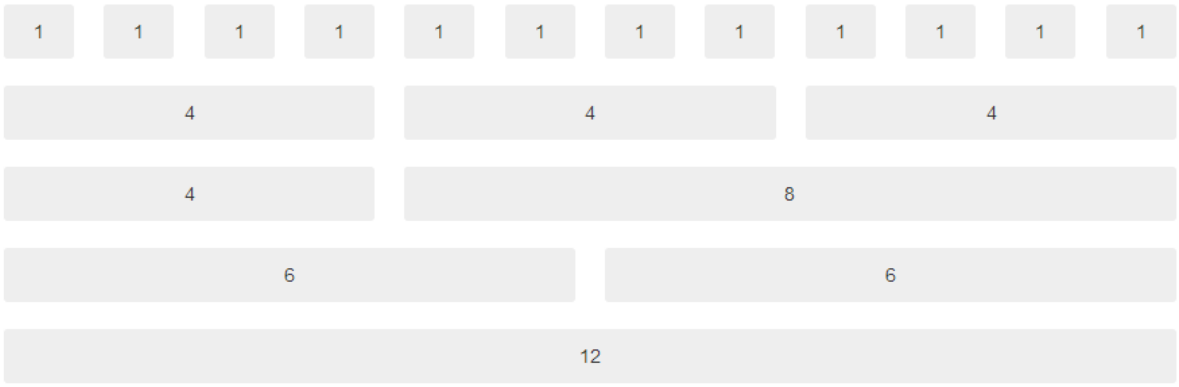



Figure 4.16: Flexible Grid System [42].

Another issue of UI design is the navigation menus that are often span across the header section. A low-resolution screen could not accommodate many menu items in its header. However, these menu items are important navigation concerns and could affect usability of the system. Thus, the system needs a solution to present the menu items while not explicitly occupying spaces in the header section when they are needed. A good solution of [42] would be temporary hiding the menu items from the header section and presenting a button to toggle its visibility as shown in Figure 4.17. This allows users to access the navigation menu whenever they need.

In addition, presenting tabular data on the mobile could be challenging due to table row spanning horizontally. The column that has more text if shrunk down will result in the text wrapping together and making the table look untidy. The solution would be hiding less important columns when viewing at smaller solution as shown in Figure 4.18. The hidden information can be accessed through the "*details"* button, which bring users to a more detailed page for the selected row.

Most data entry in web applications involves form. Users will need to go through every field as in filling out a paper form in real life. On a wider screen, the form could be arranged with

its field labels staying side by side with its input element. This again is a problem for the smaller screen because the input elements will span out of view. This introduces the risk of unfilled fields due to poor visibility. The solution would be stacking up the labels and inputs vertically as shown in Figure 4.19. Users could browse every fields by scrolling vertically; hence, it less likely to miss a field.

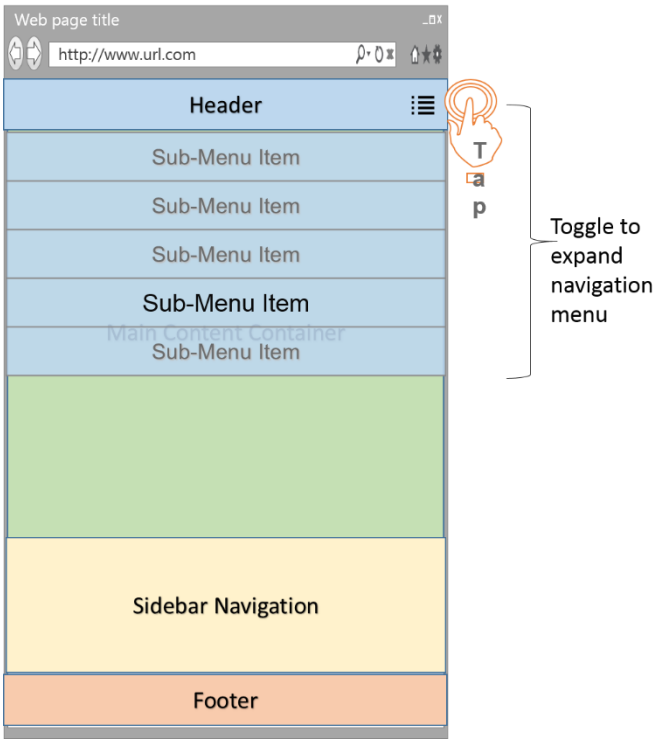



Figure 4.17: Toggling menu button to view navigation menu.

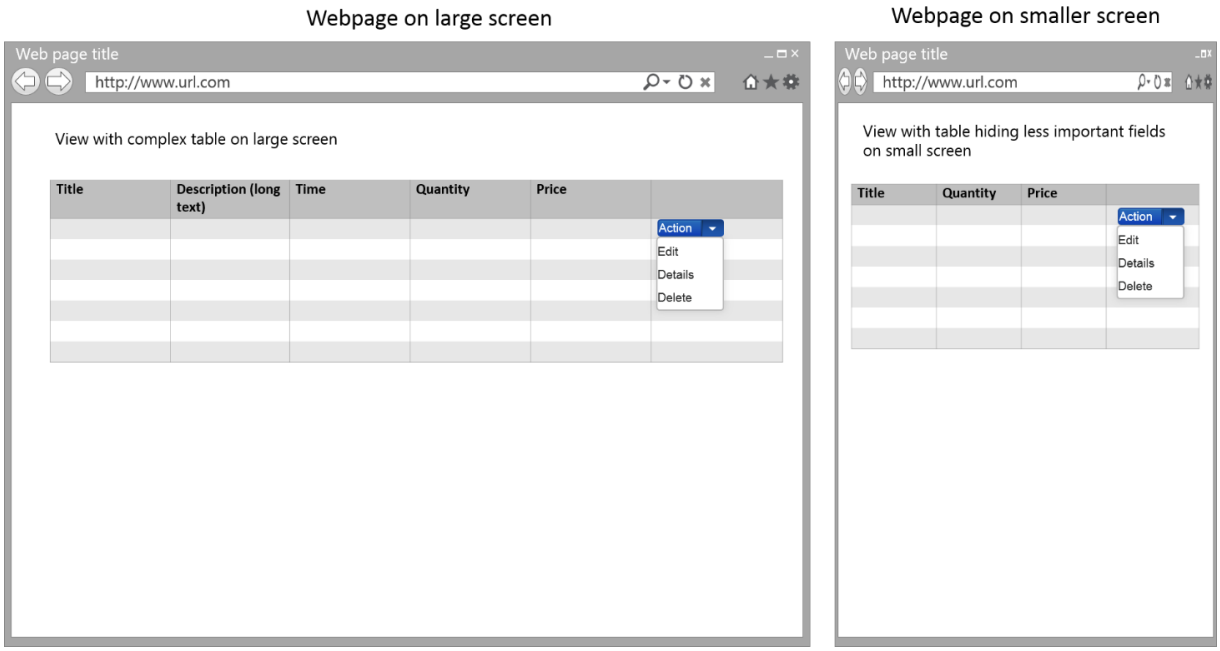



Figure 4.18: Hiding less important columns on smaller screen.

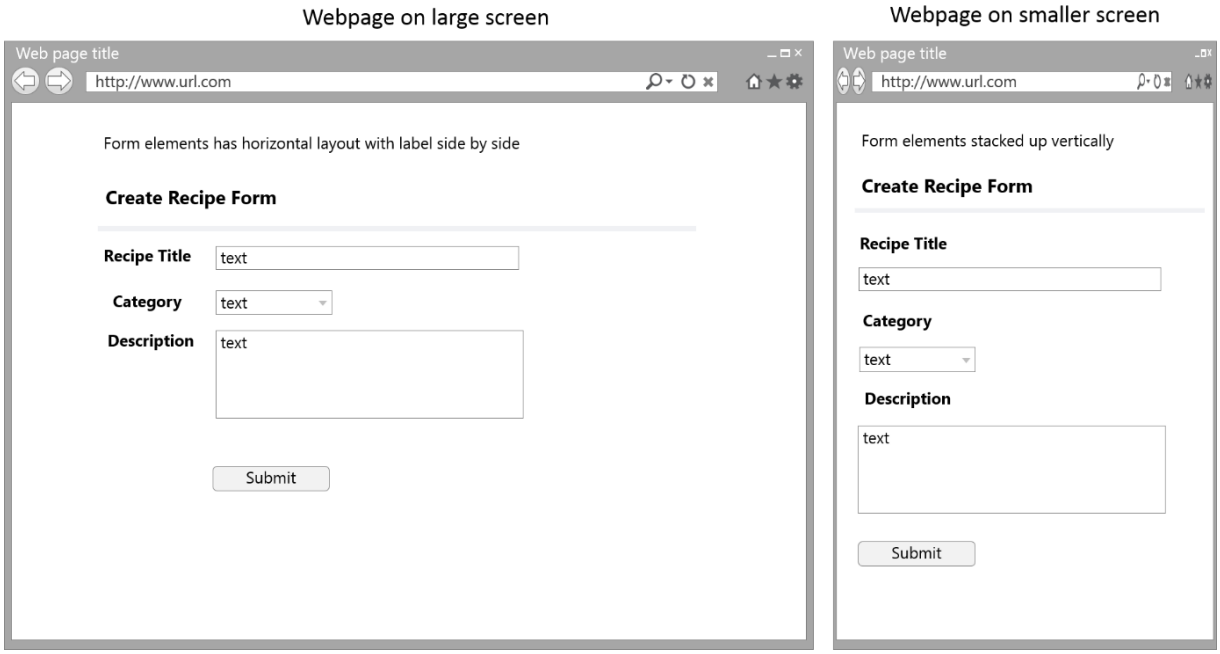



Figure 4.19: Form elements are stacked vertically on smaller screen.

## 4.5 Conclusion Remarks

This chapter has explored software design concepts in detail – after conducting rigorous researches and explaining how the author formulated the solutions to the requirements. The principles of agile development for software design set the best practise for design activities – and were used by the author. Architecture design established the high-level vision for all the system components design. System modelling enabled abstract representation of a solution that could guide implementation activities. Finally, user interface design addressed the usability and presentation need of the prototype software.

While it is not possible and realistic to include every models that has been produced, the design process has proven that it has carefully considered requirements and constraints that must be met – and indeed were met. The design models are guidelines but not intended to set implementation details in stone. Thus, it is quite possible that more good practises adopted during implementation have not been explicitly specified.

The next chapter specifies the actual implementation process. It explains how the software that has been implemented, based on the design ideas that are developed in this chapter.

# Chapter 5. Implementation

This chapter will cover the theory behind software implementation and complex features; that had to be researched and then implemented. It will mainly discuss the development details of the previously discussed design ideas (§4.3 and §4.4) while addressing project requirements (Appendix C and Appendix D). It first explains the implementation technologies – which are required for web application development. Then, it describes the server-side implementation and client-side implement of the system. The server-side implementation will mainly broken down by architecture layers (see §4.2.1.3) which also includes the algorithm of data processing. Client-side development is concerned with the UI implementation. Finally, this chapter presents a series of walkthroughs that depicts the actual usage of the implemented system.

## 5.1 Implementation in Agile Development

As discussed in §2.3.4, XP was employed as the software process model. Implementing the project requirements will involve iterative and incremental release [68] – a core agile practise. In this phase, the development process will be broken down into series of time-boxed implementation iterations. The iteration will involves software designing, programming and testing. Before each iteration begins, a set of requirements to be implemented is specified. The deliverables of each iteration will be an incrementally functional prototype software based on the specified requirements. In WCRS, the breakdown of iteration follows a series of prototype version. This is shown in the prototype implementation table in Appendix M.

Iteration development in XP is at the "extreme" level. Different software features are embedded in the system during the iteration. It is important to take account that this (way of evolving the system) could degrade the software structure; the more changes that are introduced can lead to it becoming increasingly difficult to implement [22] due to evolving complexity. XP addresses these issues by suggesting developer should constantly refactor the design. In addition, testing should be taken into consideration before integrating new changes. This is aligned to the concept of *continuous integration* (CI) [69] – an approach to address integration problems by ensuring new changes are verified and software rework may be expected after adding these changes. CI best achieved by utilizing CI software to manage

source code repository and software version builds. The CI software that is used in WCRS will be discussed in §5.2.5.

## 5.2 Web Development Framework

Implementation of a software product requires various software development kits (or frameworks) such as programming tools, build system and source code repository. Utilizing the right tools will significantly improve the development efficiency, performance and software quality. Given that this project was set to develop a web application, it involves both the server side and client side of development. Client side development mainly involves constructing UIs (mainly written in HTML) for the web browser. Conversely, server side development employs the OO paradigm when implementing the required algorithmic and data structures. Thus, it needs a *web development framework* that could bridge the gap and the mismatch between the two programming environment; client- & server-side.

This project employs ASP.NET MVC as the *web development framework.* ASP.NET MVC is

> *"a framework for building scalable, standards-based web applications using well-established design patterns," [70].*

The term *design patterns* refer to the enormous number of built-in software libraries that could be used to solve common web development issues such as model-view separation, UI, routing, dynamic content, etc. The framework and its architecture are based on the MVC pattern; which is an excellent candidate to achieve *separation of concern.* Another key factor is ASP.NET's support for server push technology[10], which employs the publish-subscribe model[11] to maintain communication between server and web client. This is particularly important for the order notification between chef and waiter to ensure that the order requests are handled in a timely manner. In addition, ASP.NET MVC has security features that are built into the application framework. Particularly *role-based authentication*[12] is a perfect candidate for WCRS since it will be accessed by various user role.

[10] A technology that allows server to send data to client without recreates new connection.
[11] A communication pattern that allows any receiver that interest with particular topics, subscribes to the message producers (sender) and receive message when the sender broadcast the topic related messages.
[12] An approach to restrict users' access to the system based on their job functions.

In additional, ASP.NET MVC employs a web template engine, named *Razor*, to create dynamic HTML. Web template engine refers to a solution that separates the presentation of HTML from its data, allowing them to be interchangeable at the run time. The steps to use a template engine are specified by [71] as below:

- Specifying a template to use;
- Assigning data to the parameters as the actual content; and
- Injecting the template with the parameters to generate HTML results.

*Razor* allows the programming flow to easily swaps between .NET programming language and HTML. Most of the HTML tags could be generate through the helper class that bind to the data of the view. The generate result is the HTML content that contains object instances' data. The usage of *Razor* will further be discussed in §5.3.3.1.

### 5.2.1 Server-Side Programming Language

The underlying server side technology for ASP.NET is the Microsoft .NET Framework. While there are a number programming languages available under .NET Framework, C# and Visual Basic (VB).NET are the most widely adopted languages. VB.NET employs OO paradigm and emerges as an evolution to its predecessor Visual Basic[13]. The syntax of VB.NET is more literal and closer to the pseudo code expression. In contrast, C# syntax is much closer to the other widely adopted OO programming language such as Java and C++. The code block is wrapped with braces and could be omitted for a single line statement. Both languages are fully supported by the .NET Framework and easily port to the counterpart.

C# has been selected as the programming language for this project due to the author's familiarity with the language and the fact that many code samples in .NET community are published in C#. Other interesting features of .NET that heavily exploited in the project are LINQ [72] and *Lambda Expression* [73]. LINQ stands for Language-Integrated Query and is particularly useful to perform query and operation against data. *Lambda Expression* is a style of anonymous function that could be passed as arguments to another function call. A combination of both features allow the developer to define efficient queries against the data source such as entity class of ORM or enumerable object collection.

[13] Event driven programming language for Component Object Model programing model.

### 5.2.2 Client-Side Development Language

The client-side of the web application refers to the web browser. Regardless of whichever technology is employed to construct the UI, the web browser will only process the received view as HTML. Hence, the client-side UI of WCRS is mainly written in HTML5. HTML5 is the new standard of HTML and currently supported by a wide range of web browsers [74]. Some HTML5 characteristics desired by the project are [75]:

- Adding new functionality is purely based on HTML, CSS, Document Object Model (DOM), and JavaScript. This avoids unnecessary installation of external software plugins to the user device; and
- HTML5 prefers more mark-up to replace scripting. Many common features (e.g. validation, UI elements) are included in the standard of HTML5, hence the project can avoid reinventing the wheel.

However, HTML is static and its presentation unlikely to change once rendered by the web browser. Therefore, the project also utilizes JavaScript to introduce dynamic behaviours to the HTML pages; e.g. responding to user inputs, changing content structure and styles.

*JavaScript* has become the dominant client side of scripting technology in recent years. It is open platform and has numerous libraries that enable the streamlining of modern web development. Most web browsers today support *JavaScript* execution, thus allowing more interactive and responsive experiences built into web applications. ASP.NET comes with several helpful built-in JavaScript libraries. The most notable *JavaScript* library is *JQuery* and its validation plugin. *JQuery* provides API [14] for HTML document traversal and manipulation, event handling, animation, and Asynchronous JavaScript and XML (Ajax). It allows the developer to create rich and dynamic UIs at the client side. Besides, the validation plugin contains the common validation functionality for HTML input such as textbox. This could significantly reduce data entry error and improve data integrity

Plain HTML is not appealing to users and is likely to provide a poor user experience. Hence, HTML often uses CSS to define its look and feel. Aligned to the concept of RWD discussed in §4.4.2, the project needs a CSS framework to provide the responsive features and to enable the views scale properly across different resolution devices. Besides, it may also save

[14] Application programming interface, describes the approaches to interact with software components.

significant development time when customizing the layout. Hence, the project utilizes Bootstrap as the main CSS framework. Bootstrap is a

> *"Sleek, intuitive, and powerful front-end framework for faster and easier web development.project,"[42]*.

Besides supporting RWD, it also includes the various ready-built templates for tab control, form, etc. The project will also uses other custom written CSS files when custom styling is required.

### 5.2.3 Integrated Development Environment (IDE)

The default *Integrated Development Environment (IDE)* for ASP.NET development is *Microsoft Visual Studio* (VS) [76]. It is developed by Microsoft to support a broad range of application developments, ranging from simple standalone to large-scale enterprise solutions. VS is an excellent choice because it has built-in support for almost all aspects in software development. The project mainly utilize following features to ease development activities:

- Package management for third party libraries;
- Rapid code refactoring and design warning;
- Step through debugging;
- Unit test management and execution;
- User friendly code editor with responsive code completion;
- Source code version control; and
- Database management.

### 5.2.4 Relational Database Management System (RDBMS)

As specified in §4.3.3.2, this project employs RDBMS as its data storage method. The project does not explicitly constrain the RDMBS selection to a particular vendor. In fact, the RDBMS usage could be particularly different when viewed from the development stage and the deployment stage. In the development stage, the data is often dummy and easily generated. The data schema also changes rapidly as design decisions changed. The database's tables need to be constantly dropped, altered, recreated; and facilating testing. Such required functionality would be better implemented by a lightweight RDBMS; that would allow instant creation of a database from scratch. On the other hand, the performance

and functionality of RDBMS is more demanding when the system is deployed into the actual environment.

The RDBMS selected for the project is *Microsoft SQL Server* [62] and more specifically *SQL Server Express Local DB* [77], which was utilised during the development stage and the final full version at deployment stage. These are both supported by Visual Studio and can be easily integrated into the system. The Local DB maintains minimum configuration to start the database engine thus allowing the developer to focus on processing data rather than configuring the server. The full version of SQL Server has a wide range of tools to manage, monitor and analysis data stored in the database. The database is used to hold restaurant operational data when deployed to the production environment.

An interesting note regarding the choice of RDBMS is that this could be easily switched to other another vendor if necessary. This is because the ORM solution is used to construct the database schema and perform queries. *Entity Framework* (EF), the ORM solution of the .NET Framework, facilitates various vendor integrations [78] and thus allows the data store to seamlessly switch to other vendor solution such as *MySQL* or *Oracle DB*. The system owner could consider switching the vendor of RDBMS if necessary; once the system has been deployed.

### 5.2.5 Continuous Integration (CI) Software

As discussed in §5.1, the project will require CI software for source control and manage software builds. The source control software is mainly used to manage the different versions of source codes for the prototype software – as it is being developed. This is particularly important in XP because new features are introduced to the code base rapidly. Versioning source code allows developers to roll back software changes in the event of these changes affect the existing system – or present release versions. It also helpful when the developer is fixing errors, indicating what changes introduced bugs and failed builds – if test harnesses are utilised. Although not relevant to this project, the software is also valuable to the project when the project involves multiple developers. The software provides friendly UI to manage and resolve conflicts between concurrent editing on same version of source code. There are many source control software, ranging from open source solution such as *Apache™ Subversion (SVN)* [79] and *Git* [80], to the proprietary solution *Team Foundation Server* [81].

Managing the software builds is another scenario where CI software was found to be useful. In XP development, tests should be passed before committing new changes to the code base. However, building and testing all the software packages of the system for small changes could be time consuming. Build management software addresses this issue by automating the build process when new changes are committed. The built software then verifies these changes by a series of automated tests. This helps to identify compiling error and run-time errors and thus provide meaningful indications if the new changes work.

WCRS employs *Team Foundation Service* (TFS) [82] as its CI software. TFS is a cloud-based solution for *Team Foundation Server* – which provides full support for application life cycle management. TFS allow developer to manage source code repository, plan agile process, and continuous delivery by automatic test execution (see Figure 5.1). VS have native support for TFS and thus, minimal effort is required in integrating them. The author can perform check-in and queue build directly from the IDE interface. Another primary factor that contributed to this decision is TFS has a free plan that allows collaboration up to five users. This implies that there is no cost involved in utilizing the software for this project. Since TFS requires minimal set-up and configuration, the author can rather focus on development activities and achieve better output. Screen shots of their usage are shown in Appendix N.

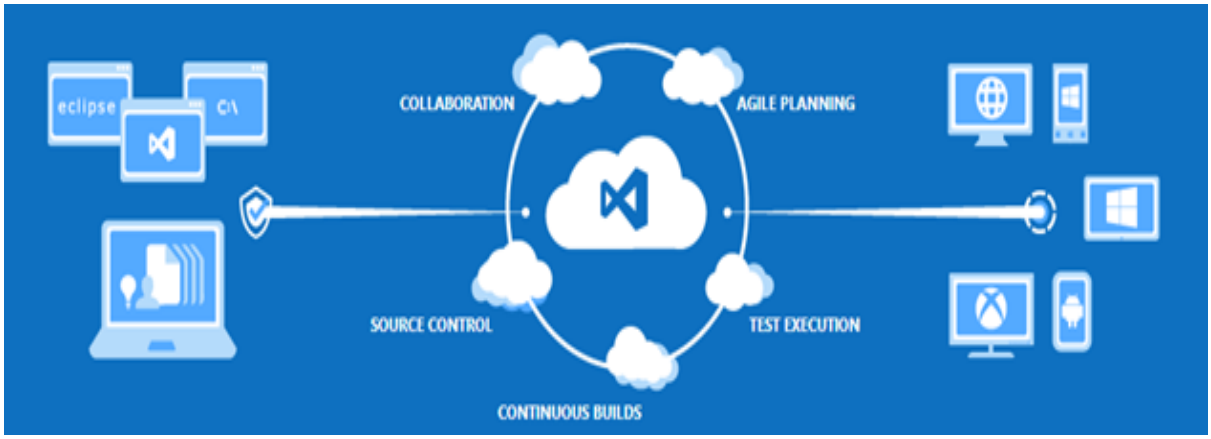



Figure 5.1: Overview of TFS Features [82].

## 5.3 Implementation Details

The implementation of the WCRS is best explained using the architecture layer design methodology established in §4.2.1.3. The *software architecture* is broken down to three major layers: *Presentation Layer (PL)*, *Business Logic Layer (BAL)* and *Data Access Layer (DAL)*. Each layer involves using different technology to solve their issues. In order to manage these layers, the software packages are structured identically. However, the *domain entity* classes of BLL, which will be used across all layers, are placed in another package.

The development and implementation of a specific requirement will involves all three layers. For instance, developing and implementing the *manage recipes* feature involves defining the *Recipe* entity class. This is then followed by configuring the ORM class to accept this entity. All the operational logics that is related to *Recipe* such as CRUD operations will be specified in the *RecipeServices* class. When the backend logic is ready to perform its function, the controller class and relevant view will be implemented to support user interaction. The view implementation is divided into client side and server side implementation. The server side implementations construct the relevant HTML views and send them to the client's web browser. At client side, JavaScript will be written if it requires dynamic functionalities or behaviours. The entire process is abstracted in Figure 5.2.

The next sections will explain the implementation details of each layer. In these sections, some of their implementation details will be depicted in the form of actual code snippet or algorithmic abstraction. Variables will be preceded by a $ symbol when abstraction form is used.

### 5.3.1 Data Access Layer (DAL) Implementation

WCRS requires a solution to manage its data access and persistence. The answer to these requirements will be *Entity Framework* (EF). EF is:

> *"an object-relational mapper (ORM) that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write," [78].*

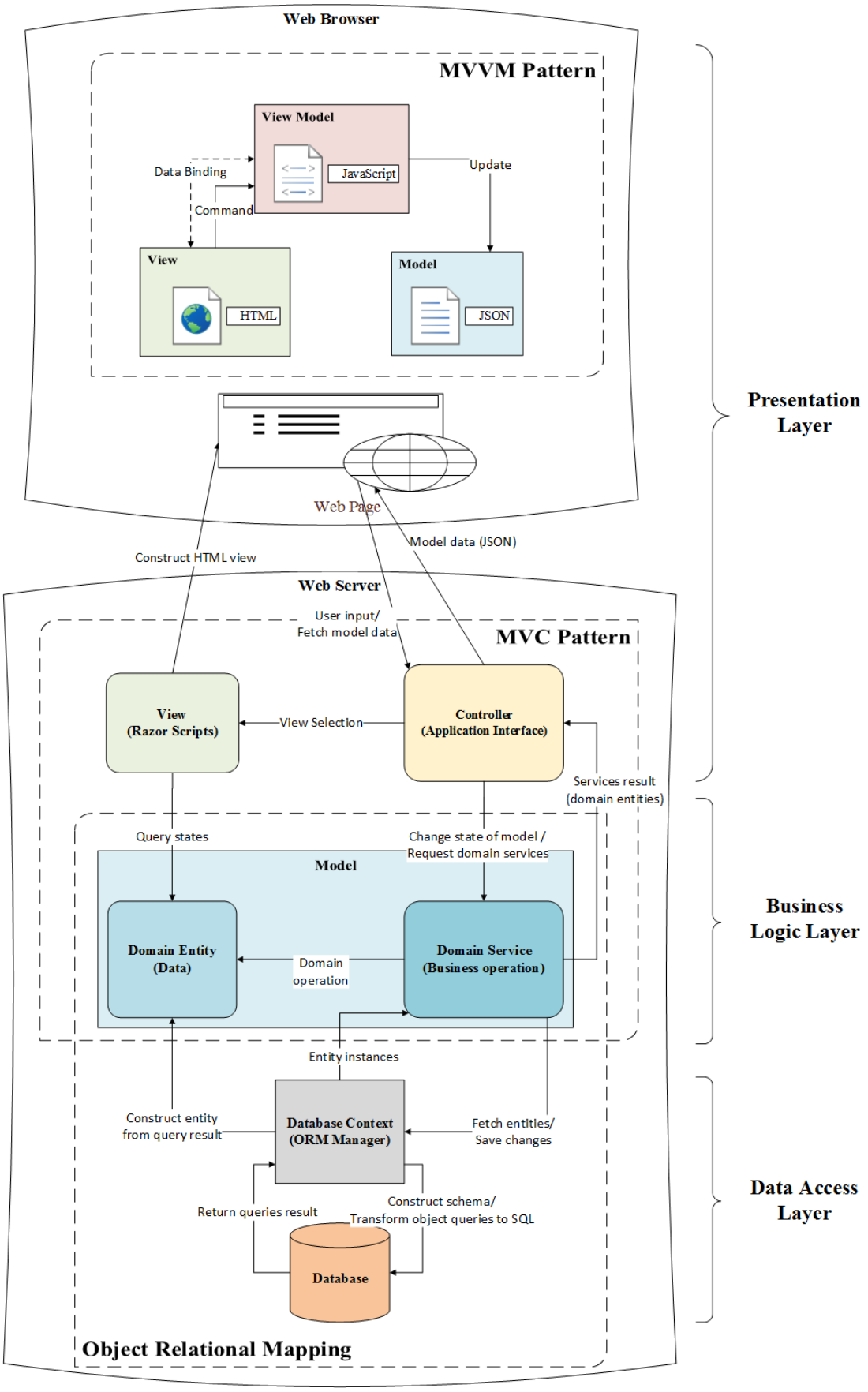



Figure 5.2: Overview of WCRS implementation.

The ORM solution is increasingly popular among software projects that need to integrate relational database into the OO environment. As discussed during §4.3.3.5, it helps to address the "*object-relational impedance mismatch*" problem. The problem is directly aligned to the different ways the OO model and Relation model access their data. The OO paradigm traverses objects via their references and relationships to other objects while the relational paradigm involves joining (or associating) data rows of tables [83]. In conventional applications without ORM, an additional layer of class wrappers named the *Data Access Object*, is used to retrieve the data row and convert it to the target object type. This creates a tight coupling between the domain entities and the persistence strategies.

Utilising the ORM solution reduces the large amount of code that is overhead when using the previous method, especially when implementing SQL functionality. ORM has clear advantages with respect to speeding up the development process by reducing the overload on writing SQL query. This allows developers to focus on the business problems rather than the repetitive CRUD logic in SQL. In addition, the ORM solution caches loaded entity in memory thus reducing the round-trip to the database.

#### 5.3.1.1 Using ORM for Data Access

Utilizing the EF in WCRS is a simple and straightforward step; though it requires a full comprehension of the EF. It starts by defining a class (i.e. *CrsContext)* that must inherit *DbContext* class of the framework – this associates the EF in the classes inheritance structure; and allows access to numerous methods, properties, and attributes in the *DbContext* class. For each domain entity class that need to be stored in the database, the *CrsContext* class needs a property of *IDbSet (*generic collection type) for that entity class (see Figure 5.3).

When the application is initialized at runtime, EF will navigate all the properties of *CrsContext* that are a type of *IDbSet* and create a SQL schema at the target database based on the convention discussed in §5.3.1.2 – a process called type discovery. Properties of *IDbSet* type will be instantiated as the *DbSet* class object when it is first accessed through *CrsContext* class. Example of the *CrsContext* class usage can be viewed from Figure 5.4. The *CrsContext* class will manage the database transaction and the connection when it is created and accessed. In WCRS, it is short-lived and will be disposed of after the database operations have been completed.

```
public class CrsContext : DbContext
{
    public IDbSet<Recipe> Recipes { get; set; }
    public IDbSet<RecipeMaterial> RecipeMaterials { get; set; }
    public IDbSet<RecipeCategory> RecipeCategories { get; set; }

    public IDbSet<Material> Materials { get; set; }
    public IDbSet<MenuGroup> MenuGroups { get; set; }

    public IDbSet<Table> Tables { get; set; }
    public IDbSet<Reservation> Reservations { get; set; }

    public IDbSet<Payment> Payments { get; set; }
    public IDbSet<Order> Orders { get; set; }
    public IDbSet<OrderItem> OrderItems { get; set; }

    public IDbSet<Staff> Staffs { get; set; }
    public IDbSet<Membership> Memberships { get; set; }
    public IDbSet<Role> Roles { get; set; }

    public IDbSet<SettingNameValuePair> SettingNameValuePairs { get; set; }

    // the rest are omitted...
}
```

Figure 5.3: Code Snippet of ORM class.

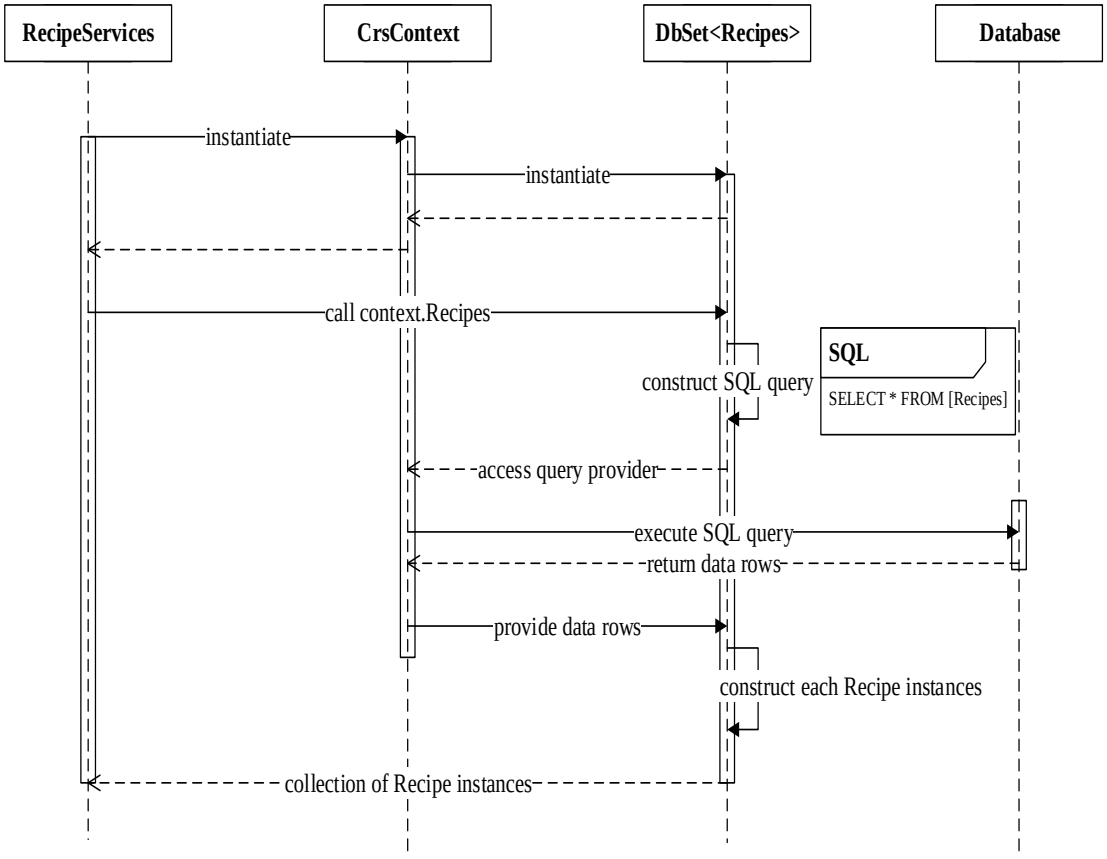



Figure 5.4: Sequence Diagram shows *CrsContext* usage flow.

Table 5.1 shows CRUD operations derived from a *DbSet* class of the *Recipe* entity. Most of the operations can be perform directly on the class instance and pass back to the *DbContext* class. When the operations are completed, the *SaveChange* method is called to commit every change to the database that that has been performed during the query. All of the changes will be translated into SQL statements and then executed on the database (see Listing 5.1) – it is done automatically in EF. These operations are often invoked from the domain services of BLL.

Table 5.1: Examples database operations with EF.

| **Database Operation** | **Query Syntax** |
|---|---|
| **Create a new record** | `$context.Recipes.Add($recipe);` |
| **Update a record** | `$context.Update($recipe);` |
| **Search records** | `$context.Recipes.Find($id);`<br><br>`$context.Recipes.Where(r => r.Title == $titleName);` |
| **Delete a record** | `$context.Recipes.Remove($recipe);` |
| **Commit changes** | `$context.SaveChanges();` |
| **Join child tables** | `$context.Recipes.Include(`<br>`r => r.RecipeMaterials.Select(x => x.Material));` |
| *$context: an instance of **CrsContext** class<br>*$recipe: an instance of **Recipe** class | |

```
exec sp_executesql

N'insert [dbo].[Recipes]([Title], [Serving], [Description], [Price],
[RecipeCategoryId], [MenuGroupId])

values (@0, @1, @2, @3, @4, null)

select [RecipeId]

from [dbo].[Recipes]

where @@ROWCOUNT > 0 and [RecipeId] = scope_identity()',

N'@0 nvarchar(max) ,@1 int,@2 nvarchar(max) ,@3 decimal(18,2),@4 int',

@0=N'$title', @1=$serving, @2=N'$description', @3=$price, @4=$categoryid
```

Listing 5.1: Generated SQL queries for adding a Recipe record.

#### 5.3.1.2 Schema Management

The design of database schema presented in the §4.3.3.6 is the result of several revisions. One of the most challenging problems in working with the database is to drop and recreate the necessary table when new fields or relationships are introduced. The scenario is worst when the table is having multiple relationships with other tables. Substantial effort will be spent on configuring and updating these schemas rather than solving the business problem at hand.

```
public class Recipe
{
    // Primary key
    public int RecipeId { get; set; }

    // Navigation Property
    // one to many relationship
    public virtual ICollection<RecipeMaterial> RecipeMaterials { get; set; }

    [NotMapped] // decorate attribute to ignore by framework
    public decimal Profit
    //... the rest of property omitted
}

public class RecipeMaterial
{
    // primary key
    public int RecipeMaterialId { get; set; }

    // foreign key
    public int RecipeId { get; set; }
    public int MaterialId { get; set; }

    // many to one relationship
    public virtual Recipe Recipe { get; set; }
    // many to one relationship
    public virtual Material Material { get; set; }

    //... the rest of property omitted
}

public class Material
{
    // primary key
    public int MaterialId { get; set; }
    // one to many relationship
    public virtual ICollection<RecipeMaterial> RecipeMaterials { get; set; }

    //... the rest of property omitted
}
```

Figure 5.5: Code First Convention for Recipe related classes.

To address this issue, the project utilizes *code first development* approach supported by EF. *Code first development* approach allows developer to define all the database entities in the form of object classes and their relationships are embedded in the methods and logics of these classes. EF will analyse properties and the reference rules of the classes and create the corresponding database schema through conventions that are defined in EF. For instance, the convention automatically discovers the primary key and the relationships based the

naming convention(s) such as the field with “Id” name and the reference type as shown in Figure 5.5.

In order to facilitate entity class or database schema change(s), the project configures the database initialisation to drop and recreate mode. Again, the framework will manage the construction of the updated schema. However, if the previous schema needed to be maintained and achieved (or stored), the project addresses the migration of schema by storing the previous state in VS with EF’s migration features. This allows the project to rollback to the previous version of schema if required. In most case, all data in the database are dummy records during the development thus, it is safe to drop the database and recreate a new one.

### 5.3.2 Business Logic Layer (BLL) Implementation

BLL is where all the business rules and business process logic reside. There are mainly two kinds of business logic [57]:

i) *domain logic* that deals with the problem of the domain; and
ii) *application logic* that deals with application responsibilities.

An example of *domain logic* could be the strategy of computing price, while *application logic* may refer to particular workflow such as initializing components and the algorithmic sequence of method calls. One benefit of layered architecture is that it facilitates separating both logics (*domain & application logic*) and allows them to be reused independently. For instance, this project utilises web interface for the system, hence it is possible to develop another mobile or desktop application that is directly connected to the business logic layer and reuse the functions exposed by this layer.

They are two main components in BLL of WCRS: *Domain Entities* and *Domain Services*. The *domain entity* represents the data content of the real life business object. On the other hand, *domain service* encapsulates a set of operations and business logic to the domain entities to provide functionalities to the system. The primary objective of this segmentation is to decouple the entity object from the business logic. In ever-changing business, entity object content and business logic may change rapidly, for example, new fields are introduced to the object or a new procedure is added to the business process. Combining them under the same model will result in a class that is cluttered and difficult to change. Besides, the entity object will also be less reusable across all layers [57].

#### 5.3.2.1 Domain Entities

Implementing *domain entities* is often done by writing a class with many important field that describes the business characteristic. For instance, the *Recipe* class refers to the requirement of recording a food dish's name(s) and its (their) ingredients. It could belong to particular menu group such as Starter, Main Course, etc. These are initially designed utilising *CRC cards* (abstraction technique) and then the fine details in a *Class Diagram*; hence translating them into C# classes is straightforward. However, they also should take into consideration the schema creation as specified in Figure 5.5.

In addition to that, there are validation rules that could be integrated within the entity to accommodate DRY principle[15]. For instance, some of the fields should not be null and checking them at multiple places in code is redundant. Hence, *Data Annotation* [16] [84] is utilized to define validation rules and expose them together with the entity to the client that consumes them, e.g. PL and DAL. The field that requires validation will be aligned to an attributes with particular validation rules. Figure 5.6 showing that the *Table* class has an *Identifier* field that is not null (required when instantiated) and has a max length of 10 characters.

```
public class Table
{
    [Key]
    public int TableId { get; set; }

    [Required]
    [StringLength(10)]
    public string Identifier { get; set; }

    [Required]
    public int Seats { get; set; }

    // the rest of properties omitted..
}
```

Figure 5.6: Code Snippet of Table class.

EF will check if the validation rules are satisfied before trying to save them to the database. At PL, ASP.NET will generate the corresponding validation rules on the client side view. All of these steps simplify the essential data validation and in turn enhance the data integrity. When these entities are implemented, the next step would be implementing the *domain services* for these entities.

---

[15] Don't Repeat Your Self, a principle of avoiding duplication of information.
[16] A NET Framework's feature that allow decorating the class properties with validation rules, display format and data modelling (e.g. editable).

#### 5.3.2.2 Domain Services

Generally, implementing *domain services* in WCRS is about defining set of operations. The operations needed by the services are often reflected by the client who depends on it. In other words, they exposed the necessary business functions to the PL that interacting with these services. In any data-oriented application, most of the business operations will involve CRUD operations. However, in a more complex scenario, this will need sophisticated data processing logic. This section will concentrate on several important algorithmic of the business operations.

Most of the data processing regarding the CRUD operations involves integrating the business rules. Business rules are often aligned to domain problems and usually involve checking multiple domain entities at the same time. For example, it is meaningless to create redundant *recipe category* with same name. Listing 5.1 depicts the algorithm required to process (and check for) the redundant name in the *recipe category*. The algorithm is applicable to creation of *Table* and *Staff* entity. The different being identifier of *Table* and username of *Staff* is checked instead.

```
SET $name to name of new recipe category
IF $name existed in database THEN
        Show duplicate name exception message
ELSE
        Add the new recipe category to database
        Commit changes
ENDIF
```

Listing 5.2: Algorithmic for adding new recipe category.

In addition to this, several non-trivial operations in *domain services* need to be addressed. One of the features of WCRS is allowing customisation of menu availability. The system provides flexible customizing whether the menu is currently active, available for selected days, and within a particular time range. When no values are provided for days and time fields, we can assume that they are available all time. When a waiter needs to place an order for a customer, the system will retrieve the available menus based on the criteria and return every recipe under retrieve menus. The algorithmic for this operation is shown in Listing 5.3.

```
SET $day to current day of week
SET $currentTime to current time
SET $recipes to empty array
FOREACH $menu IN database
        IF $menu is mark as active THEN
                IF $menu available days is 'all' OR $menu available days contains $day
        THEN
                        IF $menu available time period is empty
                            OR $currentTime within available time period THEN
                                IF $menu available day contains $day THEN
                                        Add recipes of $menu into $recipes
                                ENDIF
                        ENDIF
                ENDIF
        ENDIF
ENDFOR
RETURN $recipes
```

Listing 5.3: Algorithmic for retrieving available menu recipes.

Handling order submission is a complex operation that requires intensive data processing on multiple entity. When the system receives an order submission, it must evaluate whether the current material stock level can handle this order request. If the current stock level is unable to fulfil the order request, the system will return the cook-able quantity for each recipe to allow the waiter to informs the customer. If the order is submitted successfully, the system will then update the table status to busy, deduct the material quantity and send a notification to kitchen. These algorithms are shown in Listing 5.4 and Listing 5.5.

Computing for payment is another essential part of the order processing cycle. When customers request the bill, the system needs to generate the total amount to be paid. The figure includes the VAT rate and the service charge rate if applicable. The system will first read the parameter (stored in the database) specified by the restaurant regarding the VAT and the service tax rate. Following this, it constructs relevant details (how the bill is layered and presented) and then dispenses the bill. The algorithm to compute the payment is shown in Listing 5.6.

```
SET $uncookableRecipes[] to empty hash table
FOREACH $orderItem IN order
    SET $recipe to recipe specified by $orderItem
    SET $minimunPortion to max integer
    FOREACH $material IN materials used by $recipe
        SET $cookablePortion to material available quantity / recipe usage quantity
        IF $cookablePortion < $minimunPortion THEN
            SET $minimunPortion = $cookablePortion
        ENDIF
    ENDIF
    SET $availableQuantity to $minimunPortion
    IF quantity of $orderItem > $availableQuantity THEN
        Add $availableQuantity for $recipe into $uncookableRecipes
    ENDIF
ENDFOR
RETURN $uncookableRecipes
```

Listing 5.4: Algorithmic for checking cook-able quantity of ordered recipes.

```
SET $currentTime to current date and time
CALL CheckCookableQuantities WITH order RETURNING $uncookableRecipes
IF $uncookableRecipes has any element THEN
    Show error message and $uncookableRecipes for the order
ELSE
    SET order submission date to $currentTime
    SET order last modified to $currentTime
    Add the order into database
    SET table status of the order table to "Busy"
    Deduct the consumed material quantities from database
    Commit changes
    Notify kitchen about new order
ENDIF
```

Listing 5.5: Algorithmic for the order submission.

```
READ setting
SET $vatRate to setting vat rate
SET $serviceRate to setting service rate
SET $orderTotalAmount to 0
FOREACH $orderItem IN order
        SET $subTotal to recipe price * quantity of $orderItem
        SET $orderTotalAmount to $orderTotalAmount + $subTotal
ENDFOR
SET $vatAmount to $vatRate * $orderTotalAmount
SET $serviceAmount to $serviceRate * $orderTotalAmount
SET $grandTotalAmount to $orderTotalAmount + $vatAmount + $serviceAmount
Assign $orderTotalAmount,, $vatAmount , $serviceAmount and $grandTotalAmount to
bill
```

Listing 5.6: Algorithmic for computing payment amount for bill.

### 5.3.3 Presentation Layer (PL) Implementation

Implementation of the PL mainly focuses on the UI and presentation logic. The UI is often the main concern of users when using a software product. It usually gives the user the first impression as to whether they can accomplish their goal easily – which is termed usability. The UI of any good software should guide the users through series of tasks to achieve their goal. Initially this is addressed in §4.4.1, which formed the ideas regarding the required layout and content of UI. In this section, the design idea will be evolved into an actual UI for WCRS.

Construction of web application UIs often involves various kind of files. It defines the layout of the content through HTML, uses CSS for styling the look and feel of the UI, and embeds JavaScript to introduce dynamic behaviour on the client side. These are increasingly difficult to manage as the system grows. In addition, the same data could be viewed in different way depending on user needs, making the application further hard to change when UI and data are tightly coupled.

To address these issues, the system utilized two *Presentation Separation* patterns: *Model-View-Controller* and *Model-View-View Model*. The term, *Presentation Separation* pattern, is just a generalized name for the all the design patterns (see Figure 5.7) that encourage separation of the UI related logic, from application logic and data [85]. At this point, an interesting question may arise: Why are two patterns of similar goals required in this project?

To answer this question, the focus areas of both patterns are explained in the following sections.

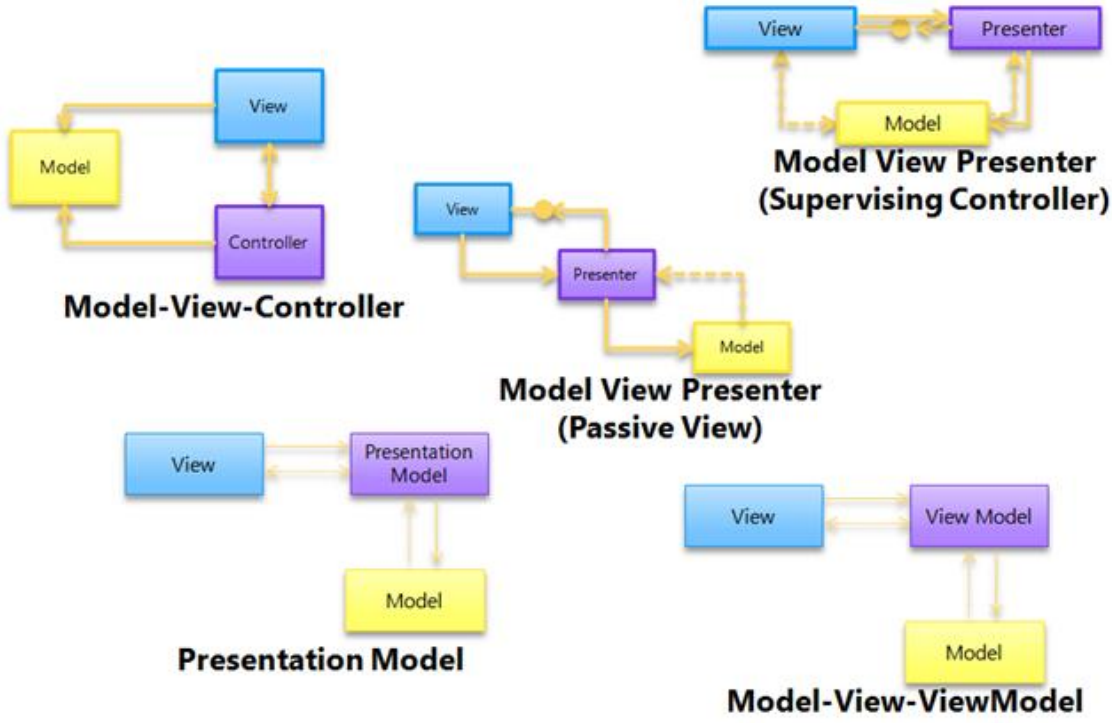



Figure 5.7: An overview of Presentation Separation patterns [85].

#### 5.3.3.1 Model-View-Controller (MVC)

Since Trygve Reenskaug introduced MVC pattern for Smalltalk application in late 1970 [86], it has become one of the major practises in software engineering history. As discussed in §2.2.1, MVC pattern addresses the responsibilities of three major components in UI construction, they are as below:

I. *Model* refers to data and behaviour of the application. It is responsible for providing the current state of its data and handle instructions for states change according the client requests;
II. *View* is the user interface that presents the state of data and manages the way they are presented; and
III. *Controller* captures the user inputs from user interface and manages the flows to update the model state and view information.

The clear segregation of their responsibilities results in several benefits [87]:

- The logics of presentation (view), input (controller) and business process (model) could be changed and allowed to evolve independently, hence achieving separation of concerns;
- The view is decoupled from the model allowing multiple ways to present the same data that accommodate user concerns; and
- Loose coupling between the view and the controller also enhance the testability of the application.

PL of WCRS adheres to this pattern but has a richer model as it separate its concerns into basic business logic (*domain service*) and data (*domain entity*) as shown in Figure 5.8. ASP.NET MVC has a built-in infrastructure to implement the necessary *Controller* class and construct the *View*. The user inputs could be viewed as HTTP requests that are sent from client web browser. The framework will route the request based on the request URL to the correct *Controller* class.

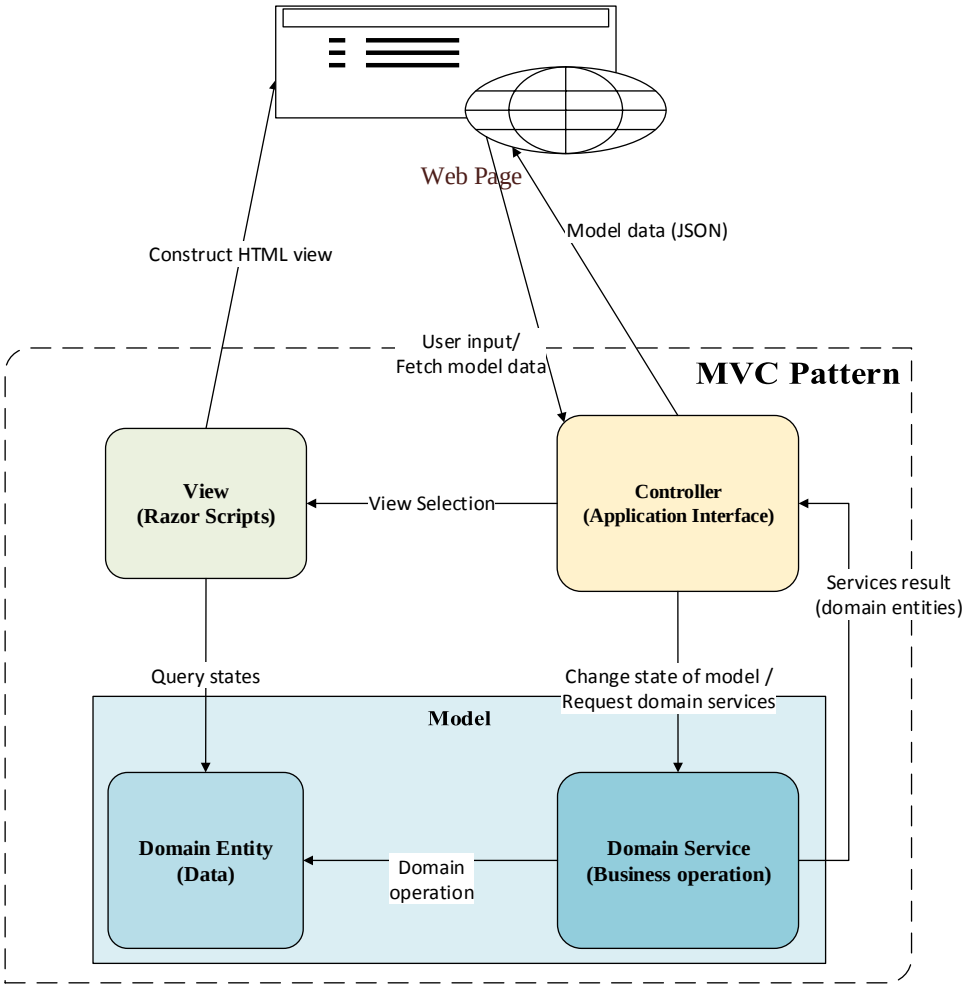



Figure 5.8: MVC pattern at Presentation Layer.

The request URL follows *"http://domain/{controller}/{action}/{parameters}"* pattern where

- *controller*, is the name of controller;
- *action*, refers to the request that user initialise such as view details, create new, etc.; and
- *parameters*, are the values that would needed by controller to perform the action such as order id.

When a HTTP requests is routed to the *Controller* class, it finds the appropriate method to execute based on the pattern described above. The *Controller* class can provide overloaded version of the method depending on whether the request type (GET or POST). This is illustrated in Figure 5.9**,** which shows *OrderController* class that maps the URL to the methods in the class. Next, the controller performs the method by invoking the *domain services* to retrieve or update the *domain entities*. It then selects the *Views* to be presented to the user. The selection of the view also follows the convention of the method name. For instance, the *Details* method will expect a file named "*Details.cshtml*" at the "Order" sub-directory within the "Views*"* directory at the application file structure. *Razor* code (view engine) is then used to construct the View content based on selected template and supplied data. The data refers to the *domain entity* of BLL in this context.

Figure 5.10 shows how Razor allows natural binding of the model property to the display HTML elements. In addition, Razor provides flexibility of presenting the property, whether it is textbox, label or other. If the HTML element is an input, the validation rules specified during domain entitiy implementation (see §5.3.2.1) also applied to the field. This allow validation of user input to be seamlessly done at the client side. Besides, the framework also provides the default validation message based on the data type of the field. Figure 5.11 shows an example of HTML form implementation to capture details when creating a recipe. It shows the creation of field label, input and validation could be bind to a particular property of the entity. Combination of these approaches has significantly simplified the creation of view and maintaining their logic.

```
public class OrderController : Controller
{
    private OrderServices _services;

    //
    // GET: /Order/Details/5
    public ActionResult Details(int id)
    {
        Order order = _services.FindOrderById(id);
        if (order == null)
        {
            return HttpNotFound();
        }
        return View(order);
    }

    //
    // POST: /Order/Create
    [HttpPost]
    public JsonResult Create(Order order)
    {
        // omitted...
    }
}
```

Figure 5.9: Code snippet of the Order Controller class.

```
@model CRS.Core.OrderModule.Order

<h2>Details</h2>

<fieldset>
    <legend>Order @Html.DisplayFor(model => model.OrderId)</legend>
    <dl>
        <dt>
            @Html.DisplayNameFor(model => model.Table.Identifier)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Table.Identifier)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.Status)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Status)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.OrderItems)
        </dt>

        <dd>
            @Html.Partial("_OrderItemsPartial", Model.OrderItems)
        </dd>
    </dl>
</fieldset>
<p>
    @Html.ActionLink("Back to List", "Index")
</p>
```

Figure 5.10: Code snippet of Order details view.

```
@model CRS.Core.RecipeModule.Recipe

@Html.HiddenFor(model => model.MenuGroupId);
<div class="control-group">
    @Html.LabelFor(model => model.Title, new { @class = "control-label" })
    <div class="controls">
        @Html.EditorFor(model => model.Title)
        @Html.ValidationMessageFor(model => model.Title, null, new { @class = "help-inline" })
    </div>
</div>

<div class="control-group">
    @Html.LabelFor(model => model.Serving, new { @class = "control-label" })
    <div class="controls">
        @Html.EditorFor(model => model.Serving)
        @Html.ValidationMessageFor(model => model.Serving, null, new { @class = "help-inline" })
    </div>
</div>

<div class="control-group">
    @Html.LabelFor(model => model.RecipeCategoryId, "RecipeCategory", new { @class = "control-label" })
    <div class="controls">
        @Html.DropDownList("RecipeCategoryId", String.Empty)
        @Html.ValidationMessageFor(model => model.RecipeCategoryId, null, new { @class = "help-inline" })
    </div>
</div>

<div class="control-group">
    @Html.LabelFor(model => model.Description, new { @class = "control-label" })
    <div class="controls">
        @Html.TextAreaFor(model => model.Description, new { @class = "input-xlarge" })
        @Html.ValidationMessageFor(model => model.Description, null, new { @class = "help-inline" })
    </div>
</div>

<div class="control-group">
    @Html.LabelFor(model => model.Price, new { @class = "control-label" })
    <div class="controls">
        @Html.EditorFor(model => model.Price)
        @Html.ValidationMessageFor(model => model.Serving, null, new { @class = "help-inline" })
    </div>
</div>
```

Figure 5.11: Code snippet of Create recipe view.

In summary, the implementation of WCRS can accommodate changes easily by adopting the MVC pattern. The clean separation of concerns also allows individual part to be tested easily hence simplify the debugging experience. However, MVC pattern may address the problem of view creation on the server side but not at the client side. As specified in §5.2.2, once the view is served to the client side. JavaScript is used to introduce behaviours to the view. The JavaScript often needs to target a particular HTML element in the view and changes to view structure will easily break existing implementation. The problem of tight coupling between presentation and application behaviour still exits if they are mixed together at the client side. Hence, the second design pattern, MVVM pattern is used to address this issue.

#### 5.3.3.2 Model-View-ViewModel (MVVM)

MVVM general goal and concepts similar to MVC are to enforce better *separation of concerns* among UI components and allows them to change easily. John Grossman first introduces MVVM for building WPF[17] applications in his blog [88]. Its name implies that they are similar to MVC but different in the sense that presentation logic and data is encapsulated in *ViewModel* (VM) rather than *Controller*. At client side of PL, they all also has different semantic compared to MVC. Figure 5.12 shows an overview of MVVM pattern and description of its component are listed as below:

I. *Model* refers to JSON (JavaScript Object Notation) that mirrors the domain entity of server side;
II. *View* is the generated by HTML through the view engine of the server; and
III. *ViewModel*, written in JavaScript, is an abstraction of view that consists of View's state and behaviour [89]. It exposes the model's properties, commands and additional states to the view.

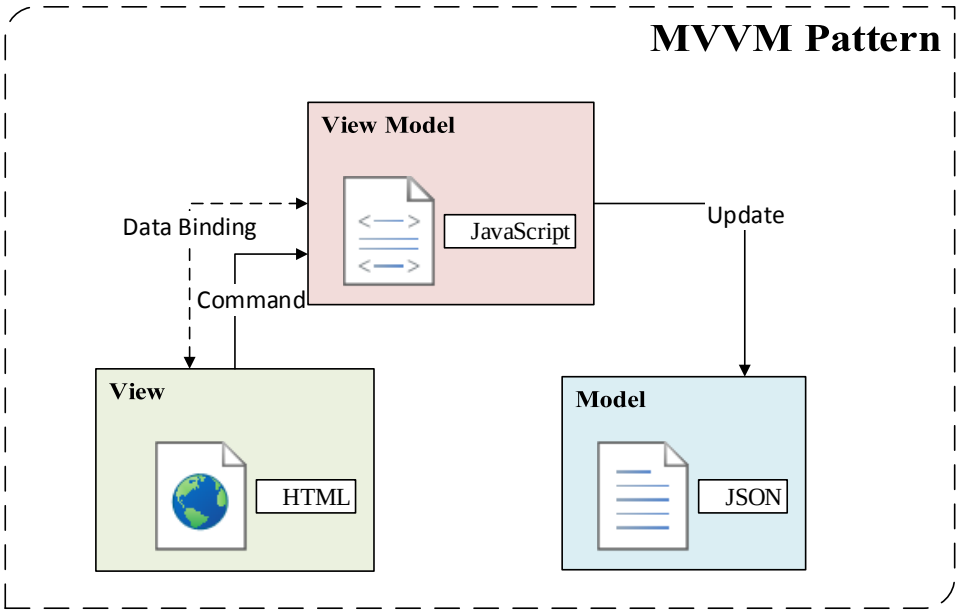



Figure 5.12: MVVM pattern at Presentation Layer.

One of the important differences between the MVC and the MVVM is that *ViewModel* does not directly reference the *View* as managed by the *Controller*. Instead, it leverages the data binding technology to bind the view to properties and functions of *ViewModel*. The

[17] Window Presentation Foundation, a programming model for Windows-based applications.

properties often consist of the data contained in the model and other states specific to the view [89]. When the properties change, the *View* that binds to the properties is updated. The functions of *ViewModel* are a set of commands that reflect application behaviour, for instance, adding a new order item to the order. In short, the state synchronization and command between the *View* and *ViewModel* are handled automatically by the data binding technology. Hence, it is a key enabler of this pattern [90].

At the client side of PL in WCRS, some of the *Views* need to be kept synchronized with the data from the server. One of the most relevant *View* is the *kitchen view*. The chef needs to update the status of the Order for following purpose:

i) Inform the waiter that the order is ready to be served; and
ii) Warn other chefs that the order is currently being prepared and avoid redundant jobs.

Apparently, MVVM is the ideal solution to this requirement considering the problems it can solve.

In order to apply MVVM pattern at the client side of PL, the project utilizes *Knockout.js* [91], a library whose aim is to:

> *"simplify dynamic JavaScript UIs by applying the Model-View-View Model (MVVM) pattern," [91]*.

The library utilizes the Observer pattern[18] to help UI stay in sync with the data model of *View* [92]. It relies on declaration of *ViewModel*'s properties as:

- *Observable*, used to bind property that has a single value, such as text or numeric value;
- *Observable Array*, used to bind property that is a collection or array of values; and
- *Computed Observable*, used to bind property that needs to be updated when others depending observable changes. For instance, the total count of order items changes when new item added to order.

Figure 5.13 shows how the *KitchenViewModel* leverages *Knockout.js* to update UI dynamically based on the status of the Order. It contains two main properties: a list orders and their total count. In addition, the *ViewModel* exposes the *Change Status* function to update the order status. The *ViewModel* retrieves the orders data from the server through

[18] A software design pattern that utilize the public-subscribe model for communication among interested party.

Ajax. These data are exactly one to one mapping with the domain entity from BLL (including their hierarchy). Hence, the order data is are actually the *Model* in this pattern. When the *ViewModel* receives the data, it converts the necessary properties to observable to update them dynamically.

```
var KitchenViewModel = function() {
    var self = this;
    self.Orders = ko.observableArray([]);
    self.TotalItems = ko.computed(function () { return self.Orders().length; });

    self.initialize = function () {
        $.getJSON('/Order/GetKitchenOrders', {},
        function (data) {
            for (var i = 0; i < data.length; i++) {
                data[i].Status = ko.observable(data[i].Status);
                data[i].LastUpdatedStaff = ko.observable(data[i].LastUpdatedStaff);
            }
            self.Orders(data);
        });
    };

    self.ChangeStatus = function (order, status) {
        if (order.Status() === status) {
            return;
        }
        var postData = { orderId: order.OrderId, status: status };
        $.ajax({
            type: 'POST',
            dataType: 'json',
            url: "/Order/UpdateStatus",
            data: JSON.stringify(postData),
            contentType: "application/json; charset=utf-8",
            success: function (data) {
                if (data.Success) {
                    toastr.success(data.Message || "Success");
                    order.Status(status);
                    if (status === "Completed") {
                        self.Orders.remove(order);
                    }
                } else {
                    toastr.error(data.Message || "Server Operation Failed");
                }
            },
            error: function (err) {
                toastr.error(err || "Server Operation Failed");
            },
        });
    };

    // ....other functions omitted
};

$(function() {
    var viewModel = new KitchenViewModel();
    ko.applyBindings(viewModel);

    // ....other functions omitted
});
```

Figure 5.13: Code snippet shows the KitchenViewModel.

```
<h2>Kitchen</h2>
<div class="row-fluid">
    <h4>Current Orders: <span data-bind="text: TotalItems"></span></h4>
    <ul class="thumbnails" data-bind="foreach: Orders">
        <li class="span4 thumbnail">
            <div data-bind="css: Status">
                <dl>
                    <dt>
                        @Html.DisplayNameFor(model => model.OrderId)
                    </dt>
                    <dd data-bind="text: OrderId"></dd>

                    <dt>
                        @Html.DisplayNameFor(model => model.Table)
                    </dt>
                    <dd data-bind="text: Table ? Table.Identifier : '<Counter>'"></dd>
                    <dt>
                        @Html.DisplayNameFor(model => model.OrderItems): (<span data-bind="text: OrderItems.length"></span>)
                    </dt>
                    <dd>
                        <ul data-bind="foreach: OrderItems">

                            <li>
                                <strong><span data-bind="text: Recipe.Title"></span>
                                    <span class="pull-right" style="padding-right: 30px" data-bind="text: Quantity"></span>
                                </strong>
                                <!-- ko if: $data.Comment -->
                                <br />
                                <small>Note: <span data-bind="text: Comment"></span></small>
                                <!-- /ko -->
                            </li>
                        </ul>
                    </dd>
                    <dt>Last Updated
                    </dt>
                    <dd>
                        <span></span>by <span data-bind="text: LastUpdatedStaff().UserName"></span>
                    </dd>
                </dl>
                <div class="btn-group">
                    <button class="btn"
                        data-bind="css: { 'btn-primary': Status() == 'New' }, click: function () { $root.ChangeStatus($data, 'New') }">
                        New
                    </button>
                    <button class="btn"
                        data-bind="css: { 'btn-primary': Status() == 'Preparing' }, click: function () { $root.ChangeStatus($data, 'Preparing') }">
                        Preparing
                    </button>
                    <button class="btn"
                        data-bind="css: { 'btn-primary': Status() == 'Completed' }, click: function () { $root.ChangeStatus($data, 'Completed') }">
                        Completed
                    </button>
                </div>
            </div>
        </li>
    </ul>
</div>
```

Figure 5.14: Code snippet shows the Kitchen view.

On the *View* side, these properties are bound to the relevant HTML elements through the "*data-bind*" attribute as shown in Figure 5.14. The binding could be directly presenting the value of the *observable* property such as "*text*" or "*css*". It is also possible to introduce control flow such as "*foreach*". The "*foreach*" binding is a powerful binding that loops every element of *observable array* and creates the appropriate HTML elements for the property in the array. When the item is removed from the array, so are the HTML elements. The "*click*"

binding deals with the click event on the HTML element and maps to the corresponding command (e.g. Change order status).

With this kind of binding, the *ViewModel* does not need to know about the structure of the *View*. The way of presenting the *View* could be changed any time and new behaviours can be added to *ViewModel* easily. It simplifies the development process without writing boilerplate code to synchronize the view. In addition, the application is capable of providing an interactive and richer user experience.

#### 5.3.3.3 Remote Procedure Call (RPC) and Server Push

One of the important considerations at the client side processing is getting data for the view and sending data back to the server. In most cases, they are done through initializing a RPC with the web server. RPC is process of sending a request with parameters over the network to another environment where the required procedure will executed, the result is then returned to the caller. While the caller process is waiting the result, other processes can continue to execute. RPC is necessary in WCRS because it is constantly committing data to the server based on user inputs.

The RPC is often use as pull model – which the request is initiated by the client (web browser). In order to obtain the latest data from the server, the client constantly needs to start a new request to pool server data. As the number of clients increase, significant resources of the server will be consumed. To address this issue, the *Server Push* approach is adopted. *Server Push* is a technology that allow the server to push data without required to start a new connection [93]. It mainly operates around the publish-subscribe model to deliver data to interested clients. In combining both RPC and *Server Push* approach, the system could achieve real time communication among the connected clients. This is particularly useful for order notification among waiters and chefs. In WCRS, these approaches are achieved by *Ajax* and *SignalR* respectively.

*Ajax* is a RPC technique which enables the JavaScript client to send and retrieve data without reloading the web page. The request is sent through JavaScript's *XMLHttpRequest* (XHR)[19] object and the server response is processed as an asynchronous call back [93]. Depending on the call back result, the client can then proceed to update the UI and display dynamic

[19] An API of web browser for exchanging data with server.

content. The use of Ajax should significantly improve the performance of application since it avoids the cost of full page reloading. Moreover, the UI is not blocked from user interaction since it is executed asynchronously, resulting in a more responsive user experience. The system uses *Ajax* in several places that required communication across domain entities, for instance: adding a new recipe to the menu and associating new material to a recipe. These often require multiple adding and/or deleting actions from the user. Reloading the whole page for each actions will definitely damage the system performance. In fact, Figure 5.14 demonstrated the *Ajax* usage when updating order status.

*SignalR* [94] plays a very important role aligned to realizing real-time communication between server and browser. *SignalR* is a library for ASP.NET that simplifies the creation of server push components at both client and server side. It has a simple API to enable the user to define RPCs that call JavaScript functions from the server-side code [95]. *SignalR* leverages several protocols to facilitate its communication. HTML5 transports protocols selected when client browser support HTML5 standard, as described in following [95]:

- *WebSocket*, establishes persistent and full-duplex connection between the client and server; and
- *Server Sent Events*, maintains the connection with server for push notification.

*SignalR* falls back to Comet transports [20] when HTML5 is not supported. These mainly rely on the browser to maintains long-held HTTP request with the server and are identified as below [95]:

- *Forever Frame*, creates hidden IFrame that maintains a one-way real-time connection from server to client; and
- *Ajax long pooling*, polls the server with a request that stays alive until server responds.

The order notification in WCRS heavily relies on *SignalR* to push the new order notice and status change notification to the waiter and chef. It starts by defining the order received functions at the client side as shown in Figure 5.15. At the server side, an *OrderHub* class that handles client subscription is defined. When a new order is received at the

[20] An approach of web application model that allows server pushing new data to client as opposed to explicitly polling data from server.

*OrderController*, the *OrderController* will access the Hub class and invoke the services defined at client side (see Figure 5.16).

```
var hub = $.connection.order;

hub.client.orderReceived = function (order) {
    toastr.info("New order (" + order.OrderId + ") received!");
    order.Status = ko.observable(order.Status);
    order.LastUpdatedStaff = ko.observable(order.LastUpdatedStaff);
    viewModel.Orders.push(order);
};

hub.client.statusChanged = function (order) {
    var match = ko.utils.arrayFirst(viewModel.Orders(), function (item) {
        return order.OrderId === item.OrderId;
    });
    if (match != null) {
        toastr.info("Order (" + order.OrderId + ") status changed to  <strong>" + order.Status + "</strong>");
        if (order.Status === "Completed") {
            viewModel.Orders.remove(match);
        }
        match.Status(order.Status);
        match.LastUpdatedStaff(order.LastUpdatedStaff);
    }
};

$.connection.hub.start().done(function () {
});
```

Figure 5.15: Code snippet shows SignalR client side implementation.

```
[HubName("order")]
public class OrderHub : Hub
{
}

public class OrderController : Controller
{
    protected readonly Lazy<IHubContext> OrderHub =
        new Lazy<IHubContext>(() => GlobalHost.ConnectionManager.GetHubContext<OrderHub>());

    //
    // POST: /Order/Create
    [HttpPost]
    public JsonResult Create(Order order)
    {
        // logic to submit order
        // notify all clients
        OrderHub.Value.Clients.All.orderReceived(order);

    }
}
```

Figure 5.16: Code snippet shows *SignalR* server side implementations.

Both *Ajax* and *SignalR* use the same data structure, JavaScript Object Notation (JSON), to carry the data back and forth between client and server in WCRS. JSON is a format for data-interchange that takes forms in name values pairs. JSON formatted data has a smaller data payload compared with XML, thus making it a good option for data transfer [96]. Particularly in modern web application, JSON is increasingly popular as the communication

medium. Sending data from the client side to the server side involves serializing the JavaScript object into JSON string. At server side, the received string is then de-serialized and parsed to a compatible form that is usable to the system. In ASP.NET, the conversions of JSON string into domain entity object are automatically handled by model binding of the framework. On the other hand, converting the domain entity into JSON string involves serializing the class's public properties. The framework also handles this conversion.

#### 5.3.3.4 Security

The project involves multiple users and their details are stored within the system. The data may be access and tampered with by an external party if security is not ensured . Hence, the implementation of the system needs to take precautions when considering security issues which could be detrimental to the user.

The project mainly controls the user access through *role-based authentication*. The *Controller* is aware of the viable (allowed) roles that have access to the system data. When the user tries to submit a request to the *Controller*, his or her role will be verified. If the user does not incorporate the specified roles, the system will route the user back to the login screen and inform them that the request is not authorized. In addition, the system takes another level of precaution by not presenting the unauthorized menu items in the navigation bar. The user could request the administrator or manager to associate roles if other menu items are needed.

Utilizing ORM for the database access also shield the system from SQL Injection. SQL Injection is essentially passing malicious code in the string of SQL code as parameters, which in turn is executed at the server and yields destructive results. The SQL queries generated by ORM framework is parameterized queries. This helps to escape the malicious code from the request parameter string before it is passed to the server. The system also encrypts sensitive data such as password and credit card details before storing to the database. The encryption solution is provided by ASP.NET hence it is tested and trustable.

## 5.4 Walkthrough

In order to demonstrate the usage of the system, a series of walkthroughs will be presented. This section will cover the important process of restaurant operations, starting from food order, kitchen preparation and payment made. The other features will be presented in the form of screenshots at Appendix O.

### 5.4.1 Submitting Order

The first time the waiter accesses the application, the system will prompt the user to login into the system as shown in Figure 5.17. After the waiter, successfully logs in, the waiter can then expand the navigation menu on the top right to access the order view, as illustrated by Figure 5.18. The order view in Figure 5.19 presents orders submitted on that day and their status. At this view, the waiter initiates order submission by selecting the *Submit New* button. The system will then present the view to construct the order.

As shown in Figure 5.20, the user identity is automatically associated with the order. The waiter will first select a table for the new order. The waiter can then browse the menu recipes by selecting the *Browse Menu Recipes* tab, as shown in Figure 5.21. The tab shows all the menu recipes that are currently available for order. It also allows the waiter to filter the list by using *Browse by Menu* dropdown box. The waiter then can select the *Add* link to order that recipe. The selected recipe will remove from the available menu list and bring the waiter back to order details tab. The waiter can then specify the quantity and comment on each item, which is illustrated in Figure 5.22.

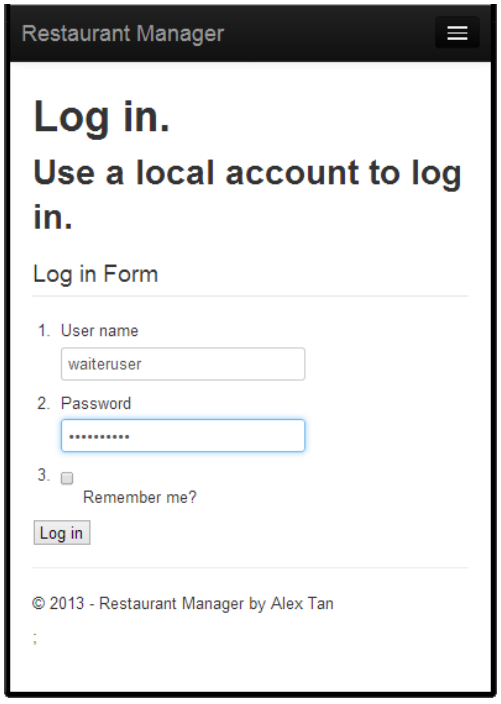



Figure 5.17: Screenshot shows waiter login form.

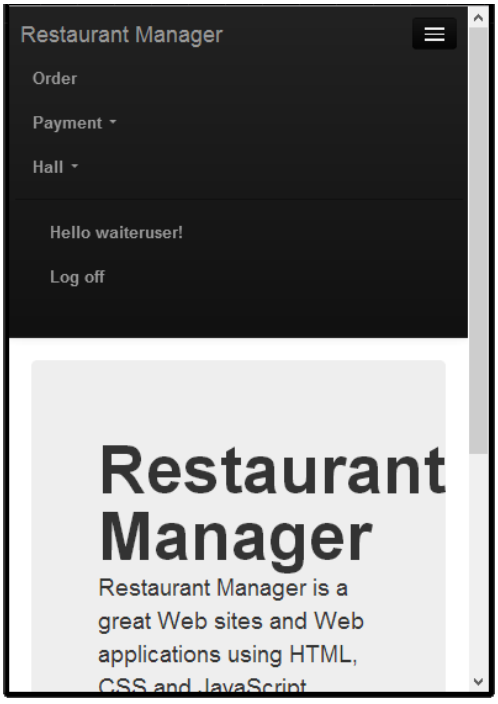



Figure 5.18: Screenshot shows waiter expand navigation menu for order view.

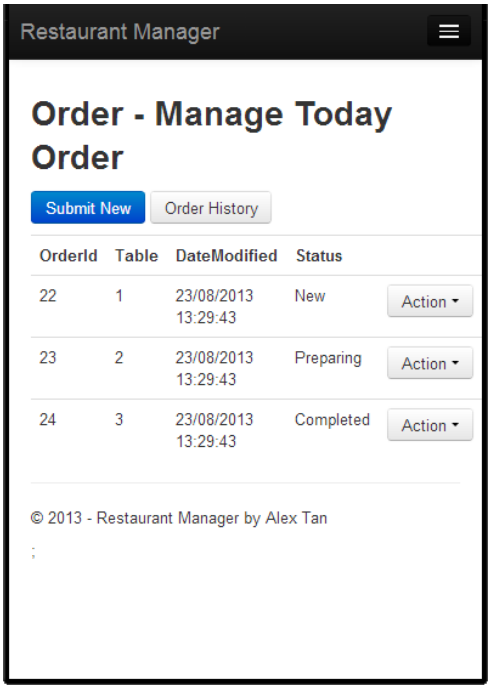



Figure 5.19: Screenshot shows system present today orders.

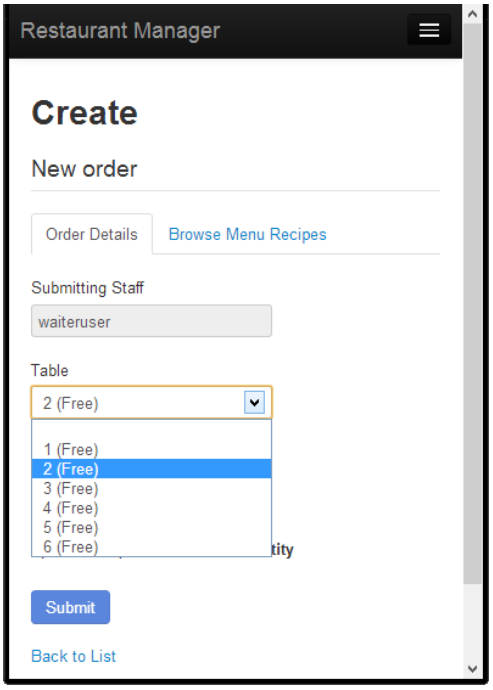



Figure 5.20: Screenshot shows that user is selecting a table in order creation.

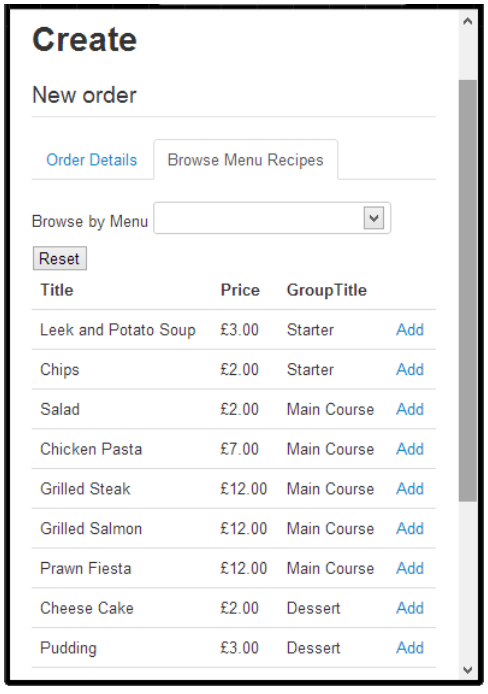



Figure 5.21: Screenshot shows that user is browsing menu recipes.

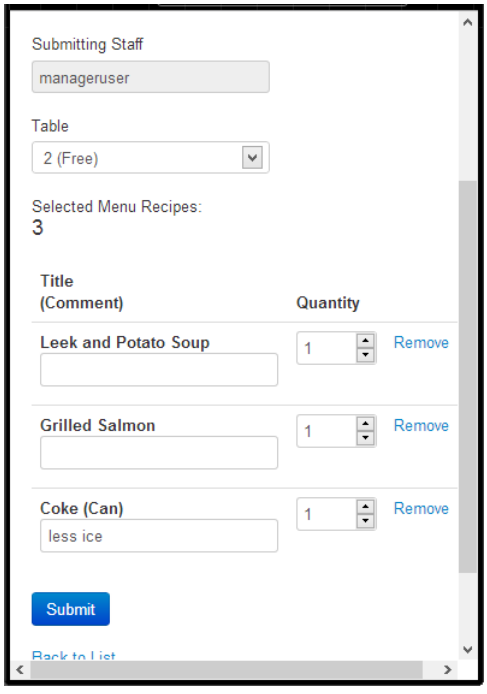



Figure 5.22: Screen shows that menu recipes are added to the order.

The waiter can then click on the submit button after specifying the order items based on customer request. Figure 5.23 shows that when the order is successfully sent, the order will be added to the current order list. However, if the system has not enough materials to proces the order request, it warns the waiter and suggests the available quantity as shown in Figure 5.24. When the order is successfully sent, the system will notify the Kitchen about the new order request in real time. The kitchen view will be immediately updated with the new order details, as illustrated in Figure 5.25.

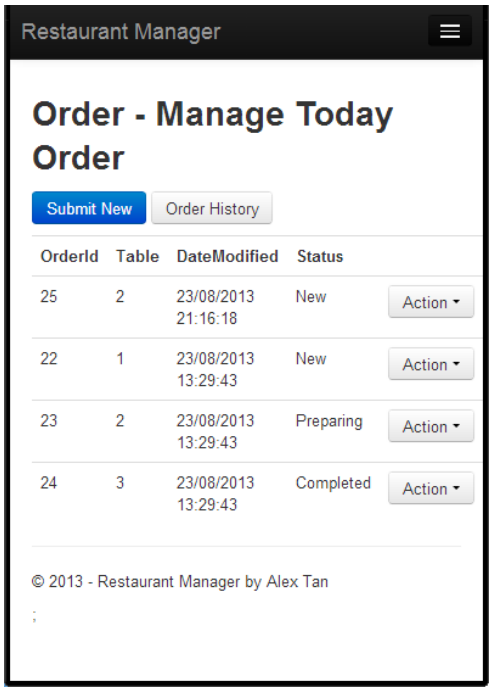



Figure 5.23: Screenshot shows that the user successfully submitted order.

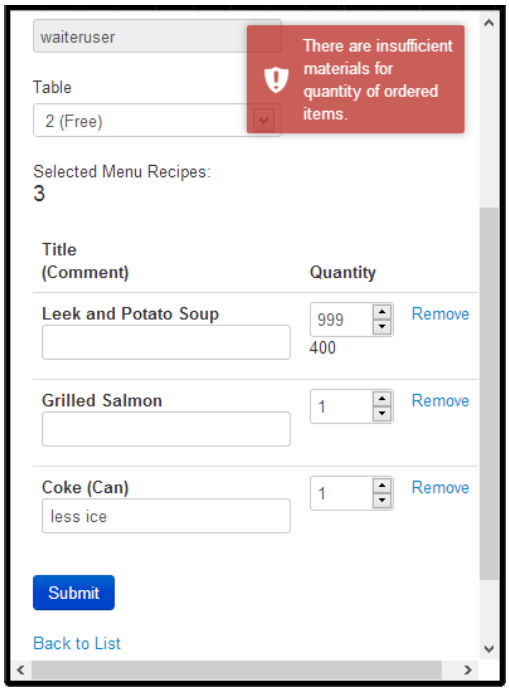



Figure 5.24: Screenshot shows that system warns insufficient materials and presents the current available quantity.

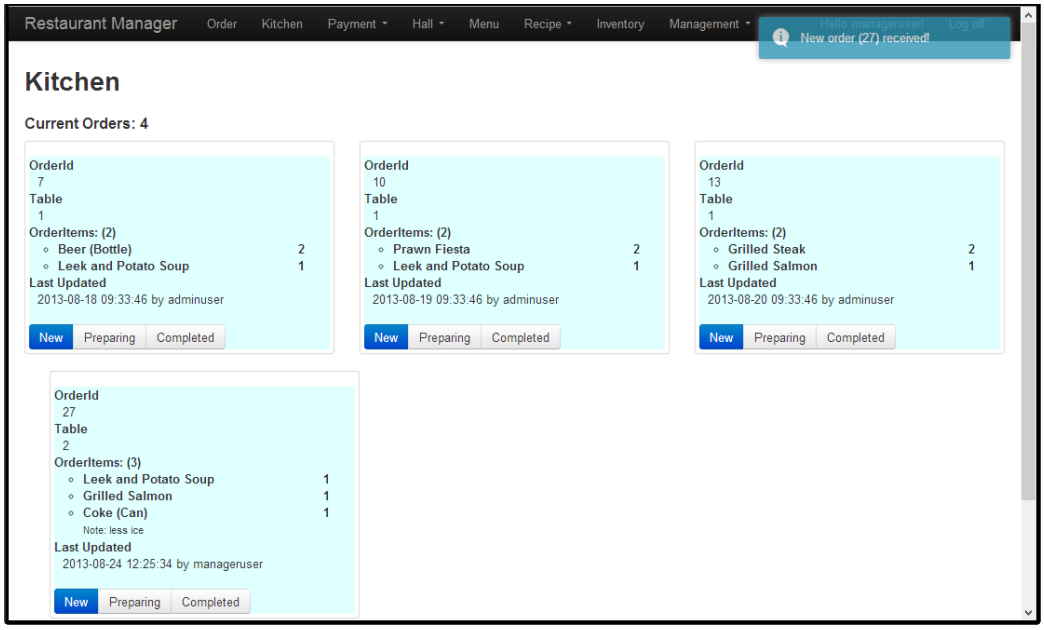



Figure 5.25: Screenshot shows that the kitchen view received new order notification.

### 5.4.2 Updating Order Status

The Kitchen view, as shown in Figure 5.26, presents the order information side by side. When the chef want to prepare the meal for the order, the chef first updates the order status to *Preparing* (see Figure 5.27). This will change the colour of the order across all clients viewing kitchen views in real time. This is to prevent the same order being attended to by multiple chef.

When the order has been prepared, the chef will then select the *Completed* button. The selected order will be removed from the kitchen view, as shown in Figure 5.28. These changes also synchronize across all connected kitchen views. In addition, the system will send a notification to the waiter to inform them that the order is ready to be served.

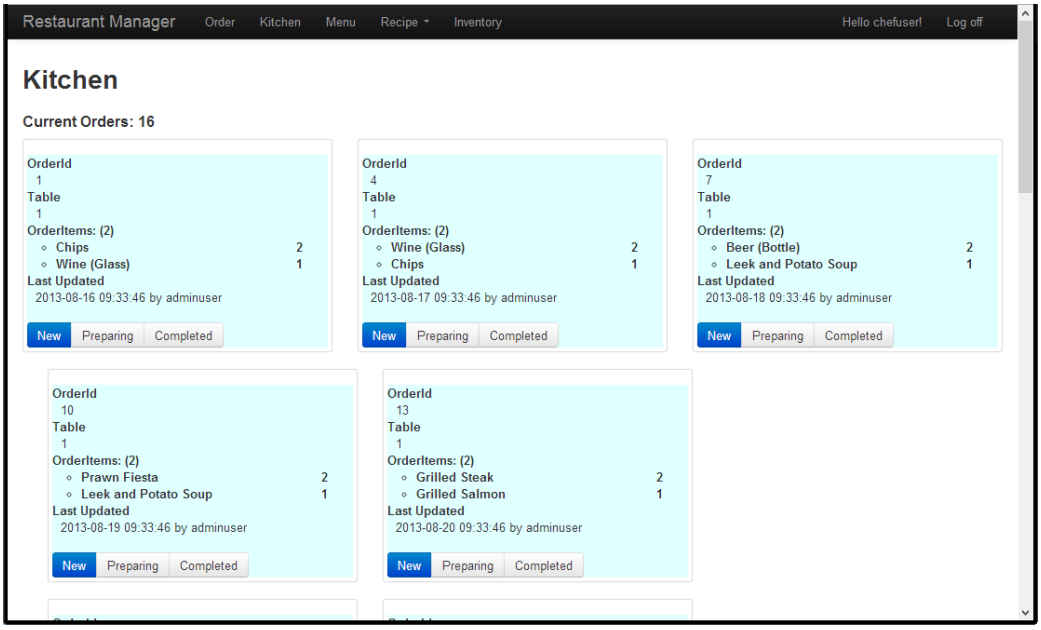



Figure 5.26: Screenshot shows the Kitchen view.

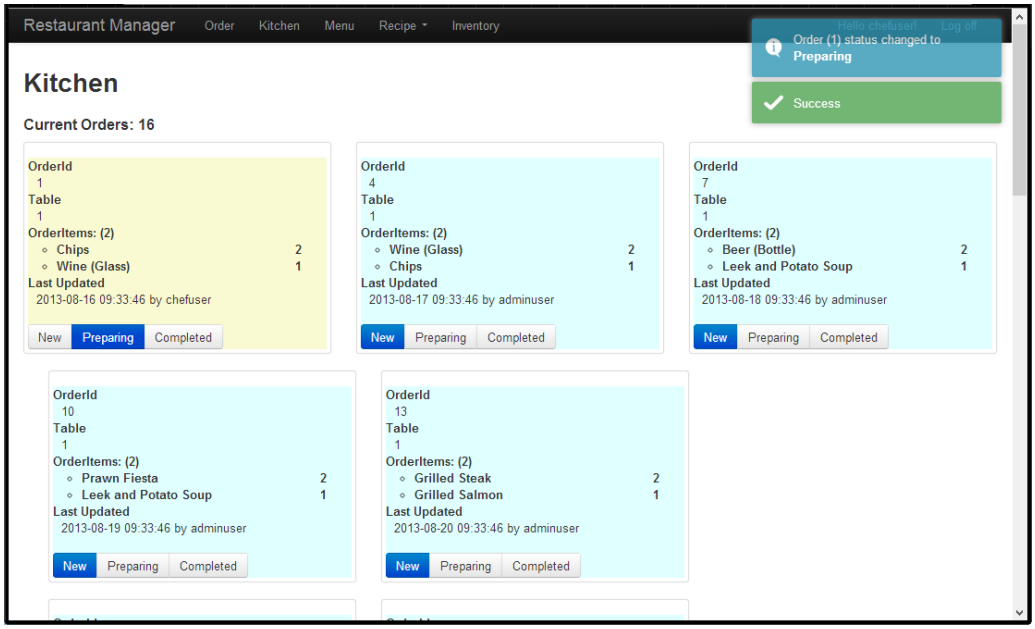



Figure 5.27: Screenshot shows updating order to Preparing status.

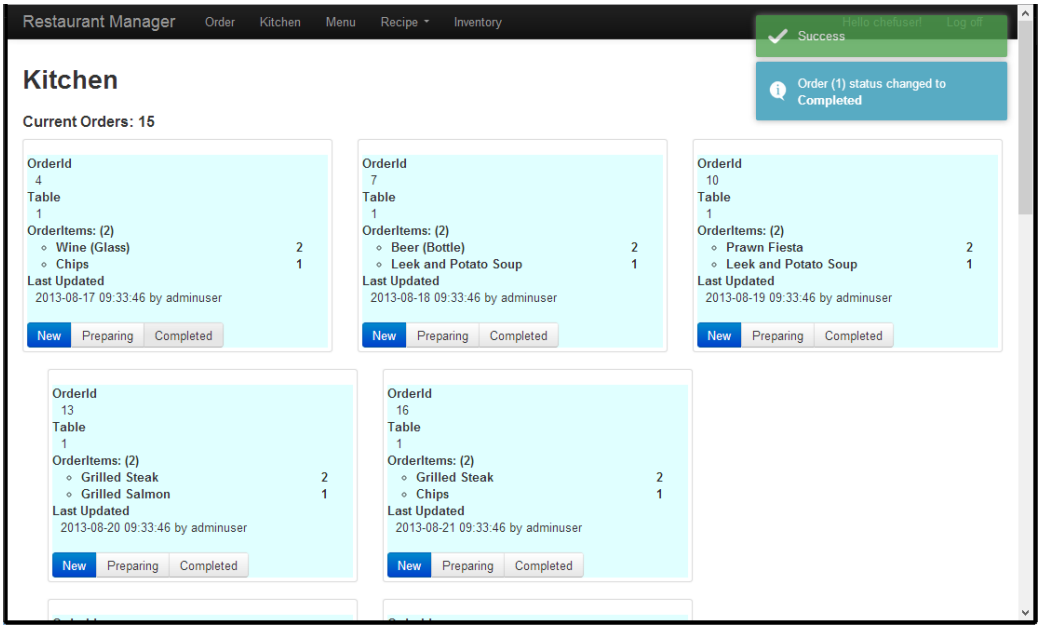



Figure 5.28: Screenshots shows completed order is removed from Kitchen view.

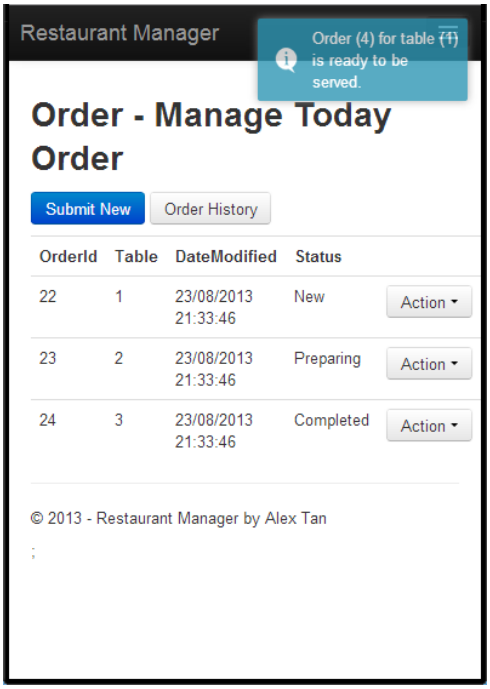



Figure 5.29: Screenshot shows that waiter is notified about completed order

### 5.4.3 Processing Payment

In order to process payment, the cashier will first present the bill to the customer through Payment view. The Payment view will list the order that is currently unpaid, as shown in Figure 5.30. Selecting the *Bill* button will send the order details to the printer, an example of which is shown in Figure 5.33. The cashier can then select the pay button to record the payment details. Record payment view will also present the bill information in case the cashier needs to reference the order details. The payment method could be cash or credit card, as shown in Figure 5.32. Finally, the receipt will be dispensed to the customer

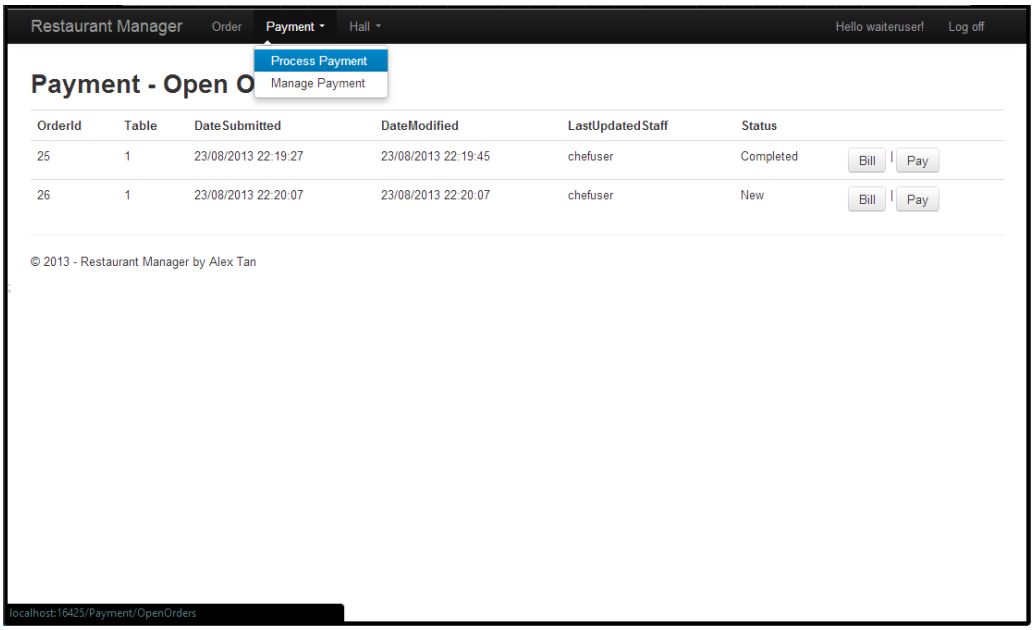



Figure 5.30: Screenshot shows the Payment view.

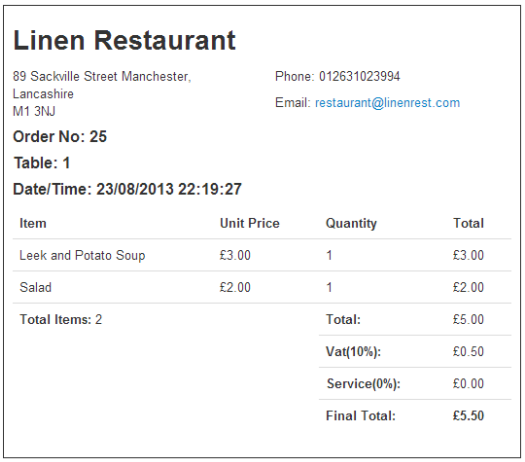



Figure 5.31: Screenshot shows the Bill for the payment.

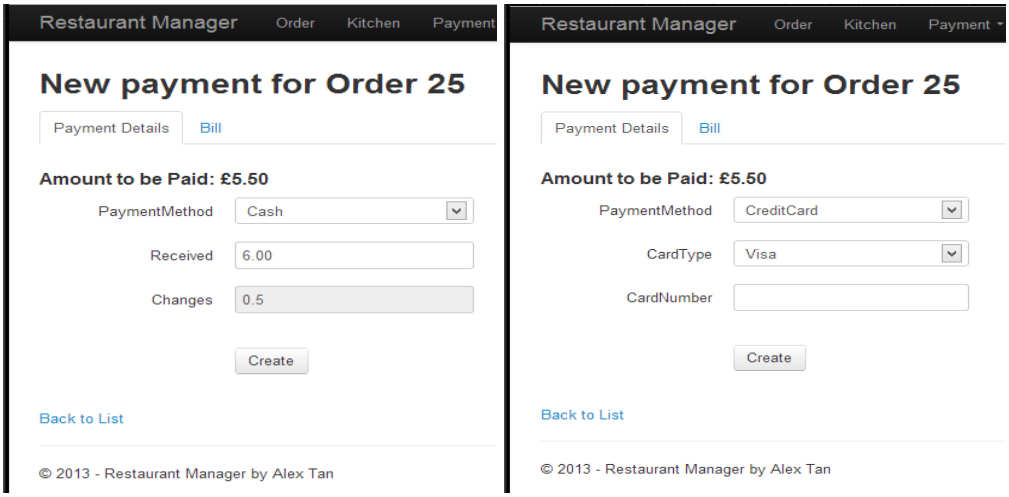



Figure 5.32: Screenshot shows payment methods for the order.

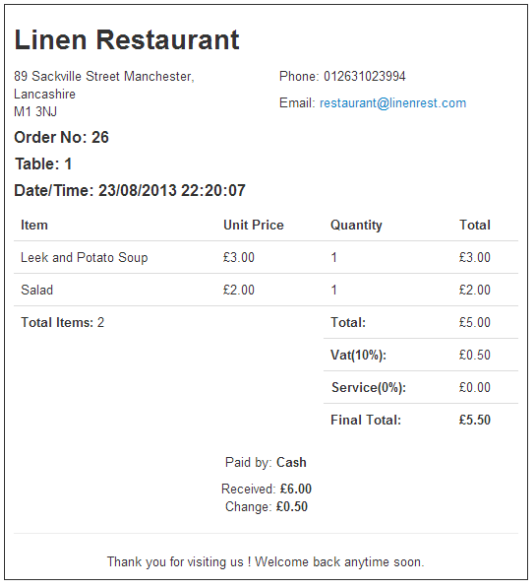



Figure 5.33: Screenshot shows the receipt for payment.

## 5.5 Concluding Remarks

This chapter has covered the underlying concepts and technologies to build the WCRS system. Segregating the implementation into different software layers has proven useful to underpin the principle of separation of concern. It allows the author to focus a smaller set of problems when implementing each layers. This reduces the complexity of the implementation process while enhancing the implementation design.

The next chapter discusses testing process of the system. It describes how the functionalities of the system could be verified and tested.

# Chapter 6. Testing

This chapter describes the software testing concerns of the project. It covers different testing approaches that adopted in the project and some of the techniques applied to realize this process. The chapter first provides an overview of software testing. Then, the processes of each adopted testing approaches are discussed in depth.

## 6.1 Software Testing

Software testing is the process of verifying software implementation work against its requirements. This is meat to inform the developer that possible errors are present before the software artefact is utilised by the actual user – final stakeholder or client. Software testing generally has two main objectives [22]:

- To ensure that the implementation is as intended and the requirements haves been met; and
- To uncover system defects, including those incorrect, undesirable or inconsistent to requirement behaviours.

Software testing could also be viewed as a key contributor to software quality [97]. A high quality software simply means a product that has high user satisfaction while maintaining a low defects rate [98]. Software that has gone through rigorous testing will be likely to yield a better quality product.

Testability of the application was always been the project's concern when designing and implementing the system. The design of the WCRS centred on the *separation of concern* principles. Testing is much easier if only verifying a small set of components. Besides, the uses of software design pattern also promote loose coupling among the software components, which allows these components to be tested in isolation rather than analysing their dependencies.

Testing was often left to the last phase of development in traditional software development methods. If however this is a complex software artefact, testing is required at the start; as it requires significant effort to fix the bugs at the end of the project. The complexity of bugs is greater when they are accumulated across the different parts of the system, resulting in more time being required to analyse and investigate how they occurred – and then find a solution. In WCRS, testing is done before or in parallel with the actual implementation to ensure that the features are worked as intended – this could be viewed as design for testability. This

approach adheres to the test first principle of XP where the sooner the testing concerns are addressed, the better, as the developer will have a clearer understanding of what is to be expected and integrated into the test cases. Hence, this lets the developer think through the solution before writing code. The developer can then effectively find the required implementation and design that solves the problems if the testing fail, and hence has minimal impacts other working part of the artefact.

Testing can be viewed from two broad perspective: *functional testing* and *structural testing. Functional testing*, sometimes referred to black box testing, is testing on the functionality of the system based on the specified requirement. The test itself has little knowledge about the testing target's internal structure. In contrast, *structural testing*, also known as white-box testing, involves examining the internal implementation. It tests the design used by the implementation to verify it correctness. [98] suggested the best approach is to combine both type of testing as shown in Figure 6.1. At the highest level, the testing ensures that specific set user requirements are fulfilled. At lowest level, it ensures that the implemented design is capable of achieving the expected result. These testing techniques will be further discussed in §6.4 to §6.6.

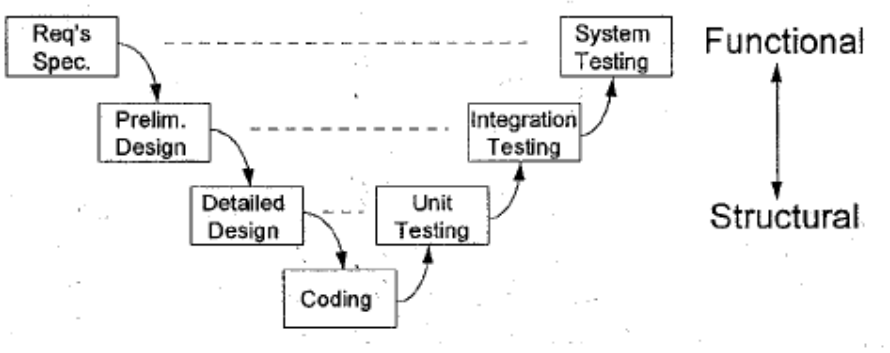



Figure 6.1: Functional Testing versus Structural Testing [98].

## 6.2 Test Automation

Software testing is a repetitive and monotonous task because it involves a repeating cycle of performing action and verifying result. Particularly when the number of items needed to test grows, doing the testing manually could be time consuming and not effective. Hence, most software testing utilizes tools to automate the testing process. A test automation tool could make testing more efficient and quicker by automatically executing the set of defined test cases and verifying their result. This involves writing the scripts to define these steps.

Most IDEs will have integrated test automation tools and let developers manage their test cases easily from the user interface. This is a desirable feature because the developer is constantly refactoring and changing codes during implementation. Integrated testing tools could detect these changes, and in turn provide errors to inform the developer whichever test case is affected. This project utilizes the test automation framework provided VS IDE. Unlike other integrated testing tools, VS provides a full application of lifecycle testing, ranging from unit testing to load testing. In addition, test automation can also be enabled at the TFS after compiling the software at the cloud (see §5.2.5). This ensures every release passed adequate amounts of testing.

While plenty of unit testing frameworks are available for VS, this project uses the default unit testing framework of VS, known as *MS Test* [99]. *MS Test* creates templates for typical unit testing flow. It allows the developer to define the initialization code at class level or method level. When initialization of the code is undertaken at class level, it executes once the class is instantiated. At method level, the code will be executed for each test method within the class. This helps to avoid redundant boilerplate code for initializing the data required for testing.

## 6.3 Regression Testing

Iterative developments of software products often introduce new changes to existing system behaviours and interfaces. However, the degree of their impact on existing systems could be difficult to estimate. Current features may stop working or new bugs may emerge after these changes. This is again aligned to the *continuous integration* principle that each changes should be properly verified before integrating them into the code base.

One of the key enablers for *continuous integration* is *Regression Testing*. *Regression Testing* is defined as:

> *"re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects,"* [23].

The conducted test cases are an excellent medium to document existing system behaviours. They encapsulate previous assumptions and concepts regarding the system. If they fail after new changes, it means that the system behaviour has deviated from previous intention. This is an excellent time to revise the requirements before proceeding further. In addition, the testing may failed because of implementation errors or poor design. This provides additional

chances to improve the system design rather than let it perform poorly when shipped to the user.

Running every test manually could be expensive and impractical [22]. Testers often trade off the cost by running only the most relevant tests, despite this they may overlook the critical tests. Hence, *regression testing* often associated with the test automation tool discussed in §6.2. In this project, the regression test is executed on every successful software build at the TFS. The build is triggered by code check-in, hence every new changes are securely verifying by the server. In addition, the server provides a meaningful description of the failing cases in log files. This easily enables tracing to the source of problem.

*Regression testing* is naturally a part of the testing process in the project. It can be executed at a different level depending on the purpose. Rerunning unit tests and integration tests are the most common scenarios when the developer is implementing a feature. When the development reaches a stable stage and is ready for release, it often involves full scale regression testing. In WCRS, full scale regression testing is especially important at the end of each prototype version.

## 6.4 Unit Testing

*Unit testing* is a testing technique that focuses on smallest or individual unit of the software [22, 23]. This testing technique is a core value of XP because it encourages the developer continuously to learn from writing tests for their own code [68]. It is beneficial to the organisation because the developer's skills are expanding with the system growth.

In an OO development environment, the smallest unit often refers to a class and their methods. When conducting a unit test for the classes, the test cases often derived from testing all methods of the class. The test inspects through the control paths in the method to ensure maximum coverage and error detection [23]. It checks if the methods yield correct output when tested for particular sets of input. In addition, it tests that the method's behaviour and functionality is relevant to its goal. Figure 6.2 shows an example of unit testing done for the project. The syntax used here is much easier to be interpret making the testing output log more self-declarative. This syntax is enabled by *Fluent Assertions* [100] – an open source unit testing library in .NET platform.

```
[TestMethod()]
public void CreateRecipeTest()
{
    using (var context = _factory.Create())
    {
        var target = new RecipeServices(context);
        var recipe = new Recipe
        {
            Title = "Testing title",
            Serving = 1,
            Description = "Testing Description"
        };
        target.CreateRecipe(recipe);
        recipe.RecipeId.Should().NotBe(0, "Because it should have generated id");

        var record = context.Recipes.Find(recipe.RecipeId);
        record.Should().NotBeNull("Because it should find record in database");
    }
}
```

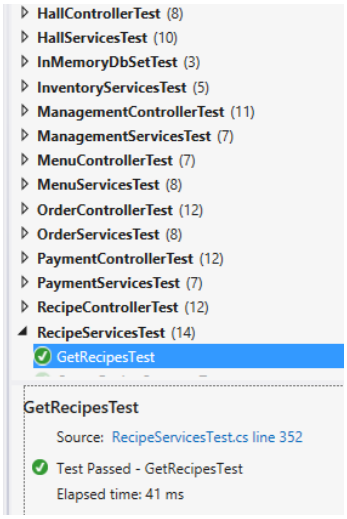



Figure 6.2: A unit test of the *CreateRecipe* method of the *RecipeServices* class.

Testing in isolation is one of the main principles within *unit testing* in the project. This is particularly challenging for layered architecture software because the top layer depends on the bottom layers. The controller will need to invoke domain service operations, while domain services require communication with the database. To address this challenge, this project use mocks and stubs to address the dependencies of the class under test. For instance, to simulate the EF behaviour when testing the domain services, the project designed *InMemoryDbContext* class to mimic the data source with the system memory. There are two primary reasons for this approach:

- Testing domain services should only be concerned with business logic but not data access; and
- Testing with in-memory data source is faster than using the real database.

Figure 6.4 shows that *unit testing* was initially run with an actual database. However, after switching to in-memory data source, unit test execution is almost four times faster as shown in Figure 6.4. As the number of test cases, they would make significant impacts to the testing process. Using an actual database for testing is likely to be undertaken during *Integration Testing*, which is discussed in the next section.

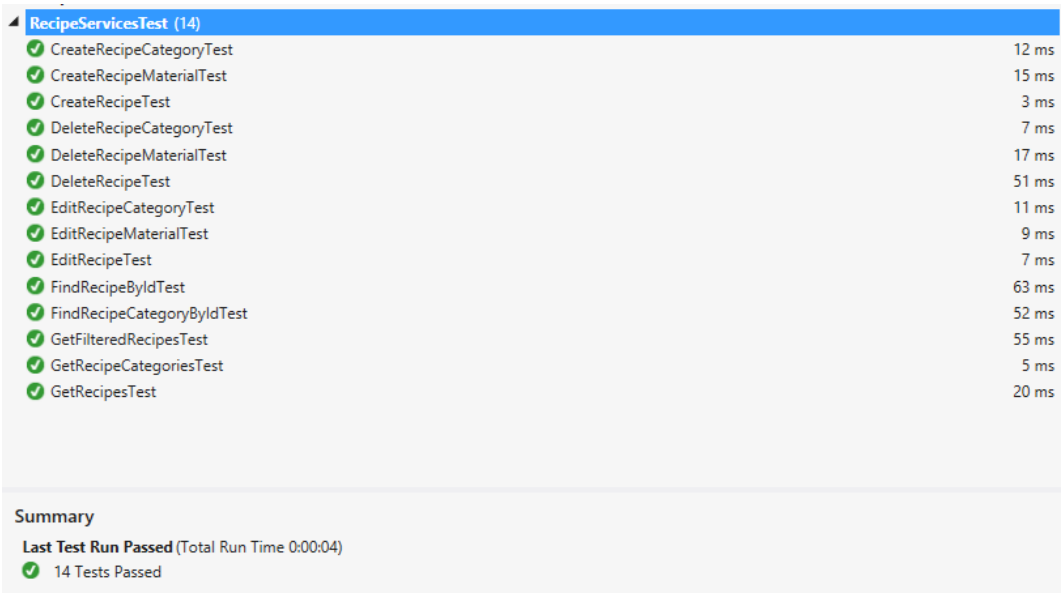



Figure 6.3: Running unit testing with actual database.

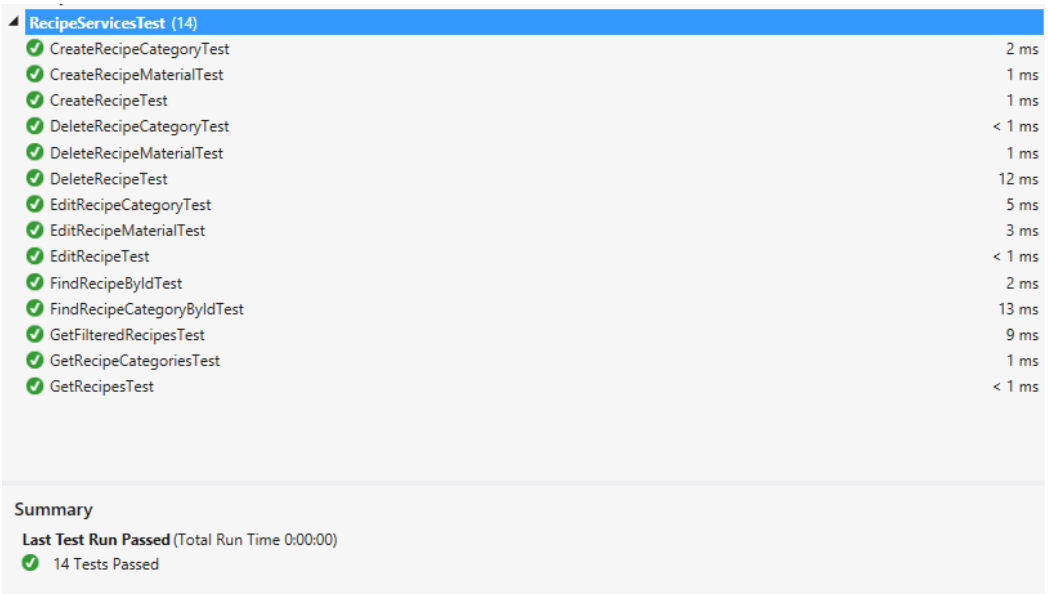



Figure 6.4: Running unit testing with system's memory data source.

## 6.5 Integration Testing

*Integration Testing* is testing multiple components which work together. Often, these components have been tested individually before the integration test. The concerns of integration testing are aligned to testing the interfaces of components and their interaction. This also investigates the techniques used for data exchange among components. It helps to uncover issues such as exposing invalid interface or that the data passed is not in a compatible format. *Integration testing* also verifies the control flow of these components'

interaction and ensures that they are in correct sequence. In the context of WCRS, *integration testing* has two main objectives:

- Testing interaction across multiple software layers; and
- Testing coordination among functional modules.

The system contains software packages that are distributed across software layers due to its three-tiered architecture. Testing interaction across multiple layers is essential to ensure that they expose the correct interface and the received data seamlessly at either end. This project utilizes a bottom-up approach where the components at the lowest level are tested first. In this case, the DAL is tested with an actual database first. Testing with the actual database is particularly important to check if data persistence takes place correctly. Switching to use the actual database is relatively easy, it involves replacing the database context class used in Unit Testing. Next, the BL layer was tested with the PL to ensure that it exposed necessary operations used for client side and view data. This type of testing is done by testing Controller operations with actual BL services and communicating with the real database.

The other goal of *integration testing* is related to testing of functional modules as discussed in §3.1.2.1. The Order module requires the Menu module for order submission, while Menu module depends on the Recipe module to present the meals. Testing at this stage involves all software layers but with different functional concerns aligned to each. This testing covers an end-to-end scenario of the system functionalities. Hence, each module is tested by accessing the user interface and performing the sets of test cases derived from requirements. Then, all the interacting modules must reflect the result of these operations. For instance, the addition of a new recipe to the menu must be visible at the order submission view. As the system is developed in an incremental process, interactions between modules were also tested incrementally. This allows errors to be discovered and fixed at a smaller level.

## 6.6 System Testing

*System testing* is testing the fully integrated system as a whole. It is the testing phase after *integration testing*; but undertaken on all system components. It aims to discover undesirable behaviours when all the components are working together [22]. It also checks if the system conforms to the requirements and expectations. In addition, it also helps to understand the limit of the system and ensures that the system is reliable. At this stage, *unit testing* and *integration testing* have addressed most of the functionality concerns. Hence, *system testing* in WCRS focuses on the non-functional aspects of the system.

### 6.6.1 Security Testing

The open access nature of web application could be a threat to the system if security concerns are involved. It can be accessed easily by devices with a web browser within the connected network. This opens up the possibility of unauthorized access and malicious security threat that would comprise the system. Hence, *Security Testing* is performed on the system to review the degree of reliability and safeness.

The security testing ultimately is concerned with the user authentication and authorization. It first tests the login process of the system and ensures that proper credentials are required to access the system. Both valid and invalid login credentials are tested and their login results are compared. As discussed in §5.3.3.4, the system utilizes *role-based authentication* to restrict the accesses of users. It will present only the navigation link for particular functions if the user has a role associated with their account. This feature is tested by assigning roles to the user account and verifying if the access is granted correctly. Then, the roles are removed from the user account and tested if access is denied.

The credit card number and password when bill payment is made should be encrypted securely before saving to the database. This is to ensure that the data would not be exposed easily even when access to the system is been compromised. The testing is undertaken by repeating inserting a new user account and payment record to check if the sensitive information is encrypted at the database.

### 6.6.2 Performance Testing

Performance testing is another testing focus with the system's reliability and capability to withstand an intense load. It is described as:

> "*running a series of test where you increase the load until the system performance becomes unacceptable,*" [22].

Since the prototype system is targeting restaurant environment, the estimated amount of concurrent user access is lower compared than ordinary Internet applications. However, understanding the limits of the system is essential because it should not fail when it is needed most, especially during peak time. Hence, this project utilized the *Web Performance Testing* and *Load Testing* tools provided by VS to execute performance testing.

The testing environment is one of the crucial factors in performance testing because a high performance hardware will definitely give a better result. In this project, the testing environment is same as the development environment as shown in Table 6.1. The test machine is considered to be excellent for data processing due it's quad-core processor. Having 8GB of memory is sufficient in handling load for small to medium restaurant, though, the system will prefer a higher memory capacity in a large restaurant.

Table 6.1: Test Machine Specification for Performance Testing.

| **Hardware** | **Hardware Information** |
|---|---|
| **Processor** | Intel® Core™ i7-3610QM CPU @ 2.30 GHz |
| **Installed Memory (RAM)** | 8.00 GB |
| **Hard Disks** | Samsung SSD 830 SATAIII 6GBps 256GB |
| **Operating System** | Window 8 Pro 64-bit |

After the test environment is defined, the next step is to figure which functions that the users will be frequently accessing. One of the most frequently accessed functions in WCRS is the order submission. It is also one of the busiest parts (phases) in the order processing cycle. The project first sets up a recording of order submission scenario with the VS as shown in Figure 6.5. It captures every request and response when the scenario that is then stimulated at the web browser. When the recording is played back, it ensures that every steps is executed and provides analysis of the response time.

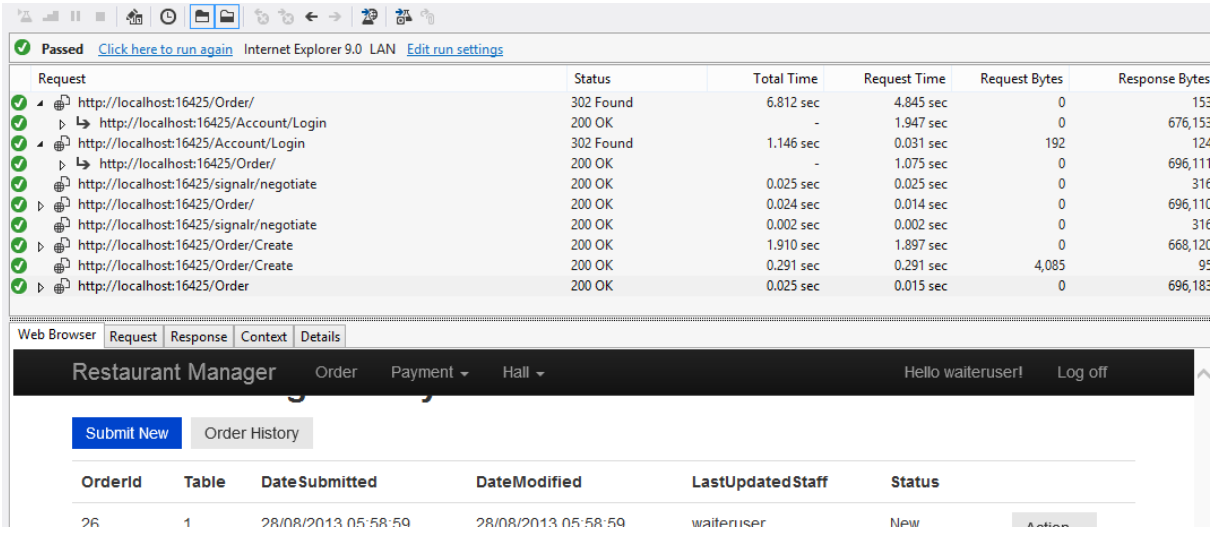



Figure 6.5: Recording of order submission scenario.

Next, the project will be configured for Load testing by launching the Load Test wizard shown in Figure 6.6. This allows tester to define the testing scenario and the test execution pattern. In this case, it selects the step load pattern where the number of users will be 5 initially, and gradually increased until 40.

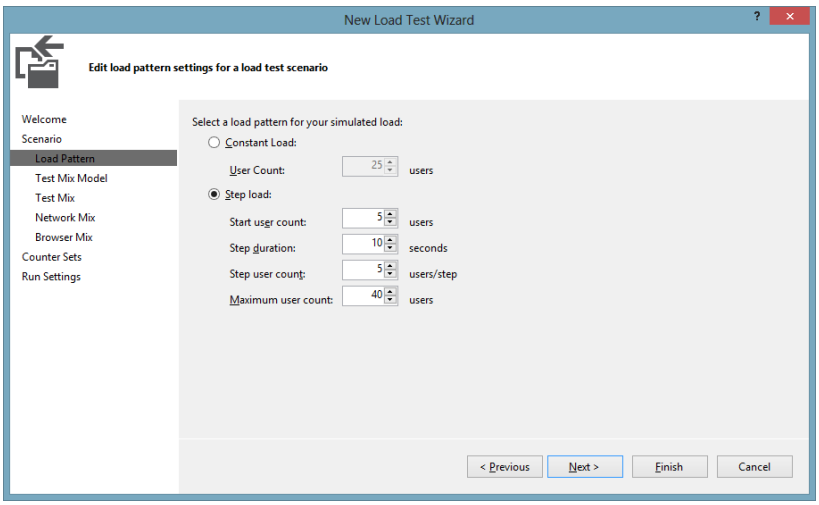



Figure 6.6: Configuration of Load Test scenario at wizard.

After the configuration is completed, the Load testing is ready to be executed. Figure 6.7 shows the load testing is executing on the test machine. The load testing tools provide informative graphs and statistics regarding the current system performance. As for web application, the page load time is the most relevant statistic because it tells the developer which page need to be optimized for performance. When the execution completed, a report of the testing result will be presented to the tester, as shown in Figure 6.8. The report's *Test Results* section indicates that the system survived the load testing, as it shows none of the order submission flows have failed. It also shows the slowest pages that need developer attentions.


Figure 6.7: Chart and statistic when executing Load Testing.

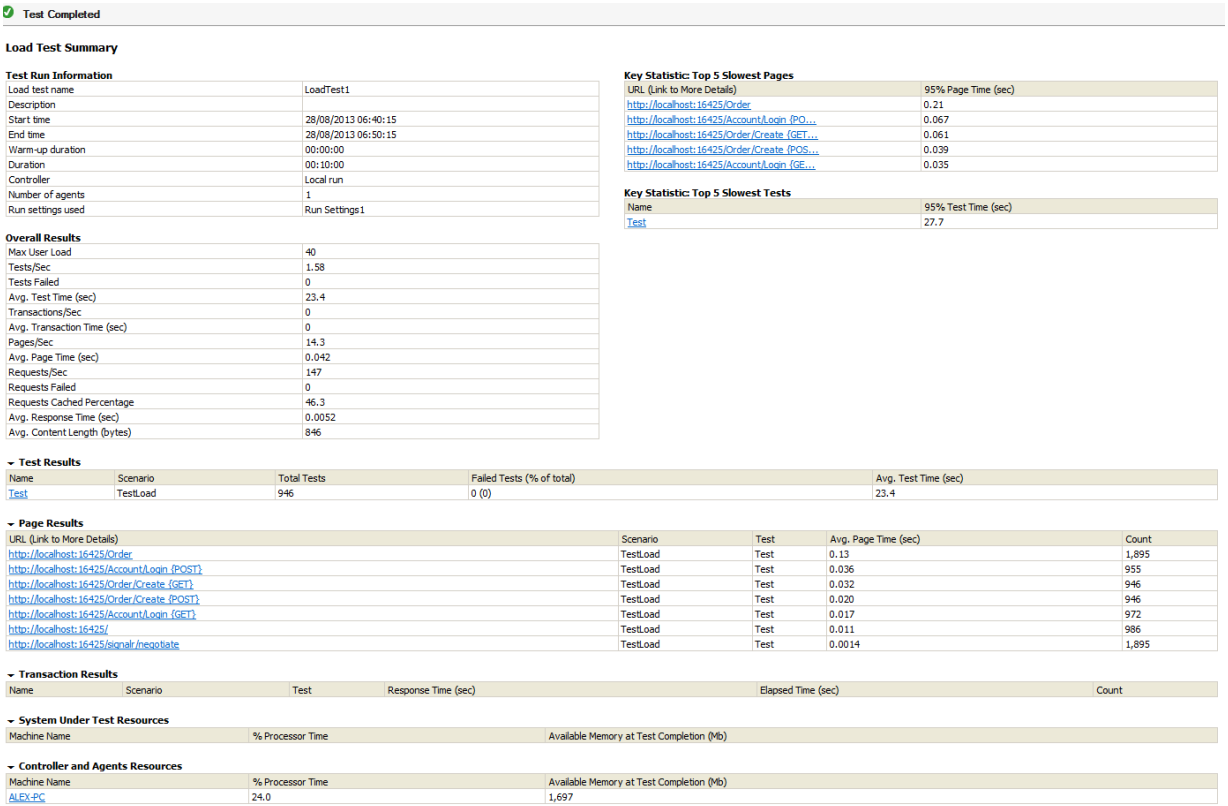



Figure 6.8: Report of Load Testing result.

## 6.7 Concluding Remarks

This chapter has documented the extensive testing approaches that were utilised in this project and the process of how they were carried out. Different and all-encompassing testing approaches have been conducted to ensure the system is as robust as possible. The system has proved to be functional and usable after all the testing was done. The chapter has marked the end of the software development process of the project.

While rigorous testing processes have covered (tested) many aspects of the system, the testing activities are likely to carry on throughout the system life cycle. This is aligned to the popular software engineering mantra by Dijkstra [101],

"*Testing shows the presence, not the absence of bugs.*"

There will be errors that not have yet been discovered by testing and will only emerge during user usage. Nevertheless, the project is structured in a way that will enable changes and fixes to be easily undertaken, hence their risks have been mitigated.

The next chapter is the final chapter of the report. It concludes the report of the project and evaluates approaches taken to complete the project.

# Chapter 7. Conclusion

This chapter concludes the overall process of the project. It looks back at the work completed and evaluates the research and approaches taken. Finally, it marked the end of the report by a summary.

## 7.1 Project Achievement

The project has gone through a series of activities to develop a complex solution for the computerized restaurant system. After analysis of the project's goal and research direction, a set of objectives were established, as specified in §1.3. All the activities done during the project were attempts to realize these objectives. At the end of the project, the developed prototype software has fulfilled these objectives by the following means:

- Objective #1 was satisfied by implementing the prototype system with three-tier architecture and presentation separation pattern.
- Objective #2 was addressed by utilizing Hierarchy Task Analysis (HTA) to model user interface's presentation and behaviour.
- Objective #3 was satisfied by integrating Responsive Web Design (RWD) to allow mobile friendly access to the system.
- Objective #4 was satisfied by adopting Remote Procedure Call and Server Push technology for real-time communication between client and server.
- Objective #5 was addressed with various testing approaches to ensure the prototype system is as robust as possible.

Employing the agile development method also proved useful in managing the software development process. The software prototype was constantly evolving thanks to the incremental and iterative development cycle. It was tested at every iteration, hence most defects were addressed early on in the project. Besides, it reduces the scope of implementation and allows the author to manage the development activities effectively. The project management techniques are discussed in §3.3 which also demonstrates their values for time and workload management. The project plan was constantly revised to reflect the progress and capability of the author based on these techniques.

The project was ambitious and time-consuming, implementing as many features as possible within the very limited timeframe. It has successfully satisfied the Functional Requirements (FR) from FR1 to FR14 and all Non-functional Requirements (NFR) of the system. These requirements have top priority and reflect the most needed features required by stakeholders. FR 15 through 19 are not implemented due to time constraints. However, they are the lower priority features that are pleasant – but not paramount. Their absence would not result in major operational issues in restaurant order processing. As the system was designed to be easily extendable, these features could be implemented in the future. At the bottom line, the system is useable in term of the stakeholder's need and operational concerns. It satisfied the basic goal of replacing the paper-based system in the restaurant operations.

## 7.2 Research Evaluation

Throughout this project, extensive researches have been carried out to address the unknown areas that the author needed to investigate. The researches within this project have covered the entire software development life cycle. The project has analysed different software process models (§2.3) and performed comparisons (§2.3.4) before adopting the most suitable model for the project.

The project has also looked at the process of requirement gathering with respect to the appropriate approaches required by the project and developed the requirements based on the most appropriate techniques evaluated (§3.1.1). The research into software design techniques has established a high-level vision of software architecture (§4.2.1) and system modelling (§4.3) that answered the stakeholder requirements.

Research into the user interface design (§2.4) has proven useful in order to construct the complex and numerous GUIs that were designed so they were usable by the users. This is further supported by research into the presentation separation pattern (§5.3.3.1 and §5.3.3.2) and introduced effective methods to build application that could be easily managed and tested.

The research into Remote Procedure Call and Server Push technology (§5.3.3.3 demonstrated real-time communication is extremely useful for restaurant operations. They answer the possibility of real time collaborations of restaurant user with web application.

Finally, the testing techniques research (§6.1) established carefully planned process to ensure the final product is robust and useable. All these researches have contributed to the success of the prototype implementation.

## 7.3 Approach Evaluation

The design approach taken in the project is aligned to the theme of separation of concerns. The project had separated the design problems into several distinct concerns. Research is undertaken for each design concerns, as discussed in §7.2. Modelling and diagraming techniques are then used to produce artefacts that describe design decisions and verify if requirements of stakeholders could be fulfilled with the design approach. The produced design artefacts in the project have covered design concerns including architecture, application behaviour, data structure, and user interface. This demonstrated the problem solving and creativity processes used are structured in a logical manner that could be understood by other software practitioners.

The project's designs evolved into actual implementation by adopting suitable technologies and SE (and HCI) techniques. The utilized technologies and techniques include various: SE methodologies, HCI principles, programming tools, frameworks and libraries during the construction of the prototype software. Besides making the development process more effectively, they have also improved the quality of system since it is a tested solution. However, the implementation approaches centred on the web development framework technologies. This required basic understanding of the mechanisms behind these technologies if the project is to be extend by other peoples. However, the development process will be accelerated once they understanding the basic concepts of these technologies.

The project had adopted different testing approaches to test the prototype software and discovered bugs during these testing were corrected. However, the performance testing done in the project is concerned with general server performance. The testing could also be extended to understand the performance of the JavaScript implementation at the client side. Nevertheless, this largely depends external factors such as devices capabilities and type of web browsers used. Hence, the testing may help in optimizing code efficiency but not aiming to support every possible devices with a web browser.

## 7.4 Reflection

The author has attained various skills throughout this project. The most relevant skill was the software project management. The project covered end-to-end scenarios of building real life software applications and was directly aligned to the lifecycle of the software product. Hence, the author has learned to apply lifecycle management and utilise software process models on a software project.

In terms of technical skill, the project also significantly contributed to the author's knowledge regarding web technology. Throughout the development of the web application, the author has gained a deeper understanding of the underlying communication protocol and client-server architecture. This project has prepared the author to handle more distributed system development in the future.

As rigorous research into software design and design pattern was conducted, this author learnt different approaches of abstracting software development problems by utilising modelling techniques and diagrammatic abstraction methods. These helped the author evolved through the implementation of small details (via the first prototypes) to focusing on high-level design challenges. Hence, this has improved the authors' analytical and problem solving skills.

The author also gained exposure to the latest technology trends in software development and hands on experience with development tools. The skills attained while working with these tools include:

i. Utilizing a web development framework to build a highly reusable web application;
ii. Communicating with database with object relational mapping;
iii. Implementing mobile friendly web user interface;
iv. Advanced testing techniques with test automation tools; and
v. Managing software development with continuous integration server.

The author has become more confident to handle future software development projects with these skills. In addition, the author is much more prepared to face new challenges in software engineering.

## 7.5 Future Improvement

In addition to the unfinished requirements, there are other possibilities of further improving the project. The improvements may include:

1. Presenting graphical floor plan for table management and reservation;
2. Support food order delivery and driver tracking;
3. Extension of pricing methods for individual or multiple recipes;
4. Advanced inventory control with material storage and expiry information; and
5. Managing customer loyalty membership and discount voucher.

Another interesting possibility is to host the entire system on Cloud-based services. If the restaurant business model expanded to multiple outlets, the restaurant manager could access the data of different restaurants to view their performance reports or order materials from suppliers.

## 7.6 Concluding Remarks

This chapter has concluded the report of this project. The project successfully implemented a working complex prototype of a web-based computerized restaurant system. The implemented prototype software has been fully tested throughout the project phases and it demonstrated acceptable performance. Overall, the project enabled the author to completed most of the high priority requirements. This report also documented all the relevant research details and decision-makings processes. If future extensions of the system are undertaken, the report will be helpful in assisting the completion of the remaining requirements and future improvements that might be involved. In summary, the project has satisfied its objectives and fulfilled its purpose to assist restaurant operations.

# References


[1] POSitive Technologies INC. (2013). *The Past, Present and Future of POS* [Online]. Available: http://www.positivetech.com/2013/02/27/the-past-present-and-future-of-pos/ (Last Accessed: 20/4/2013).

[2] touchPOS.net. *History of POS Systems* [Online]. Available: http://www.touchpos.net/page.html?chapter=10&id=9 (Last Accessed: 20/4/2013).

[3] G. Bisson. (1998) Getting Down To Business : Using The ST In An IBM World. *STart*.

[4] T. Shimmura, T. Takenaka, and M. Akamatsu, "Real-Time Process Management System in a Restaurant by Sharing Food Order Information," in *Soft Computing and Pattern Recognition, 2009. SOCPAR '09. International Conference of*, 2009, pp. 703-706.

[5] K. J. Patel, U. Patel, and A. Obersnel, "PDA-based wireless food ordering system for hospitality industry — A case atudy of Box Hill Institute," in *Wireless Telecommunications Symposium, 2007. WTS 2007*, 2007, pp. 1-8.

[6] Y. H. Huo, "Information technology and the performance of the restaurant firms," *Journal of Hospitality & Tourism Research*, vol. 22, pp. 239-251, 1998.

[7] R. Leung and R. Law, "Evaluation of Hotel Information Technologies and EDI Adoption: The Perspective of Hotel IT Managers in Hong Kong," *Cornell Hospitality Quarterly*, vol. 54, pp. 25-37, February 1, 2013 2013.

[8] T. Tan-Hsu, C. Ching-Su, and C. Yung-Fu, "Developing an Intelligent e-Restaurant With a Menu Recommender for Customer-Centric Service," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 42, pp. 775-787, 2012.

[9] C. Soon Nyean, C. Wei Wing, and Y. Wen Jiun, "Design and development of Multi-touchable E-restaurant Management System," in *Science and Social Research (CSSR), 2010 International Conference on*, 2010, pp. 680-685.

[10] E. W. T. Ngai, F. F. C. Suk, and S. Y. Y. Lo, "Development of an RFID-based sushi management system: The case of a conveyor-belt sushi restaurant," *International Journal of Production Economics*, vol. 112, pp. 630-645, 4// 2008.

[11] T.-H. Chen, H.-H. Lin, and Y.-D. Yen, "Mojo iCuisine: The Design and Implementation of an Interactive Restaurant Tabletop Menu," in *Human-Computer Interaction. Towards Mobile and Intelligent Interaction Environments*. vol. 6763, J. Jacko, Ed., ed: Springer Berlin Heidelberg, 2011, pp. 185-194.

[12] C. Rich, "Building Task-Based User Interfaces with ANSI/CEA-2018," *Computer*, vol. 42, pp. 20-27, 2009.

[13] D. Ansel and C. Dyer, "A Framework for Restaurant Information Technology," *Cornell Hotel and Restaurant Administration Quarterly*, vol. 40, pp. 74-84, June 1, 1999 1999.

[14] C. R. Oronsky and P. K. Chathoth, "An exploratory study examining information technology adoption and implementation in full-service restaurant firms," *International Journal of Hospitality Management*, vol. 26, pp. 941-956, 2007.

[15] M. D. Olsen and D. J. Connolly, "Experience-based Travel How Technology Is Changing the Hospitality Industry," *Cornell Hotel and Restaurant Administration Quarterly*, vol. 41, pp. 30-40, 2000.

[16] J. Lukkari, J. Korhonen, and T. Ojala, "SmartRestaurant: mobile payments in context-aware environment," in *Proceedings of the 6th international conference on Electronic commerce*, 2004, pp. 575-582.

[17] H. Xu, B. Tang, and W. Song, "Wireless Food Ordering System Based on Web Services," in *Intelligent Computation Technology and Automation, 2009. ICICTA '09. Second International Conference on*, 2009, pp. 475-478.

[18] H. W. Gellersen and M. Gaedke, "Object-oriented Web application development," *Internet Computing, IEEE*, vol. 3, pp. 60-68, 1999.

[19] S. Murugesan, Y. Deshpande, S. Hansen, and A. Ginige, "Web Engineering: a New Discipline for Development of Web-Based Systems," in *Web Engineering*. vol. 2016, S. Murugesan and Y. Deshpande, Eds., ed: Springer Berlin Heidelberg, 2001, pp. 3-13.
[20] M. Jazayeri, "Some Trends in Web Application Development," presented at the 2007 Future of Software Engineering, 2007.
[21] S. R. Schach, *Object-oriented and classical software engineering*, 6th ed ed. New York, London: McGraw-Hill, 2005.
[22] I. Sommerville, *Software engineering*, 9th ed. Boston, MA: Pearson, 2011.
[23] R. S. Pressman, *Software engineering : a practitioner's approach*, 7th international, alternate ed. New York: McGraw-Hill Higher Education, 2010.
[24] W. W. Royce, "Managing the development of large software systems," in *proceedings of IEEE WESCON*, 1970.
[25] A. Dennis, B. H. Wixom, and D. Tegarden, *Systems analysis and design with UML : an object-oriented approach*, 3rd ed. Hoboken: Wiley, 2010.
[26] SourceMaking. *Analysis Paralysis* [Online]. Available: http://sourcemaking.com/antipatterns/analysis-paralysis (Last Accessed: 1/5/2013).
[27] F. Radeke and P. Forbrig, "Patterns in task-based modeling of user interfaces," presented at the Proceedings of the 6th international conference on Task models and diagrams for user interface design, Toulouse, France, 2007.
[28] Microsoft. (2001). *Microsoft Inductive User Interface Guidelines* [Online]. Available: http://msdn.microsoft.com/en-us/library/ms997506.aspx (Last Accessed: 30/4/2013).
[29] M. Sasajima, Y. Kitamura, and R. Mizoguchi, "Task-Oriented User Modeling Method and Its Application to Service Navigation on the Web," in *Database Systems for Advanced Applications*. vol. 6193, M. Yoshikawa, X. Meng, T. Yumoto, Q. Ma, L. Sun, and C. Watanabe, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 240-251.
[30] L. Laporte, P. Eyckerman, and B. Zaman, "Designing a mobile task based UI for tourists," presented at the Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services, Bonn, Germany, 2009.
[31] G. Young. (2010, CQRS Documents by Greg Young. [Document]. Available: http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
[32] A. Janß, W. Lauer, and K. Radermacher, "Using cognitive task analysis for UI design in surgical work systems," in *Proceedings of the 15th European conference on Cognitive ergonomics: the ergonomics of cool interaction*, 2008, p. 34.
[33] D. Diaper and N. Stanton, *The Handbook of Task Analysis for Human-Computer Interaction*: Lawrence Erlbaum Associates, Incorporated, 2004.
[34] P. Hornsby. (2010). *Hierarchical Task Analysis* [Online]. Available: http://www.uxmatters.com/mt/archives/2010/02/hierarchical-task-analysis.php (Last Accessed: 30/4/2013).
[35] A. J. Dix, J. E. Finlay, G. D. Abowd, and R. Beale, *Human Computer Interaction*: Pearson Education Canada, 2006.
[36] D. Bosomworth. (2013). *Mobile Marketing Statistics 2013* [Online]. Available: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/ (Last Accessed: 26/4/2013).
[37] Google. (2012, 30/4/2013). Our Mobile Planet: UK. [Research Study]. Available: http://ssl.gstatic.com/think/docs/our-mobile-planet-united-kingdom_research-studies.pdf
[38] L. Luppes. (2012). *ASP.NET MVC 4 Mobile Web Sites Succinctly*.
[39] R. Hof. (2012). *Google Research: No Mobile Site = Lost Customers* [Online]. Available: http://www.forbes.com/sites/roberthof/2012/09/25/google-research-no-mobile-site-lost-customers/ (Last Accessed: 1/5/2013).
[40] S. Souders, *High Performance Web Sites: Essential Knowledge for Front-End Engineers*: O'Reilly Media, 2008.
[41] Quinta Group. (2012). *Responsive web design - adapt, respond and overcome* [Online]. Available: http://quintagroup.com/services/web-design/responsive-web-design (Last Accessed: 1/5/2013).

[42] Twitter. *Bootstrap* [Online]. Available: http://twitter.github.io/bootstrap/ (Last Accessed: 1/5/2013).
[43] The jQuery Foundation. (2013). *JQuery UI* [Online]. Available: http://jqueryui.com/ (Last Accessed: 1/5/2013).
[44] J. Korpi. *Less Framework 4* [Online]. Available: http://lessframework.com/ (Last Accessed: 1/5/2013).
[45] I. Alexander, "Stakeholders: who is your system for?," *Computing & Control Engineering Journal*, vol. 14, pp. 22-26, 2003.
[46] S. W. Ambler, *Agile modeling : effective practices for eXtreme programming and the unified process*. New York: J. Wiley, 2002.
[47] R. Barker, *Embedded Systems Development Guide: PHASE 1*. Manchester: UMIST, 2002.
[48] T. Mitra. (2008). *Documenting software architecture, Part 2: Develop the system context* [Online]. Available: http://www.ibm.com/developerworks/library/ar-archdoc2/ (Last Accessed: 11 April 2013).
[49] L. Ha Minh, T. Wettengel, and T. Huynh Trung, "Empowering UML application design with task models," in *Information Technology and Multimedia (ICIM), 2011 International Conference on*, 2011, pp. 1-6.
[50] Chambers & Associates Pty Ltd. (2013). *Milestone* [Online]. Available: http://www.chambers.com.au/glossary/milestone.php (Last Accessed: 1/5/2013).
[51] J. Benson. *Personal Kanban 101* [Online]. Available: http://www.personalkanban.com/pk/personal-kanban-101/ (Last Accessed: 1/5/2013).
[52] C. Hu, "The nature of software design and its teaching: an exposition," *ACM Inroads*, vol. 4, pp. 62-72, 2013.
[53] R. N. Taylor and A. v. d. Hoek, "Software Design and Architecture The once and future focus of software engineering," presented at the 2007 Future of Software Engineering, 2007.
[54] M. Fowler, "Is design dead?," in *Extreme programming examined*, ed: Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 3-17.
[55] V. Haren, *Togaf Version 9.1*: Van Haren Publishing, 2011.
[56] M. P. P. Team, *Microsoft® Application Architecture Guide*: Microsoft Press, 2009.
[57] M. Fowler, *Patterns of Enterprise Application Architecture*: Pearson Education, 2012.
[58] L. H. J. N. S. S. P. D. F. A. J. L. M. D. S. Maring. (2000). *Oracle9i Application Server Overview Guide*. Available: http://docs.oracle.com/cd/A97336_01/ias.102/a87353/title.htm (Last Accessed: 8).
[59] S. W. Ambler, *The object primer : Agile Modeling-driven development with UML 2.0*, 3rd ed. Cambridge: Cambridge University Press, 2004.
[60] uml-diagrams.org. (2013). *UML 2.5 Diagrams Overview*. Available: http://www.uml-diagrams.org/uml-25-diagrams.html (Last Accessed: 2/8/2013).
[61] T. M. Connolly and C. E. Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management*: Addison-Wesley, 2005.
[62] Microsoft. (2013). *SQL Server* [Online]. Available: http://www.microsoft.com/en-us/sqlserver/default.aspx (Last Accessed: 1/5/2013).
[63] Oracle Corporation. *Oracle Database*. Available: http://www.oracle.com/uk/products/database/overview/index.html (Last Accessed: 7/8/2013).
[64] Oracle Corporation. (2013). *MySQL The world's most popular open source database*. Available: http://www.mysql.com/ (Last Accessed: 7/8/2013).
[65] W3Schools. (2013). *Introduction to XML*. Available: http://www.w3schools.com/xml/xml_whatis.asp (Last Accessed: 7/8/2013).
[66] Hibernate.org. *What is Object/Relational Mapping?* Available: http://www.hibernate.org/about/orm (Last Accessed: 7/8/2013).
[67] K. Letchford. (2013). *The Ultimate Responsive Web Design Beginners Resource List*. Available: http://www.targetlocal.co.uk/responsive-web-design-resources/ (Last Accessed: 7/8/2013).

[68] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, pp. 70-77, 1999.
[69] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 2006.
[70] Microsoft. (2013). *ASP.NET MVC 4* [Online]. Available: http://www.asp.net/mvc/mvc4 (Last Accessed: 1/5/2013).
[71] M. Tatsubori and T. Suzumura, "HTML templates that fly: a template engine approach to automated offloading from server to client," presented at the Proceedings of the 18th international conference on World wide web, Madrid, Spain, 2009.
[72] Microsoft. (2012). *Introduction to LINQ*. Available: http://msdn.microsoft.com/en-us/library/vstudio/bb397897.aspx (Last Accessed: 5/6/2013).
[73] Microsoft. (2012). *Lambda Expressions (C# Programming Guide)*. Available: http://msdn.microsoft.com/en-us/library/vstudio/bb397687.aspx (Last Accessed: 5/6/2013).
[74] Deep Blue Sky. *HTML5 & CSS3 Support*. Available: http://fmbip.com/litmus/ (Last Accessed: 9/8/2013).
[75] w3Schools.com. *HTML5 Introduction*. Available: http://www.w3schools.com/html/html5_intro.asp (Last Accessed: 7/8/2013).
[76] Microsoft. (2013). *Visual Studio Updates* [Online]. Available: http://www.microsoft.com/visualstudio/eng/visual-studio-update (Last Accessed: 1/5/2013).
[77] Microsoft. (2012). *SQL Server 2012 Express LocalDB*. Available: http://technet.microsoft.com/en-us/library/hh510202.aspx (Last Accessed: 7/5/2013).
[78] Microsoft. (2013). *Entity Framework* [Online]. Available: http://msdn.microsoft.com/en-us/data/ef.aspx (Last Accessed).
[79] Apache Software Foundation. (2011). *Apache™ Subversion*. Available: http://subversion.apache.org/ (Last Accessed: 7/8/2013).
[80] S. Diomidis. (2012) Git. 100-101. Available: http://doi.ieeecomputersociety.org/10.1109/MS.2012.61
[81] Microsoft. (2013). *Team Foundation Server*. Available: http://msdn.microsoft.com/en-gb/vstudio/ff637362.aspx (Last Accessed: 9/8/2013).
[82] Microsoft. (2013). *Team Foundation Service* [Online]. Available: http://tfs.visualstudio.com/ (Last Accessed: 1/5/2013).
[83] S. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer*: Wiley, 2003.
[84] Microsoft ASP.NET Team. (2009). *Validation with the Data Annotation Validators*. Available: http://www.asp.net/mvc/tutorials/older-versions/models-(data)/validation-with-the-data-annotation-validators-cs (Last Accessed: 6/7/2013).
[85] E. v. d. Valk, "the difference between model-view-viewmodel and other separated presentation patterns," vol. 2013, ed. MSDN Blogs, 2009.
[86] T. Reenskaug, "Models-views-controllers," *Technical note, Xerox PARC*, vol. 32, p. 55, 1979.
[87] D. Trowbridge, *Enterprise Solution Patterns Using Microsoft .Net: Version 2.0 : Patterns & Practices*: Microsoft Corporation, 2003.
[88] J. Gossman, "Introduction to Model/View/ViewModel pattern for building WPF apps," vol. 2013, ed, 2005.
[89] J. Smith, "WPF apps with the model-view-ViewModel design pattern," *MSDN magazine*, 2009.
[90] S. Massey. (2011). *Test Driving the MVVM Pattern with ZK Ajax*. Available: http://java.dzone.com/articles/test-driving-mvvm-pattern-zk-0 (Last Accessed: 14/8/2013).
[91] S. Sanderson. *KnockoutJS* [Online]. Available: http://knockoutjs.com/ (Last Accessed: 1/5/2013).
[92] S. Sanderson. (2010). *Introducing Knockout, a UI library for JavaScript*. Available: http://blog.stevensanderson.com/2010/07/05/introducing-knockout-a-ui-library-for-javascript/ (Last Accessed: 12/8/2013).

[93] C. A. Gutwin, M. Lippold, and T. C. N. Graham, "Real-time groupware in the browser: testing the performance of web-based networking," presented at the Proceedings of the ACM 2011 conference on Computer supported cooperative work, Hangzhou, China, 2011.
[94] SignalR. *ASP.Net SignalR* [Online]. Available: http://signalr.net/ (Last Accessed: 1/5/2013).
[95] P. Fletcher. (2013). *Introduction to SignalR*. Available: http://www.asp.net/signalr/overview/getting-started/introduction-to-signalr (Last Accessed: 12/8/2013).
[96] L. Corral, A. Sillitti, G. Succi, A. Garibbo, and P. Ramella, "Evolution of Mobile Software Development from Platform-Specific to Web-Based Multiplatform Paradigm," presented at the Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software, Portland, Oregon, USA, 2011.
[97] S. Murugesan, "Attitude towards testing: a key contributor to software quality," in *Software Testing, Reliability and Quality Assurance, 1994. Conference Proceedings., First International Conference on*, 1994, pp. 111-115.
[98] J. E. Heiser, "An overview of software testing," in *AUTOTESTCON, 97. 1997 IEEE Autotestcon Proceedings*, 1997, pp. 204-211.
[99] Microsoft. (2013). *Unit Testing Framework*. Available: http://msdn.microsoft.com/en-us/library/ms243147.aspx (Last Accessed: 6/7/2013).
[100] D. Doomen. (2013). *Fluent Assertions*. Available: https://github.com/dennisdoomen/FluentAssertions (Last Accessed: 6/7/2013).
[101] J. N. Buxton and B. Randell, *Software engineering techniques: report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*: NATO Science Committee, 1970.
[102] M. Rouse. (2011). *Agile Manifesto* [Online]. Available: http://searchcio.techtarget.com/definition/Agile-Manifesto (Last Accessed: 30/4/2013).
[103] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, *et al.*, "Manifesto for Agile Software Development," in *Manifesto for Agile Software Development*, ed, 2001.

# Appendix

## Appendix A: Agile Manifesto

The Agile Manifesto was written in February of 2001, is a "*formal proclamation of four key values and 12 principles to guide an iterative and people-centric approach to software development*" [102]. The participants found consensus around four main values, which quoted as below:

> "*We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> *Individuals and interactions over processes and tools*
> *Working software over comprehensive documentation*
> *Customer collaboration over contract negotiation*
> *Responding to change over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more."* [103]

## Appendix B: Business Problem Analysis

This section intended to describe problems in the context of restaurant business. Based on these problems, possible improvements could be achieve by automating the process with assist of system that will be developed. They will later contribute to part of requirements specification of the system.

The major drive behind developing WCRS is the growing need of systematic approach to manage and assist restaurant operations. Some of the necessary process steps, issues, and management requirements are: receiving orders from customer; preparing the orders; keeping sufficient materials; and managing customer reservation – these are all are common requirements in any restaurant. Each operation will generate substantial amount of business data, which could be hard to maintain – if based on old-fashioned paper based mechanisms. For instance, collecting order information from customer could be challenging for waiter – if all he has are paper-based methodologies. Commonly, there are three major methods [4]:

i) Verbal confirmation;

ii) Order sheet; and

iii) Point-of-Sales (POS) system.

The order sheet method involves taking note of ordered items; while verbal confirmation requires waiter to memorize order details. POS system stores order information digitally, e.g. based on customer table and ordered items, and prints order sheet for kitchen. The first two methods are often prone to human error as miscommunication among waiter, customer, and chief may occur. Wrong order and overlook orders (or order items) are typical scenarios observed in almost every restaurant. The POS method only focuses on specific type of devices and restaurant need to train staff to use them. Hence, it requires proper collection and presentation of restaurant information.

Existing methods are also unable to synchronize information among restaurant staff effectively. A systematic approach needs to be taken, such as:

i) A waiter need to input order information and check if every ordered item had served;
ii) a chef need to know the order details and sequence to prepare food; and
iii) a cashier will need table information to process payment.

Often, order sheet may be lost and order sequence cannot be guaranteed when stacked up – or placed on pegs in the chiefs' kitchen preparation area. Paper based kitchen order systems are unable to reflect kitchen status in a timely and efficient manner because waiters have to manually enquire if the kitchen is preparing specific order, or specific orders or food items are ready to be serve to particular customers. Moreover, dining menu will be updated occasionally to introduce new food items and promotions [9]. This involves enormous time and effort in maintaining the changes. Thus, there is a need for mechanism to share information in real time.

# Appendix C: Functional Requirement

Table C.1 is a list of functional requirements of the prototype system. Each functional requirement has a functional requirement number, which can be traced back when testing and evaluating the final prototype. They also have priority of scale 1-3 to indicate important level of the functions and their implement sequence.

Table C.1 : Functional Requirements of WCRS.

| Requirement Number | Description | Priority ( 1-3) | Level of Risk (1 -3) |
|---|---|---|---|
| **FR1** | Creation and management of recipe collection | 1 | 1 |
| **FR2** | Creation and management of menu | 1 | 1 |
| **FR3** | Submission and management of order | 1 | 3 |
| **FR4** | Mean to interact with pending orders in Kitchen | 1 | 3 |
| **FR5** | Order processing and notification order status on cook completion | 1 | 3 |
| **FR6** | Payment computation for order | 1 | 2 |
| **FR7** | Generation of Bill and associated VAT | 1 | 3 |
| **FR8** | Mean to collect and store payment transaction details | 1 | 2 |
| **FR9** | Material inventory level monitoring; | 1 | 3 |
| **FR10** | Recording and maintaining employee information | 1 | 2 |
| **FR11** | Reporting of sales and orders | 1 | 3 |
| **FR12** | Mean to view and search order history | 1 | 2 |
| **FR13** | Adjustment of Menu available time | 2 | 2 |
| **FR14** | Creation and management of reservation | 2 | 3 |
| **FR15** | Processing order for dine-in and take away order | 2 | 2 |
| **FR16** | Reporting on materials usage | 3 | 3 |
| **FR17** | Mechanism for cancelation and changing order | 3 | 3 |
| **FR18** | Mean to attach and store recipe photo | 3 | 3 |
| **FR19** | Creation of composite menu item | 3 | 3 |

The associated level of risk defined how risky it would affect the project if that particular functional requirement failed to deliver. They are described as below:

1 - Could be solved with minimum challenges.
2 - Requires technical research to overcome challenges.
3 - Requires enormous effort and research to tackle complex problems.

# Appendix D: Non-Functional Requirement

Following table is a list of non-functional requirements of the prototype system. Each requirement has a references number which traceable to testing. Non-functional requirement address usability issues such as ease of use and performance. It employs the same priority and level of risk scale in Appendix C

Table D.2: Non-Functional Requirements of WCRS.

| **Requirement Number** | **Description** | **Priority ( 1-3)** | **Level of Risk (1 -3)** |
|---|---|---|---|
| **NFR1** | Mean to interact with external data storage | 1 | 1 |
| **NFR2** | Mean to provide timing notification of changes in operation data | 1 | 3 |
| **NFR4** | Mean to verify and validate user input | 2 | 3 |
| **NFR5** | Mean to provide meaningful exception information | 2 | 2 |
| **NFR6** | Accessibility for mobile devices | 2 | 2 |
| **NFR7** | Authentication and Access control | 2 | 2 |
| **NFR8** | Mean to provide informative representation of report | 3 | 2 |

# Appendix E: Message Descriptions

Following list all the message descriptions for the context diagram in Figure 3.2.

| **Message : Table Status** | |
|---|---|
| **Data Content** | Contains information about availability of the table |
| **Arrival Pattern** | Occurs asynchronously when waiter submit table status after customer leave. |

| **Message : Order Status** | |
|---|---|
| **Data Content** | Contains preparing status of ordered item in kitchen |
| **Arrival Pattern** | Occurs asynchronously when kitchen chef update the status on cook completion. |

| **Message : Menu Info** | |
|---|---|
| **Data Content** | Contains available menu group and recipe information for ordering |
| **Arrival Pattern** | Occurs asynchronously when waiter request the browse menu for ordering |

| **Message : Payment Data** | |
|---|---|
| **Data Content** | Contains payment information paid amount and methods |
| **Arrival Pattern** | Occurs asynchronously when cashier process payment from customer |

| **Message : Bill Information** | |
|---|---|
| **Data Content** | Contains billing information for orders such as information such as total amount and taxable amount |
| **Arrival Pattern** | Occurs asynchronously when cashier generate bill for customer payment |

| **Message : Recipe** | |
|---|---|
| **Data Content** | Contains recipe information such as title and associated materials for stock control |
| **Arrival Pattern** | Occurs asynchronously when chef create or update recipe data |

| **Message : Reservation Info** | |
|---|---|
| **Data Content** | Contains reservation details such as date and customer contact |
| **Arrival Pattern** | Occurs asynchronously when host receive reservation request from customer |

| **Message : Inventory Status** | |
|---|---|
| **Data Content** | Contains material information and name and quantity |
| **Arrival Pattern** | Occurs asynchronously when chef trying to associate materials to recipe |

| **Message : Menu plan** | |
|---|---|
| **Data Content** | Contains menu plan such as availability time and associated recipe |
| **Arrival Pattern** | Occurs asynchronously when manager plan menu availability and associated recipes |

| **Message : Staff Information** | |
|---|---|
| **Data Content** | Contains staff data such as name, gender and age |
| **Arrival Pattern** | Occurs asynchronously when manager register new staff to the system |

| **Message : Kitchen expenditure** | |
|---|---|
| **Data Content** | Contains materials usage information |
| **Arrival Pattern** | Occurs asynchronously when manager view usage report |

| **Message : Sales report** | |
|---|---|
| **Data Content** | Contains sales information about menu and recipe |
| **Arrival Pattern** | Occurs asynchronously when manager view sales report |

| **Message : Supplies data** | |
|---|---|
| **Data Content** | Contains the information about received purchase materials |
| **Arrival Pattern** | Occurs asynchronously when purchase order been fulfilled by supplier |

# Appendix F: Use Case Descriptions

Following list all the use case description for the use case diagram in Figure 3.3.

| **Use Case: Manage table reservation** | |
|---|---|
| **Description** | User intended to create, update or cancel table reservation |
| **Actors** | Host |
| **Scenario** | The use case starts when user elects to manage table reservation from Table command menu. The system will present list of opening reservations. The user then can elect either create, update or cancel reservation actions. Create and update action will require user to specify time and reservation details. |

| **Use Case: Update table status** | |
|---|---|
| **Description** | User intended to update table status to reflect available seats |
| **Actors** | Host, Waiter |
| **Scenario** | The use case starts when user elects to update table status from Table command menu. The system will present all the tables information in restaurant. The user then can elect to update table to occupy, available and reserved status. |

| **Use Case: Submit order** | |
|---|---|
| **Description** | User intended to submit a new order to kitchen |
| **Actors** | Waiter |
| **Scenario** | The use case starts when user elects to submit order from order command menu. The system will prompt user to specify table number. Then, it present menu for add and remove order item. The user need to specify quantity when add an order item. After the action is completed, the user confirms to submit order information for processing. |

| **Use Case: View menu items** | |
|---|---|
| **Description** | User intended to view menu information |
| **Actors** | Waiter, Chef |
| **Scenario** | The use case must be started by other use cases. When user enter this use case, The system will present collections of defined menu items. The user can elect to browse by category or filter for particular menu item. |

| **Use Case: Manage order status** | |
|---|---|
| **Description** | User intended to update order status of submitted order |
| **Actors** | Waiter, Chef |
| **Scenario** | The use case starts when waiter elects to cancel order or chef elect to update cooked order item. The system present ordered items and associated status to each item. User selects the order that need to be changed. When order status updated, it will be synchronized to waiter and chef view. |

| **Use Case: View ordered item** | |
|---|---|
| **Description** | User intended to view ordered items of orders |
| **Actors** | Waiter, Chef, Cashier |
| **Scenario** | The use case starts when user elects to view order detail from Order command menu. The system prompts user to specify order number. Then, it presents the list of ordered items. |

| **Use Case: Generate bill** | |
|---|---|
| **Description** | User intended to print bill for customer payment |
| **Actors** | Cashier |
| **Scenario** | The use case starts when user elects to print bill from the Payment command menu. The system prompts user to specify order number. Then, the system will calculate the required amount to be paid (including tax and other charge). After the action is completed, the user confirms to get generated bill. |

| **Use Case: Process payment** | |
|---|---|
| **Description** | User intended to process payment for customer |
| **Actors** | Cashier |
| **Scenario** | The use case starts when user elects to process payment from Payment command. The system prompts user to specify order number. Then, the user selects the payment methods. Then, the user confirms to persist the result. Finally, the system will present receipt information. |

| **Use Case: Manage employee information** | |
|---|---|
| **Description** | User intended to register, update or delete employee information |
| **Actors** | Manager |
| **Scenario** | The use case starts when user elects to manage employee information from Management command menu. The system will present list of employee. The user then can elect register, update or delete actions. Register and Update employee will prompt user to input the details. Delete action will remove employee information from database. |

| **Use Case: Generate Reports** | |
|---|---|
| **Description** | User intended to view the report of restaurant operations |
| **Actors** | Manager |
| **Scenario** | The use case starts when user elects to view report from Management command menu. The system will then present generated reports. |

| Use Case: Calculate Recipe Cost | |
|---|---|
| **Description** | User intended to know if particular recipe is profitable |
| **Actors** | Manager |
| **Scenario** | The use case starts when user elects to view recipe details after associate relevant materials. The system will present the recipe details and it associated cost. |

| Use Case: Manage menu plan | |
|---|---|
| **Description** | User intended to create, update or delete menu item |
| **Actors** | Manager |
| **Scenario** | The use case starts when user elects to manage menu plan from Management module. The system will present collection of created menus. The user then can elect Create, Update or Delete actions. Create and Update menu will prompt user to specify details (e.g. time available). After selected action is completed, the user confirms to persist the result. |

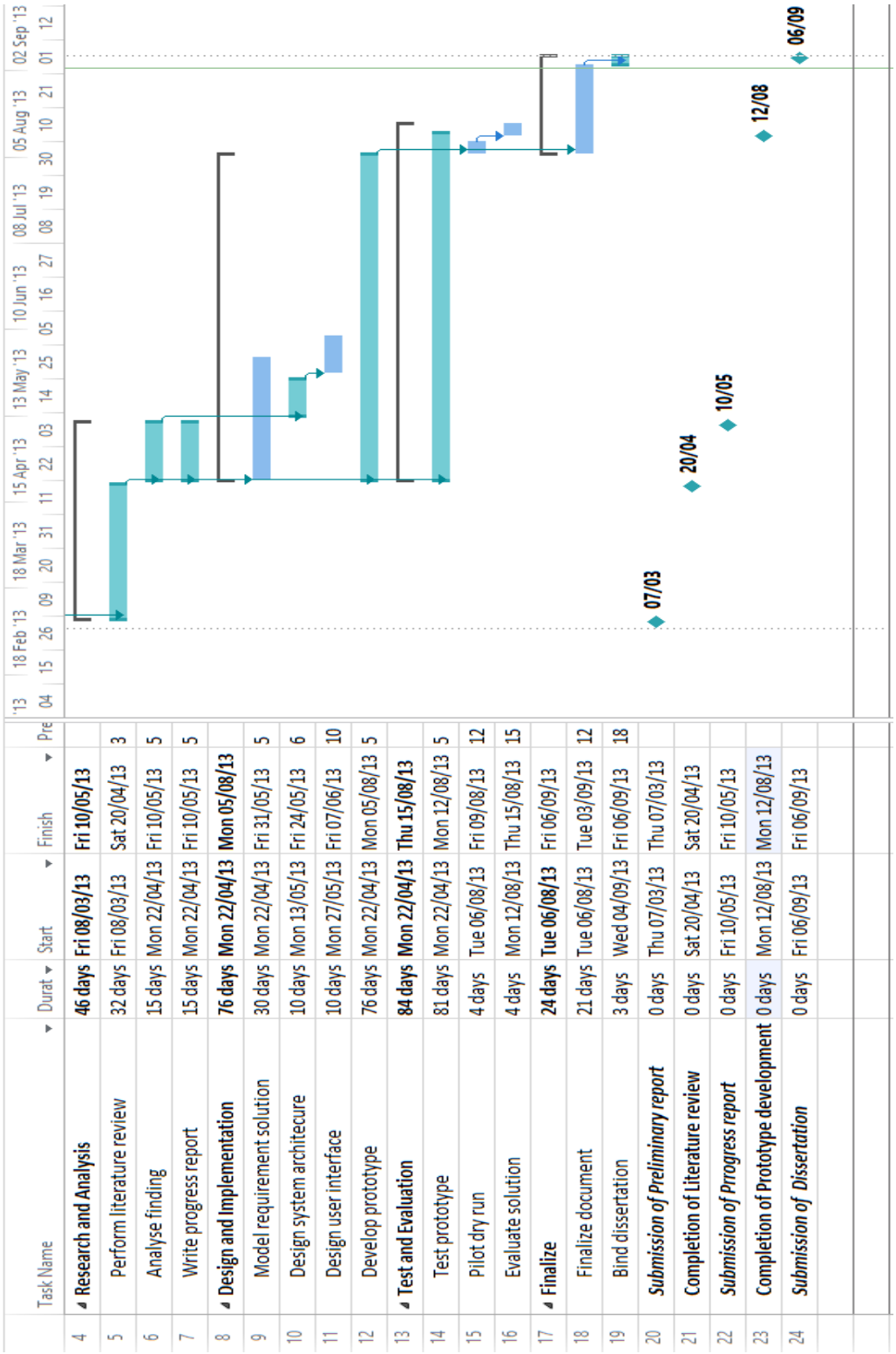
| Use Case: Plan materials supply | |
|---|---|
| **Description** | User intended to generate a purchasing order of required materials |
| **Actors** | Manager, Chef |
| **Scenario** | The use case starts when user elects to plan materials supply in Inventory menu. The system will present the collection of purchases orders. The user can elect to generate or update purchase order status. Generate purchase order will required to select recipes or specify materials to be purchased. After the actions completed, the user confirms to get a purchase order data. When purchase order is fulfil by supplier, user can update the purchase order status to add purchased material amount to inventory. |

| Use Case: Manage materials inventory status | |
|---|---|
| **Description** | User intended to update the amount of materials in the inventory during inventory reconciliation |
| **Actors** | Manager, Chef |
| **Scenario** | The use case starts when user elects to manage materials from Inventory menu. The system will present all the materials and its quantity. The user then update the quantity based on selected material. |

| Use Case: Calculate required materials | |
|---|---|
| **Description** | System need to know required materials to prepare a recipe |
| **Actors** | Waiter |
| **Scenario** | The use case starts when waiter tries to submit an order item. The system will check if they are sufficient, materials to perform the order before it allow the order to pass through. |

# Appendix G: Gantt Chart

Figure G.1 display a timeline view of the project plan. It shows all list of important activities and their due time. Several significant milestones and their dates are labelled at the end of chart. The development time is longer than the previously expected.




| | Task Name | Durat | Start | Finish | Pre |
|---|---|---|---|---|---|
| 4 | **Research and Analysis** | **46 days** | **Fri 08/03/13** | **Fri 10/05/13** | |
| 5 | Perform literature review | 32 days | Fri 08/03/13 | Sat 20/04/13 | 3 |
| 6 | Analyse finding | 15 days | Mon 22/04/13 | Fri 10/05/13 | 5 |
| 7 | Write progress report | 15 days | Mon 22/04/13 | Fri 10/05/13 | 5 |
| 8 | **Design and Implementation** | **76 days** | **Mon 22/04/13** | **Mon 05/08/13** | |
| 9 | Model requirement solution | 30 days | Mon 22/04/13 | Fri 31/05/13 | 5 |
| 10 | Design system architecure | 10 days | Mon 13/05/13 | Fri 24/05/13 | 6 |
| 11 | Design user interface | 10 days | Mon 27/05/13 | Fri 07/06/13 | 10 |
| 12 | Develop prototype | 76 days | Mon 22/04/13 | Mon 05/08/13 | 5 |
| 13 | **Test and Evaluation** | **84 days** | **Mon 22/04/13** | **Thu 15/08/13** | |
| 14 | Test prototype | 81 days | Mon 22/04/13 | Mon 12/08/13 | 5 |
| 15 | Pilot dry run | 4 days | Tue 06/08/13 | Fri 09/08/13 | 12 |
| 16 | Evaluate solution | 4 days | Mon 12/08/13 | Thu 15/08/13 | 15 |
| 17 | **Finalize** | **24 days** | **Tue 06/08/13** | Fri 06/09/13 | |
| 18 | Finalize document | 21 days | Tue 06/08/13 | Tue 03/09/13 | 12 |
| 19 | Bind dissertation | 3 days | Wed 04/09/13 | Fri 06/09/13 | 18 |
| 20 | ***Submission of Preliminary report*** | 0 days | Thu 07/03/13 | Thu 07/03/13 | |
| 21 | **Completion of Literature review** | 0 days | Sat 20/04/13 | Sat 20/04/13 | |
| 22 | ***Submission of Prrogress report*** | 0 days | Fri 10/05/13 | Fri 10/05/13 | |
| 23 | **Completion of Prototype development** | 0 days | Mon 12/08/13 | Mon 12/08/13 | |
| 24 | ***Submission of Dissertation*** | 0 days | Fri 06/09/13 | Fri 06/09/13 | |

Figure G.1: Gantt Chart for MSc Dissertation.

# Appendix H: CRC Cards

Figure below shows CRC cards that been model for WRCS.

| **RecipeMaterial** | |
|---|---|
| Know quantity<br>Know comment<br>Calculate cost | Material |

| **RecipeCategory** | |
|---|---|
| Know category name<br>Know assigned recipes | Recipe |

| **Material** | |
|---|---|
| Know name<br>Know unit of measure<br>Know quantity<br>Know cost per unit<br>Change quantity | |

| **MenuGroup** | |
|---|---|
| Know title<br>Know available day<br>Know available time<br>Know cost per unit<br>Assign recipe to menu group<br>Check availability | Recipe |

| **Order** | |
|---|---|
| Know order status<br>Know date submitted<br>Know order<br>Add order items<br>Calculate order total | Table<br>Staff<br>OrderItem |

| **OrderItem** | |
|---|---|
| Know quantity<br>Know comment<br>Know recipe<br>Calculate subtotal | Recipe |

| **Table** | |
|---|---|
| Know identifier<br>Know number of seats<br>Know table status | |

| **Staff** | |
|---|---|
| Know username<br>Know name<br>Know date of birth<br>Know gender<br>Know membership<br>Know roles | Role<br>Membership |

| **Membership** | |
|---|---|
| Know creation date<br>Know password<br>Know password change date | |

| **Role** | |
|---|---|
| Know role name<br>Assign staff to roles | Staff |

| **Setting** | |
|---|---|
| Know restaurant name<br>Know phone<br>Know address<br>Know email<br>Know VAT rate<br>Know service tax rate<br>Update setting | |

| **Reservation** | |
|---|---|
| Know start date<br>Know end date<br>Know remark | |

| **Payment** | |
|---|---|
| Know order<br>Know staff<br>Know date paid<br>Know amount<br>Know tax amount<br>Know service amount<br>Know total payable amount<br>Know cash received<br>Know credit card received | Order<br>Staff |

Figure H.2: CRC Cards for Entity classes.

| Recipe Services | |
|---|---|
| Manage recipes | Recipe |
| Mange recipe categories | RecipeCategory |
| Manage recipe materials | RecipeMaterial |
| Get recipes | Database handler |

| Menu Services | |
|---|---|
| Manage menu groups | MenuGroup |
| Get available menu recipes | Recipe |
| Add recipe to menu group | Database handler |
| Remove recipe from menu group | |

| Inventory Services | |
|---|---|
| Manage materials | Material |
| | Database handler |

| Order Services | |
|---|---|
| Manage orders | Order |
| Submit order | OrderItem |
| Calculate cook able quantity | Recipe |
| | Material |
| Update order status | Table |
| Get pending orders | Database handler |

| Hall Services | |
|---|---|
| Manage tables | Table |
| Manage Reservation | Reservation |
| Update table status | Database handler |

| Payment Services | |
|---|---|
| Manage payments | Payment |
| Calculate payments | Setting |
| Accept payment | Order |
| Construct receipt | Database handler |

| Management Services | |
|---|---|
| Manage staffs | Staff |
| Get Roles | Membership |
| Manage memberships | Role |
| Manage roles | Setting |
| Add user to role | SettingNameValuePair |
| Remove user from role | Database handler |
| Load setting | |
| Update setting | |
| Save setting | |

| Database Handler | |
|---|---|
| Manage database context (connection) | Recipe |
| | RecipeCategory |
| Execute queries | RecupeMaterial |
| Commit changes | MenuGroup |
| | Order |
| | OrderItem |
| | Payment |
| | Table |
| | Reservation |
| | Staff |
| | Membership |
| | Role |
| | SettingNameValuePair |

Figure H.3 : CRC Cards for Service classes.

| Recipe Controller | |
|---|---|
| Handle recipes management request<br>Handle recipe categories management request<br>Handle recipe materials management request | RecipeServices<br>InventoryServices |

| Management Controller | |
|---|---|
| Handle staff management request<br>Handle update setting request<br>Generate report | ManagementServices |

| Menu Controller | |
|---|---|
| Handle menu groups management request<br>Handle recipes association request | RecipeServices<br>MenuServices |

| Inventory Controller | |
|---|---|
| Handle materials management request | InventoryServices |

| Order Controller | |
|---|---|
| Handle order submission request<br>Handle order status update request<br>Notify order status update | ManagementServices<br>MenuServices<br>TableServices<br>OrderServices |

| Hall Controller | |
|---|---|
| Handle table management request<br>Handle reservation request<br>Handle table status update request | HallServices |

| Payment Controller | |
|---|---|
| Handle bill request<br>Handle payment request<br>Handle receipt request | PaymentServices<br>OrderServices<br>ManagementServices |

Figure H.4: CRC Cards for Controller classes.

# Appendix I: Class Diagram

Figure below shows design class diagram for the WCRS entities. The class diagram is focus on representing the relationships between objects.

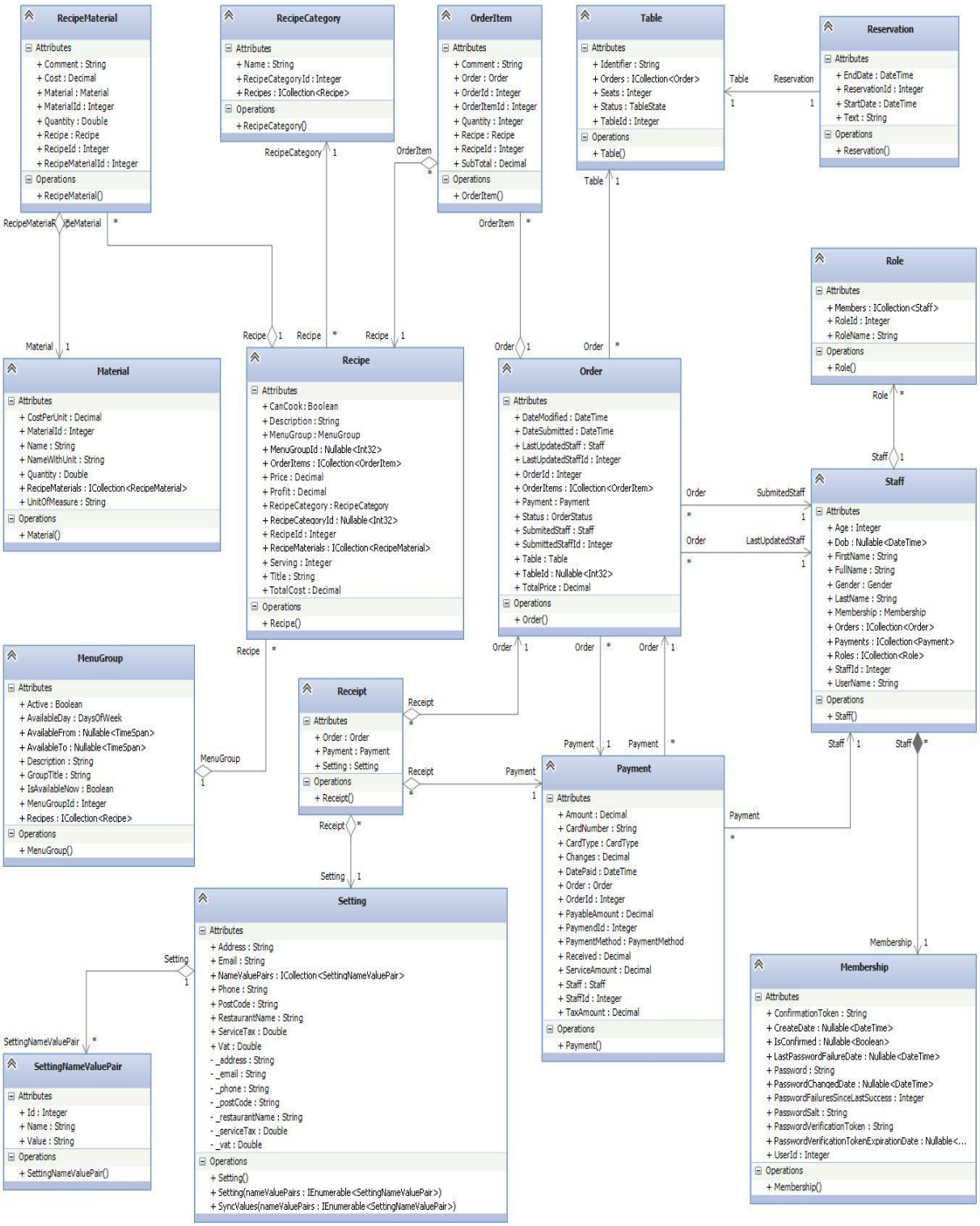



Figure I.5: Class Diagram for the WCRS entities.

# Appendix J: Sequence Diagrams

This section presents the important sequence diagrams of behaviour models.

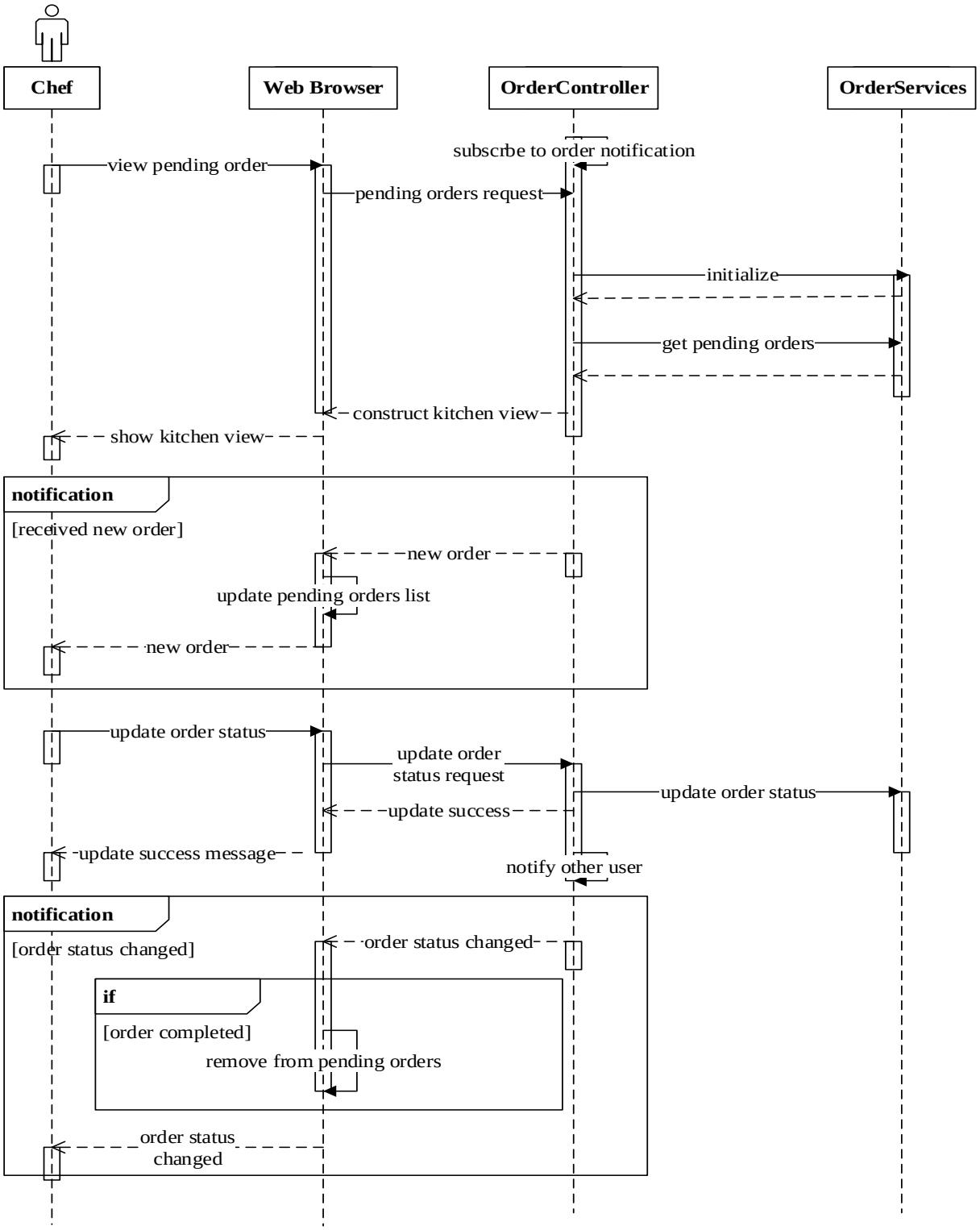



Figure J.6: Sequence Diagram for update order status.

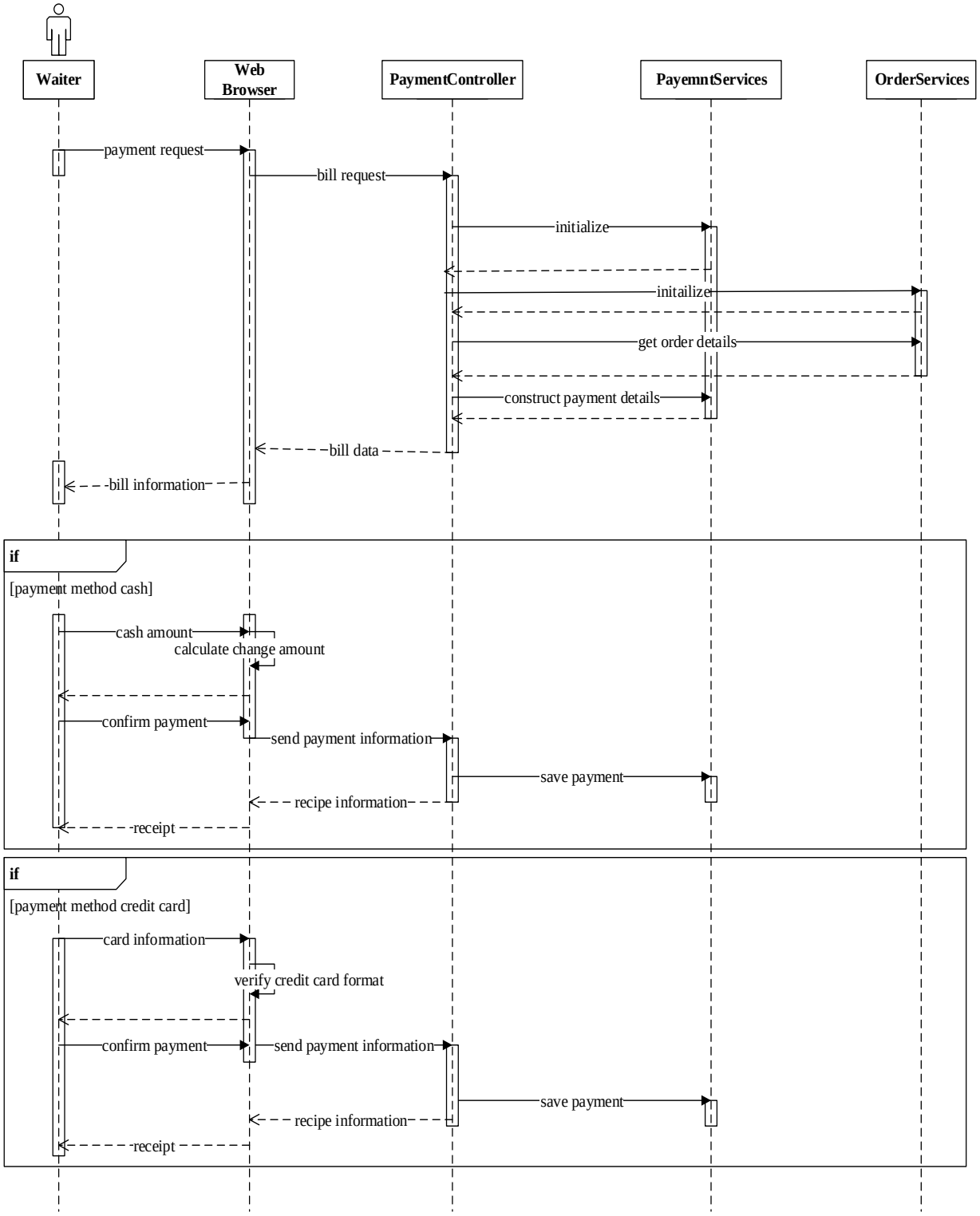



Figure J.7: Sequence Diagram for process payment.

# Appendix K: User Interface Prototyping

This sections illustrated the remaining of user interface prototyping for the HTA of submit order (refer Figure 4.13).

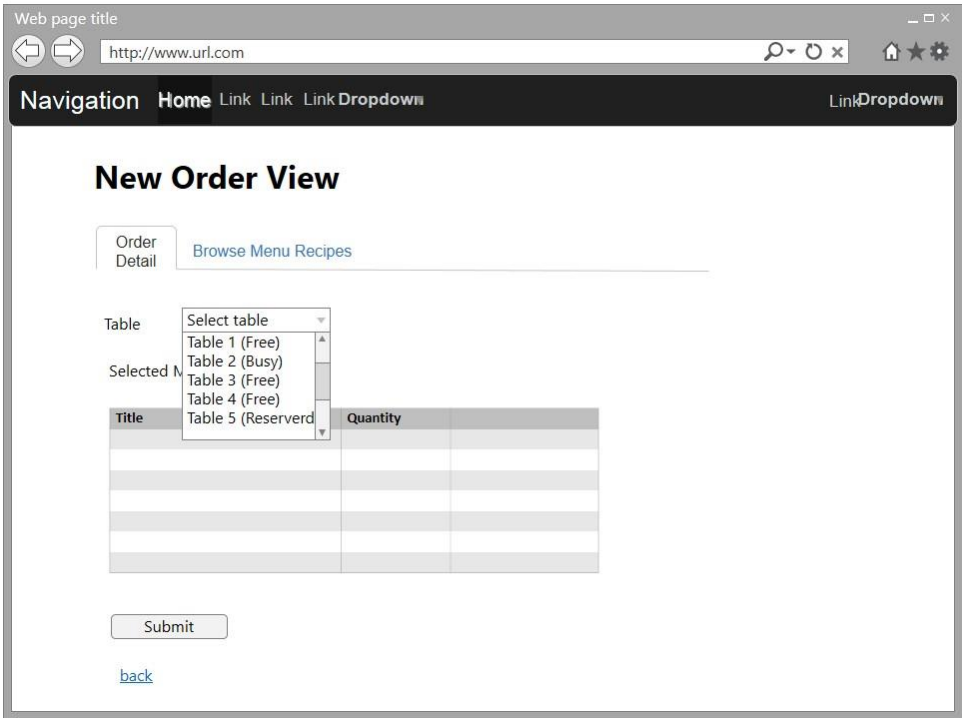



Figure K.8: Design shows user select table (HTA #2).

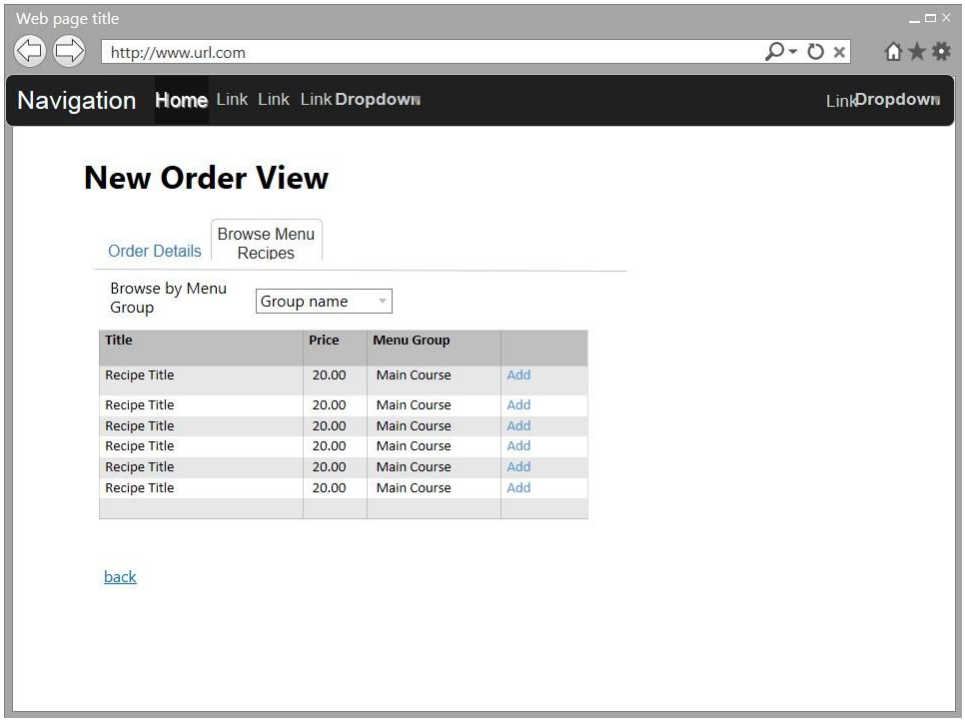



Figure K.9: Design shows user browse menu and select recipe (HTA #2.1 and #2.2).

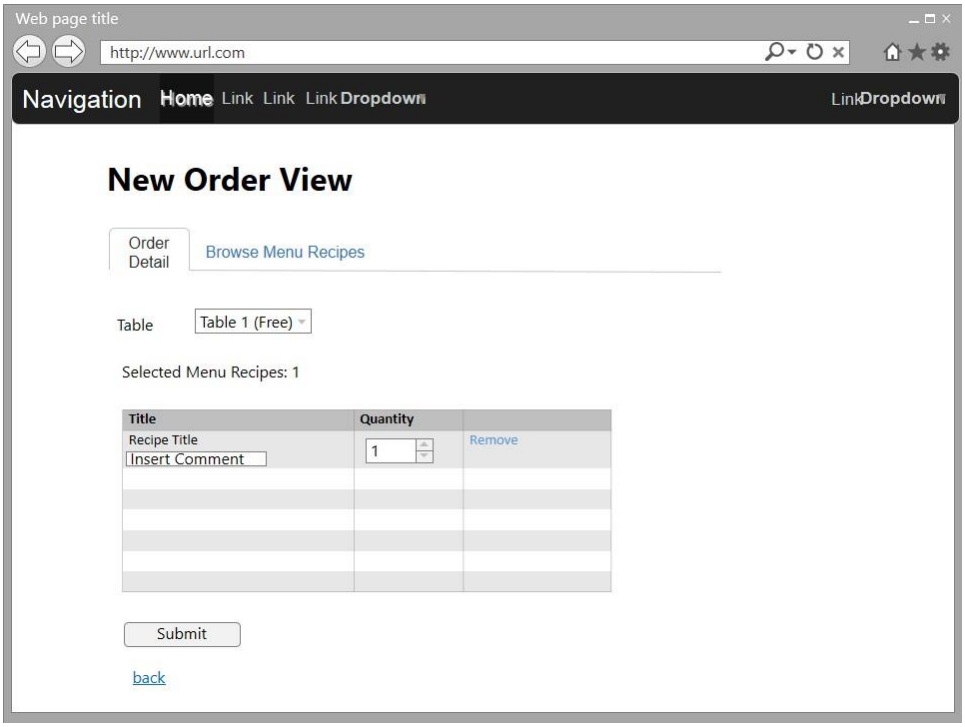



Figure K.10: Design shows user specifies quantity and comment (HTA#2.3 and HTA #2.4).

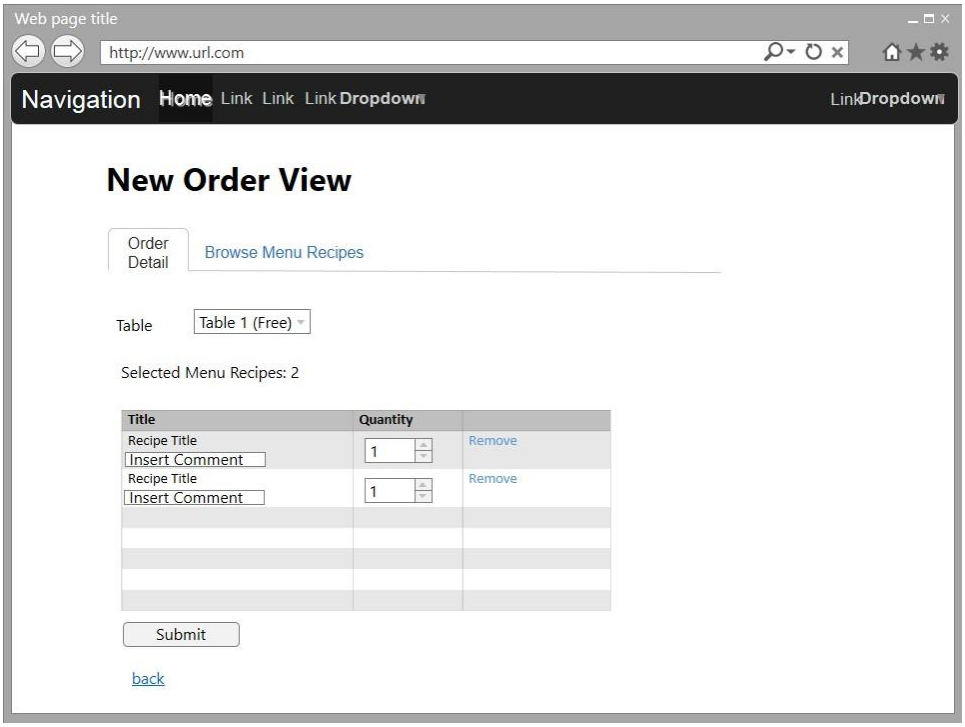



Figure K.11: Design shows user ready to submit order (HTA #4).

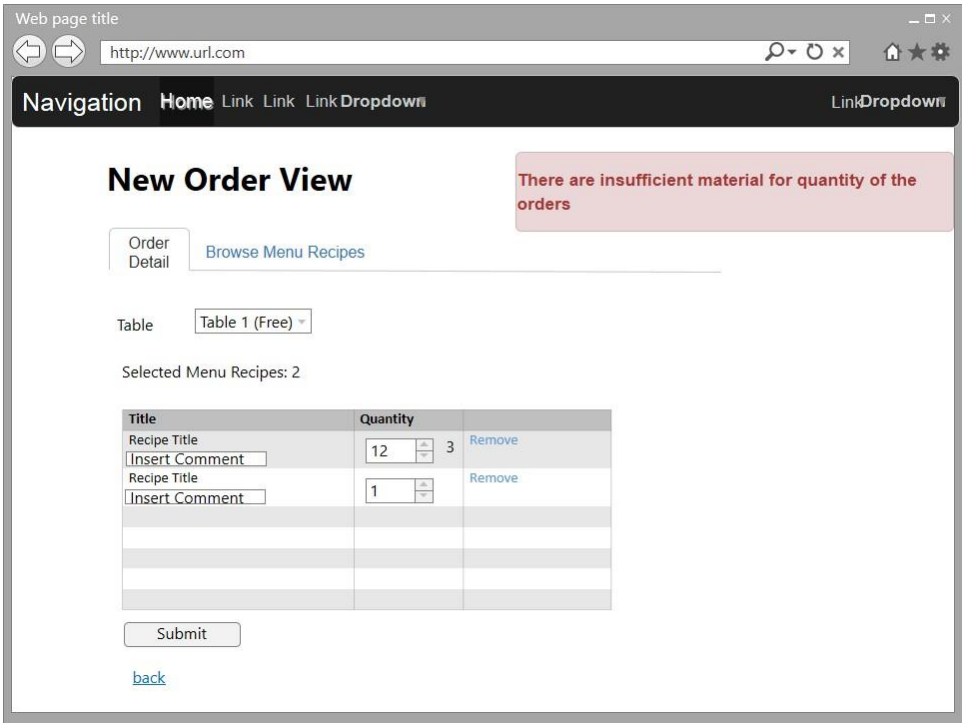



Figure K.12: Design shows insufficient material error and actual available quantity (HTA #4.1).

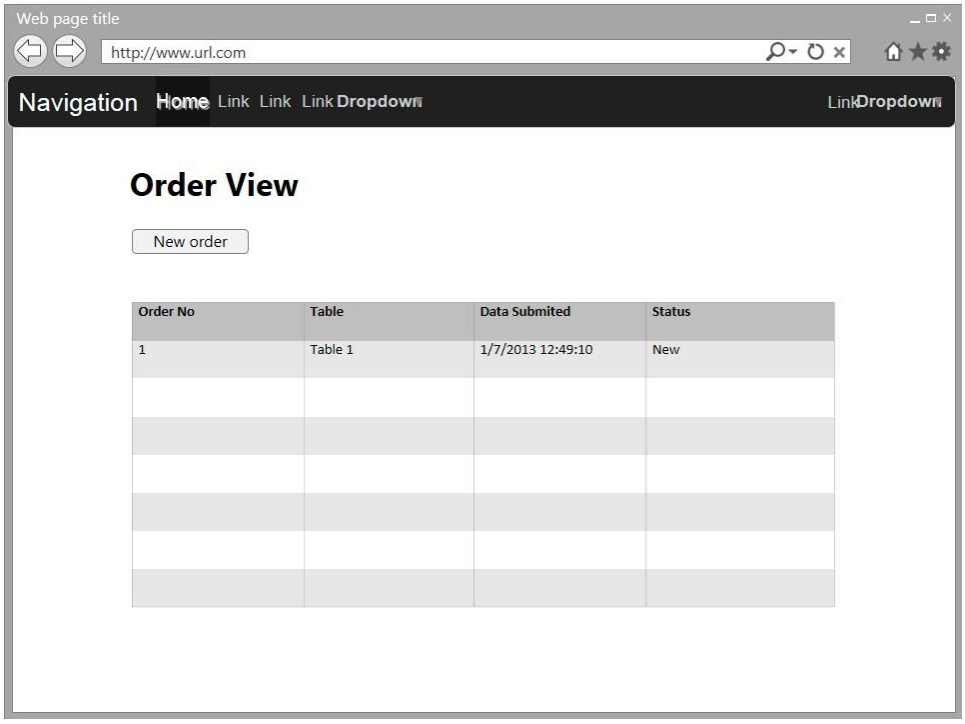



Figure K.13: Design shows that order successfully sent (HTA #5).

# Appendix L: Hierarchy Task Analysis Diagrams

This section depicts the Hierarchy Task Analysis diagrams for other non-trivial tasks.

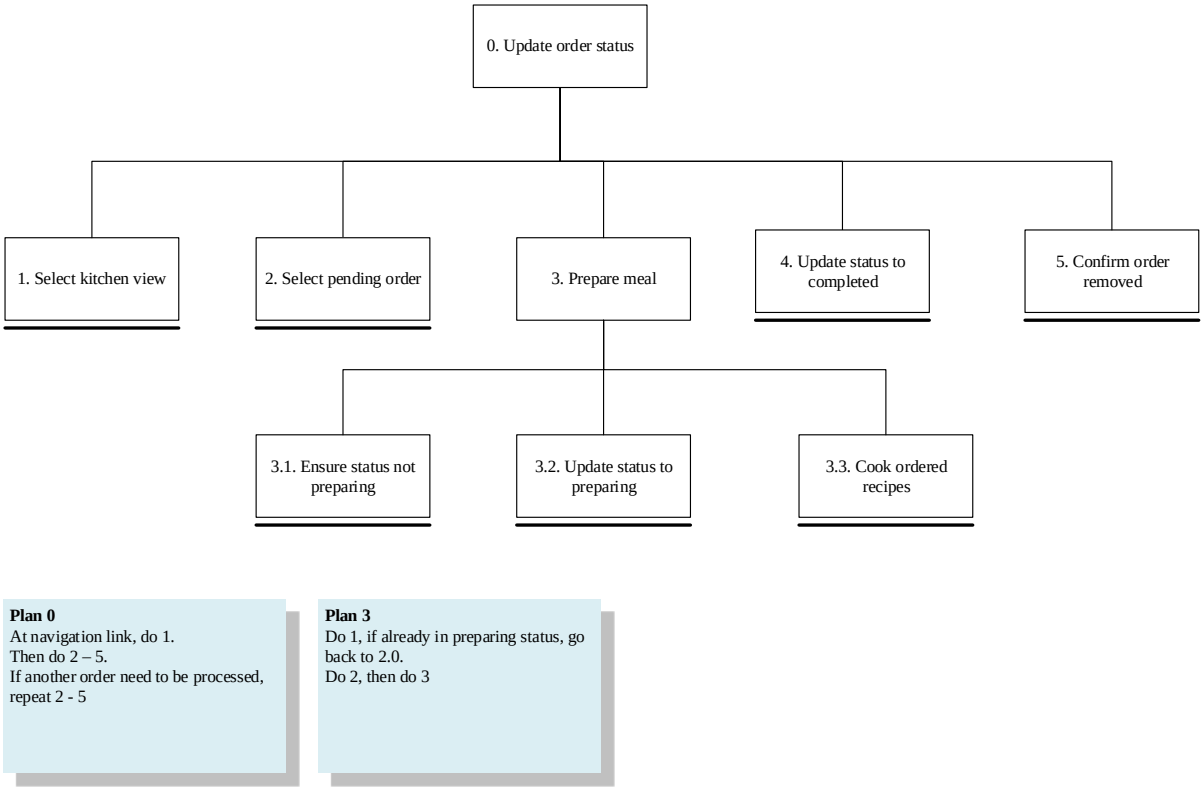



Figure L.14: HTA for update order status.

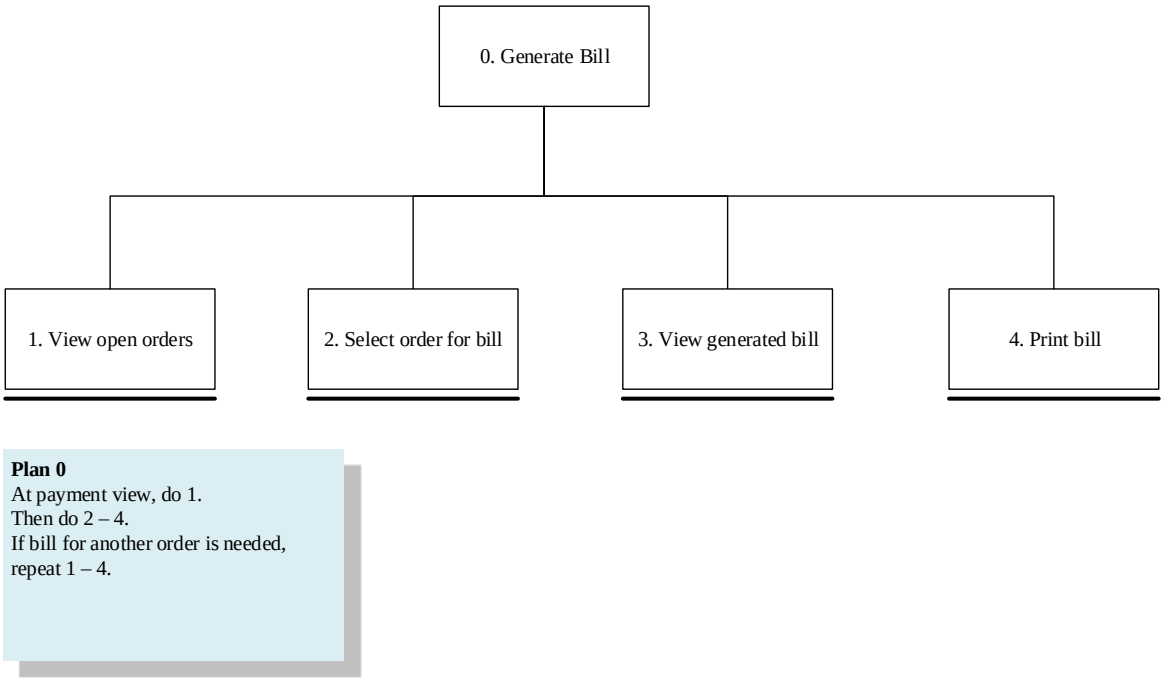



Figure L.15: HTA for generate bill.

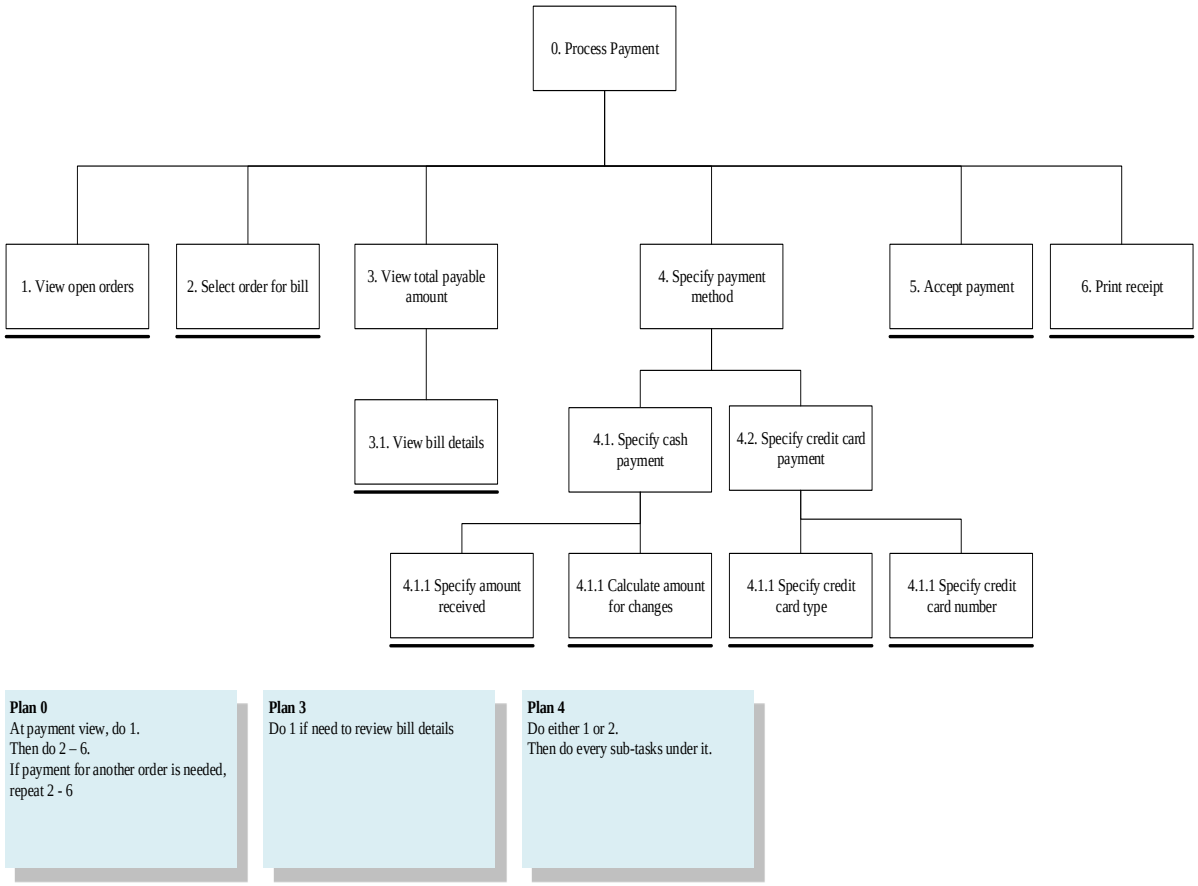



Figure L.16: HTA for process payment.

# Appendix M: Prototype Implementation Table

The following table shows the released version of the prototype, date due and its associated deliverables with incremental enhancements. A substantial amount of time has been allocated to learning the tools and integrating them. Thus, the output of each version is smaller at first and the development progress faster as the author becomes familiar with the development tools. Done definitions of the NFR are rather vague because they should be considered throughout the project; hence they are listed at the end of the table. Due to time constraints, the requirements with higher priority or interdependencies are implemented first.

Table M.3: Prototype Implementation Table.

| Prototype Version | Date of Completion | Main Deliverable |
|---|---|---|
| **P1** | 10/4/2013 | Implementation of initial projects structure and configuration of code repository, web server and database server. |
| **P2** | 24/4/2013 | Implementation of FR1, FR9 |
| **P3** | 1/5/2013 | Implementation of FR2, FR13 |
| **P4** | 22/5/2013 | Implementation of FR10, FR14 |
| **P5** | 12/6/2013 | Implementation of FR3, FR4, FR5 |
| **P6** | 3/7/2013 | Implementation of FR6, FR7, FR8 |
| **P7** | 24/7/2013 | Implementation of FR11, FR12 |
| **Final** | 12/8/2013 | NFR Implemented: NFR1, NFR2, NFR4, NFR5, NFR6, NFR7 and NFR8 |

# Appendix N: Continuous Integration Usage

This shows the usage of Team Foundation Services (Continuous Integration software) and Visual Studio for source control and builds management. Figure N.17 shows the usage of TFS for source controls with VS integration. Figure N.18 shows that the build processes are automated by TFS and provide build result information once completed.

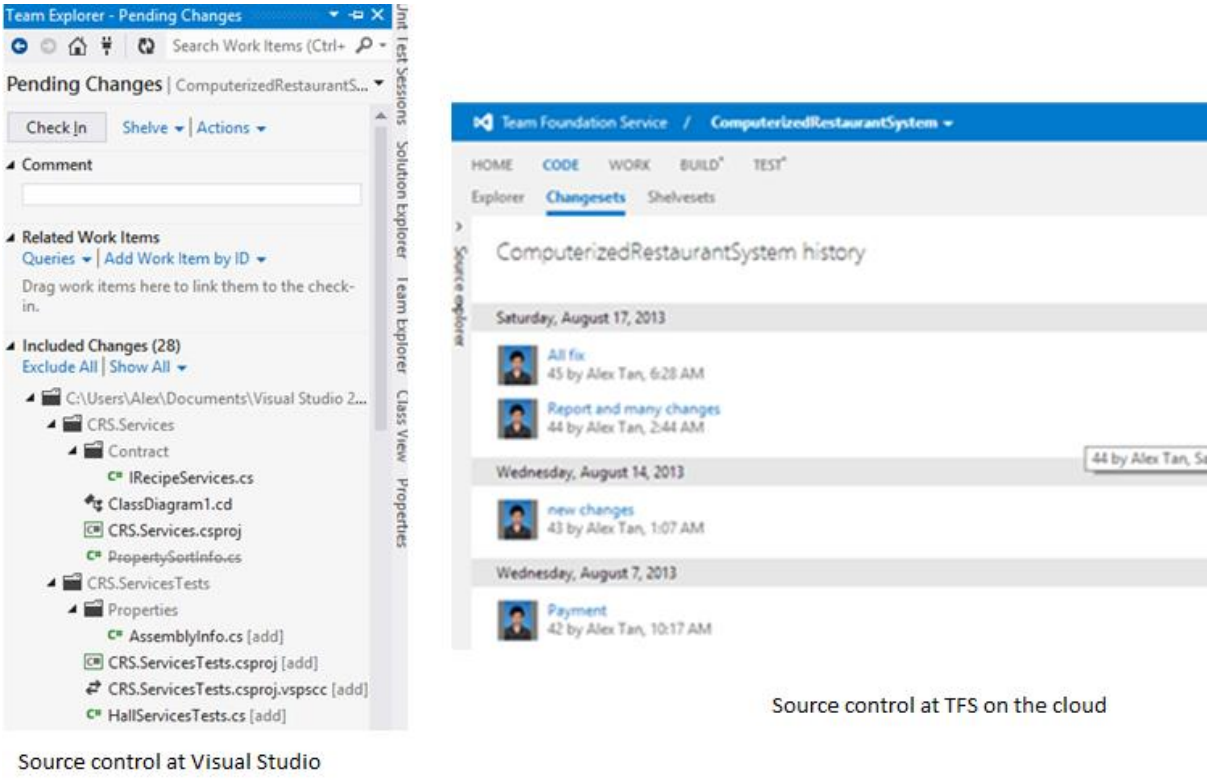



Figure N.17: Managing source code changes in Source Control.

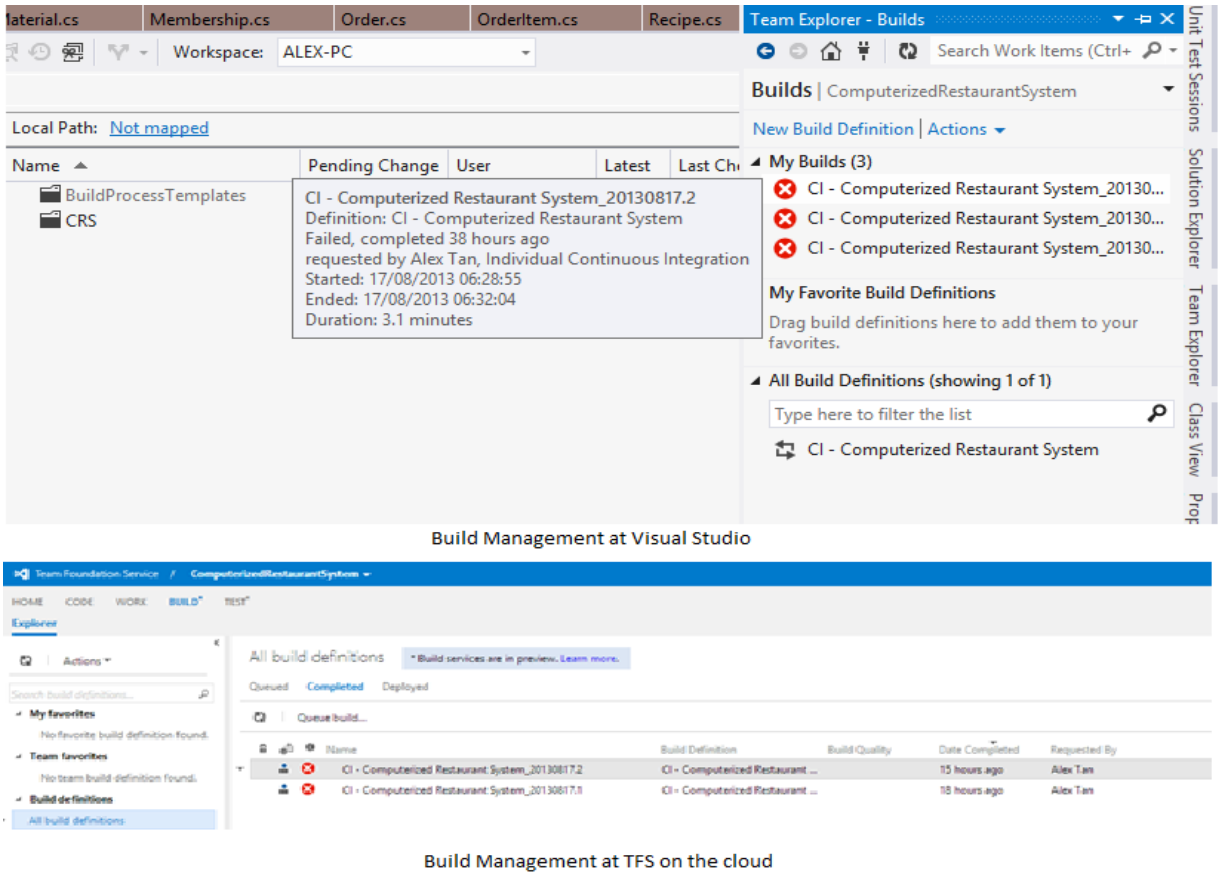



Figure N.18: Analysing build in Build Management.

# Appendix O: Screenshots

This section presents the screenshots of remaining features. This report only selects several significant screenshots to represent the features. The features are quite self-expressive once the user has followed the application UI design. They are consistent and mapped to particular tasks of the user.

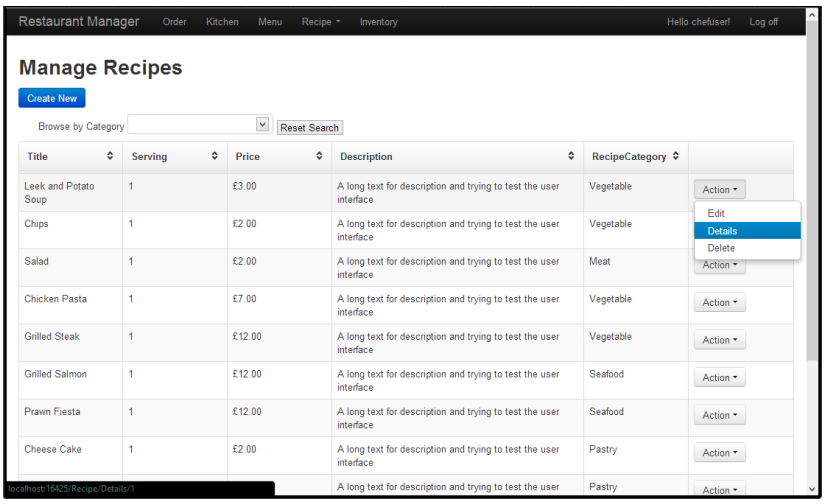



Figure O.19: A screenshot of the mange recipe view.

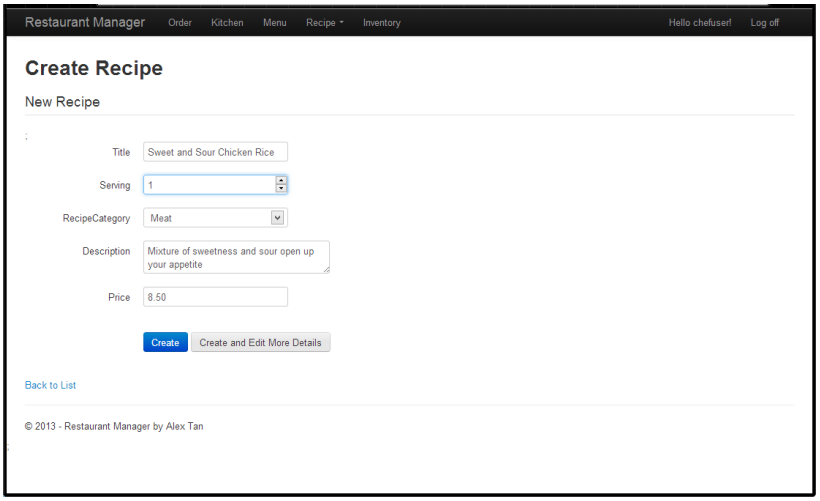



Figure O.20: A screenshot of the create recipe view.

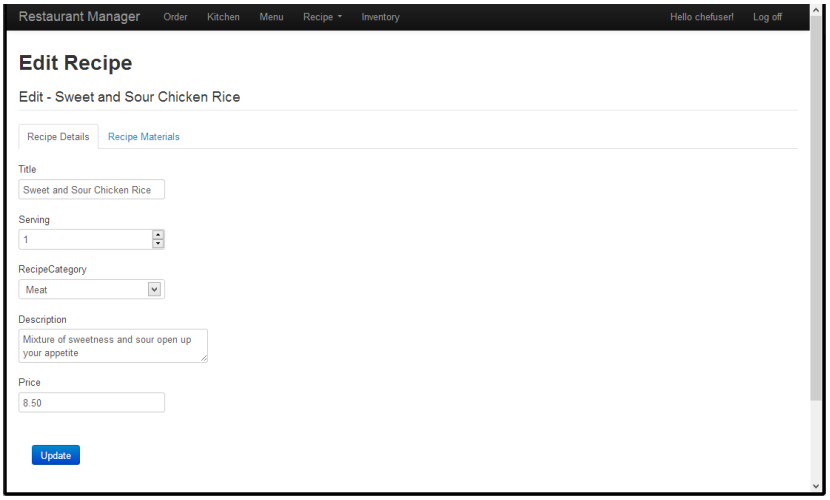



Figure O.21: A screenshot of the edit recipe view.

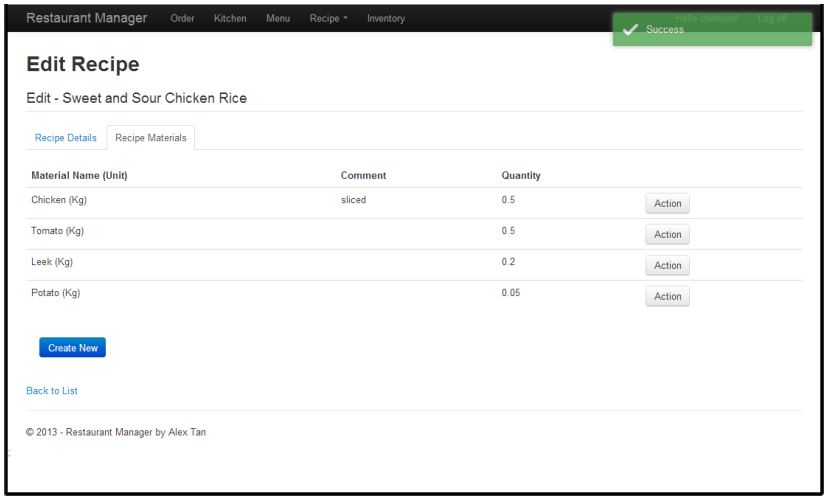



Figure O.22: A screenshot of associating materials to the recipe.

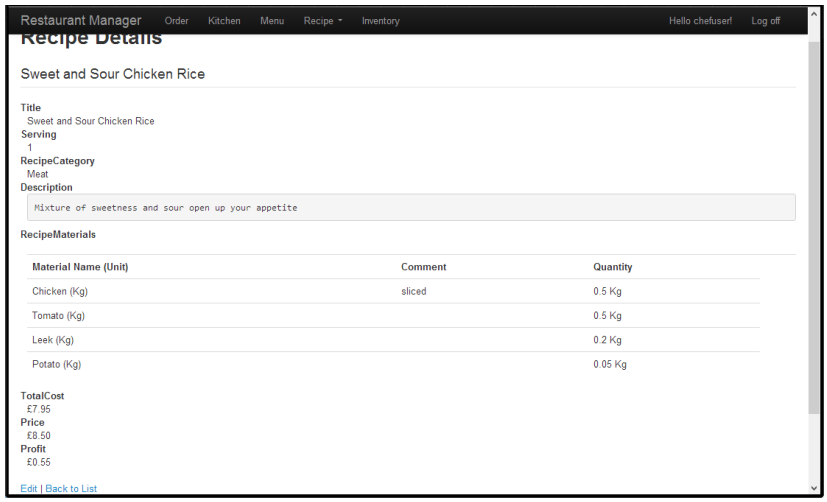



Figure O.23: A screenshot of calculated recipe cost and profit at detail view.

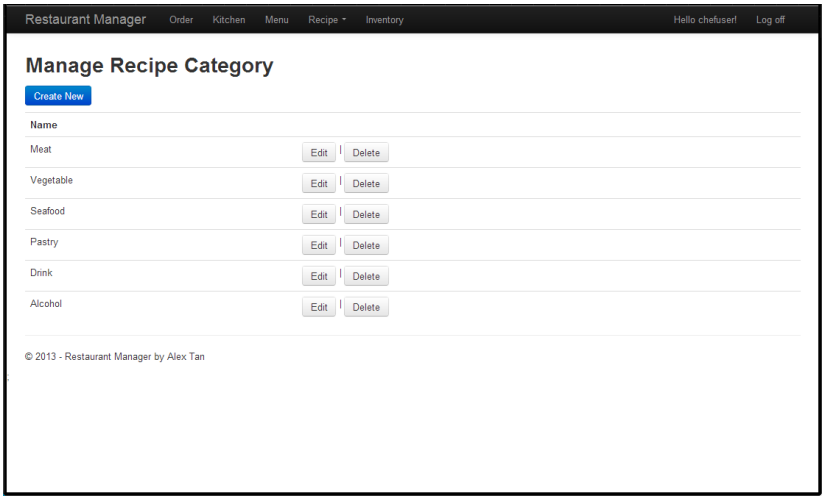



Figure O.24: A screenshot of manage recipe category view.

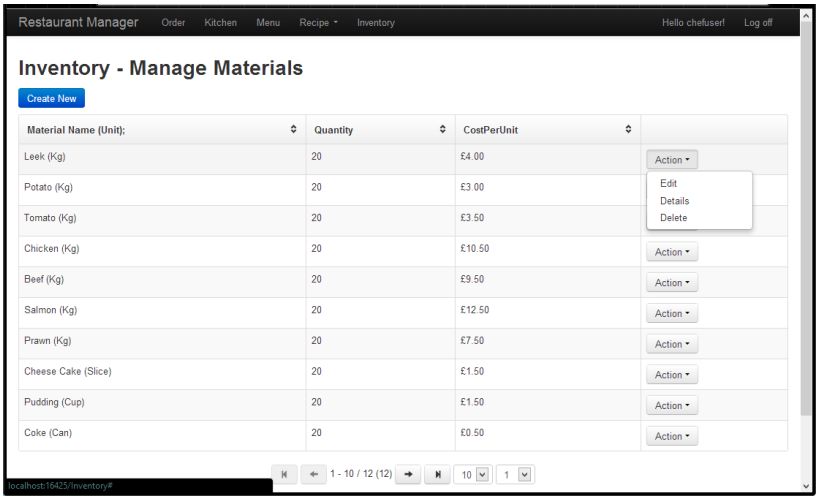



Figure O.25: A screenshot of manage materials at inventory view.

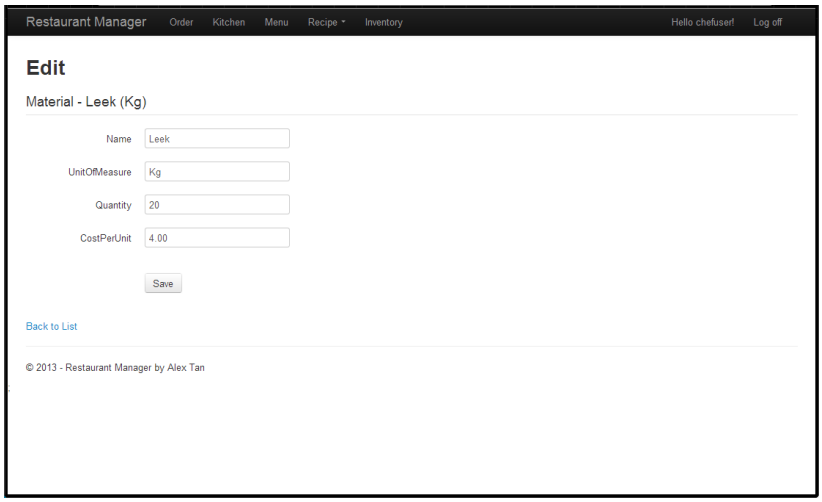



Figure O.26: A screenshot of updating materials view.

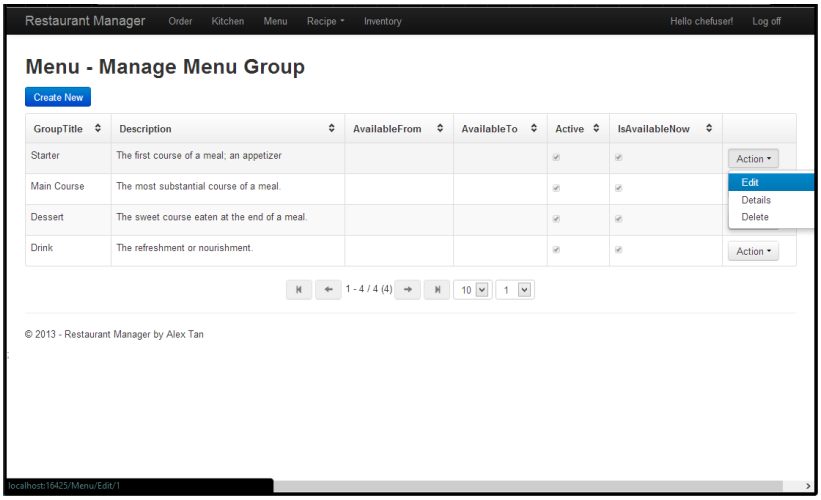



Figure O.27: A screenshot of manage menu group view.

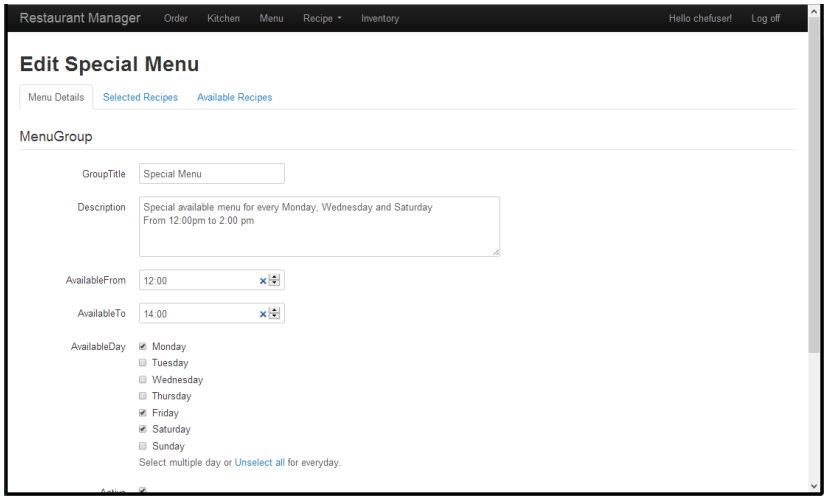



Figure O.28: A screenshot of updating menu group availability.

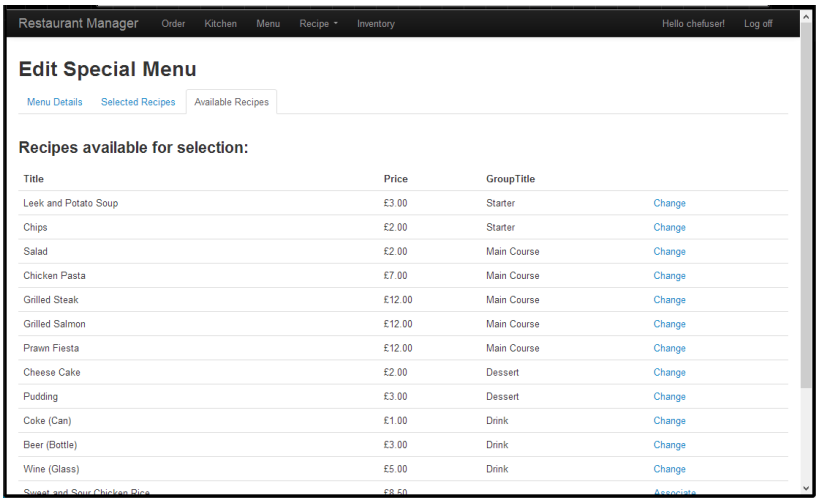



Figure O.29: A screenshot of selecting recipes for menu group.

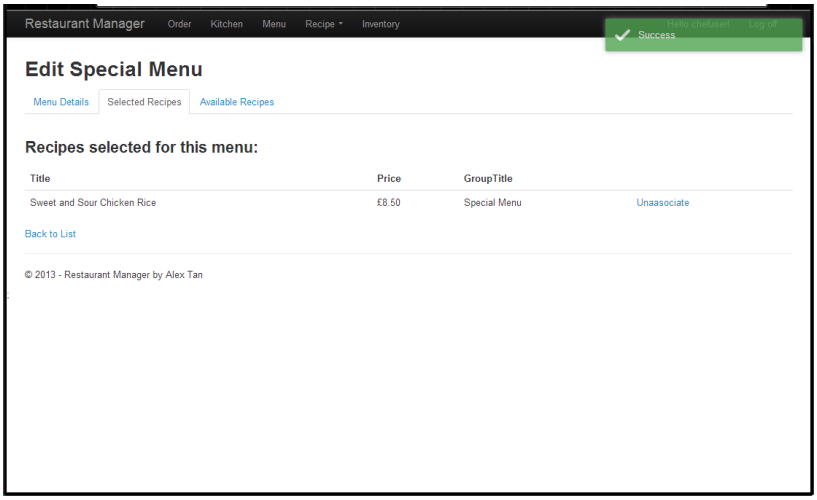



Figure O.30: A screenshot of associating recipe with menu group.

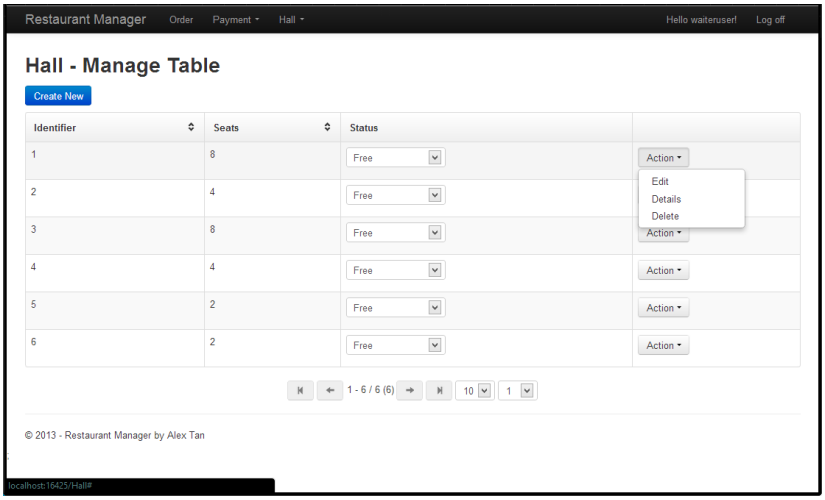



Figure O.31: A screenshot of managing table view.

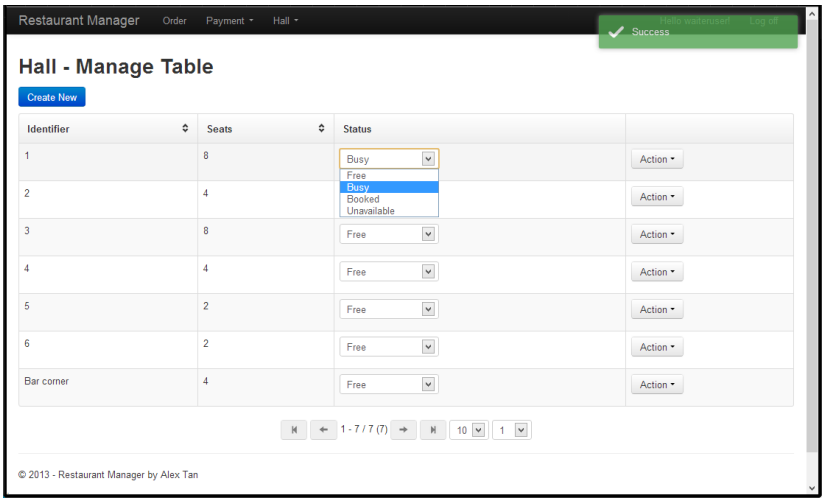



Figure O.32: A screenshot of updating table status view.

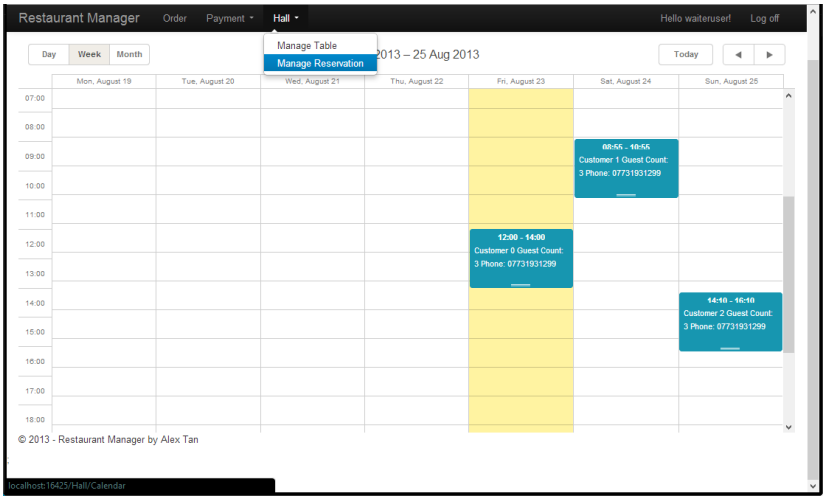



Figure O.33: A screenshot of making reservation view.

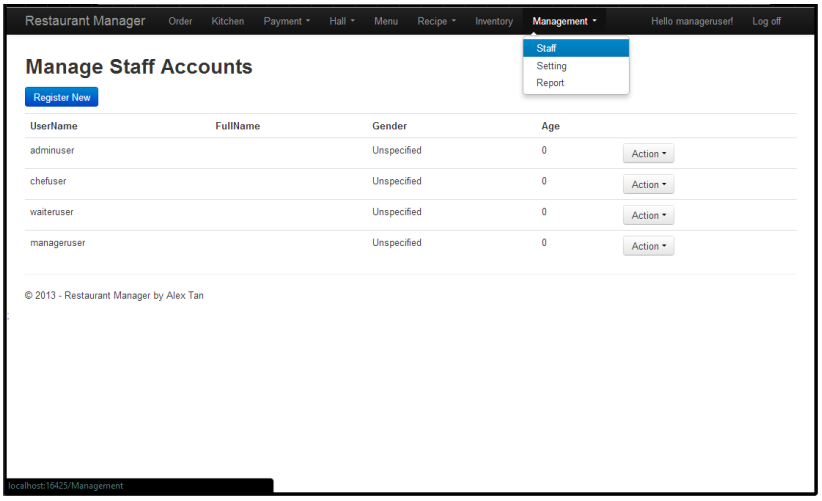



Figure O.34: A screenshot of managing staff view.

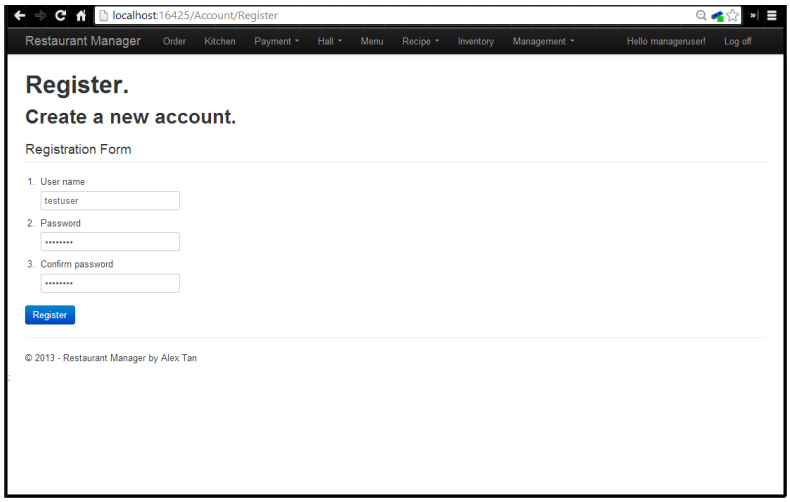



Figure O.35: A screenshot of registering new account for staff.

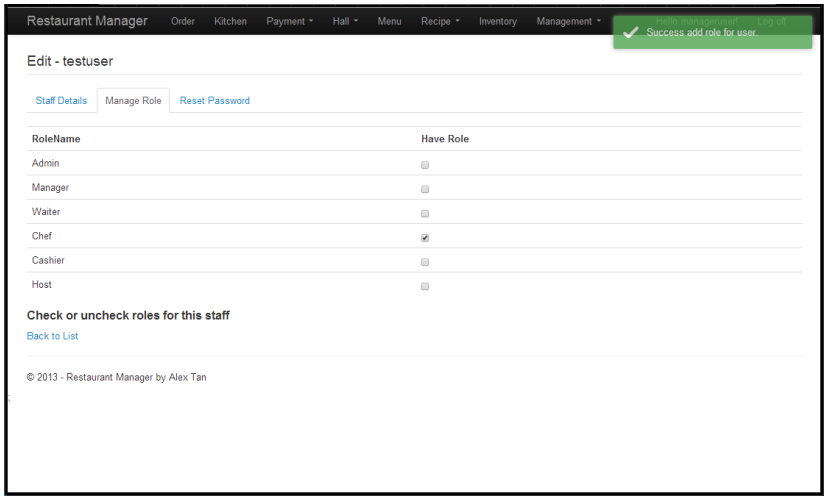



Figure O.36: A screenshot of assigning roles to staff.

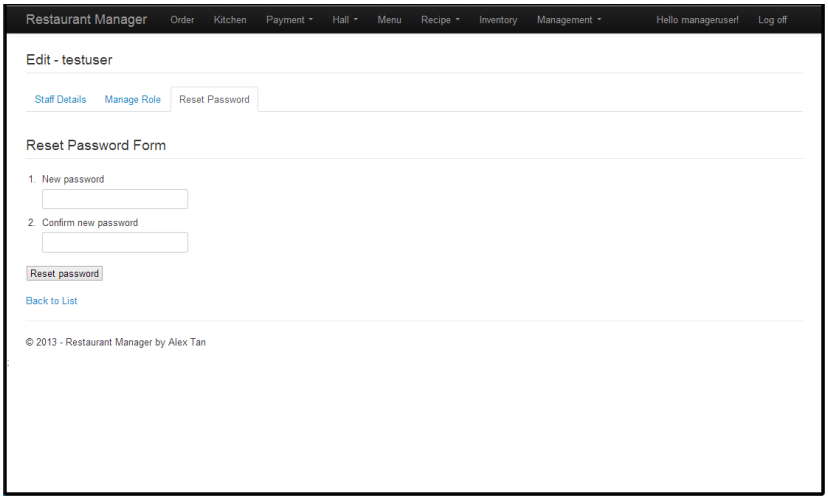



Figure O.37: A screenshot of reset password view for staff.

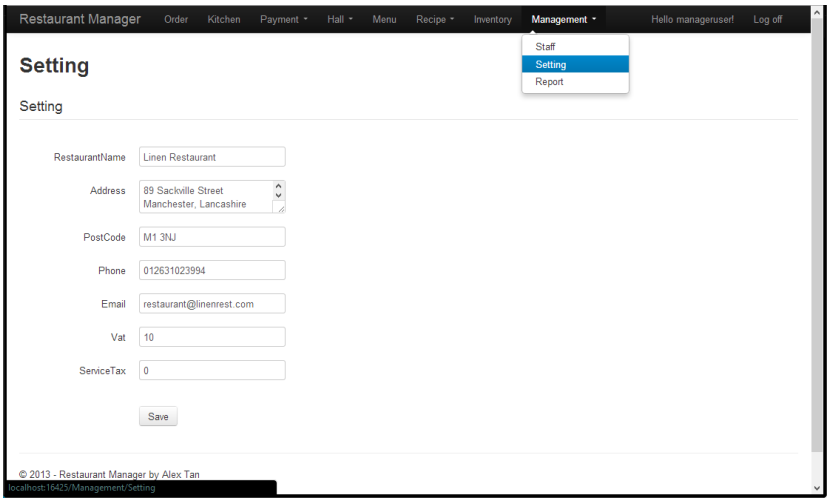



Figure O.38: A screenshot of application setting view.

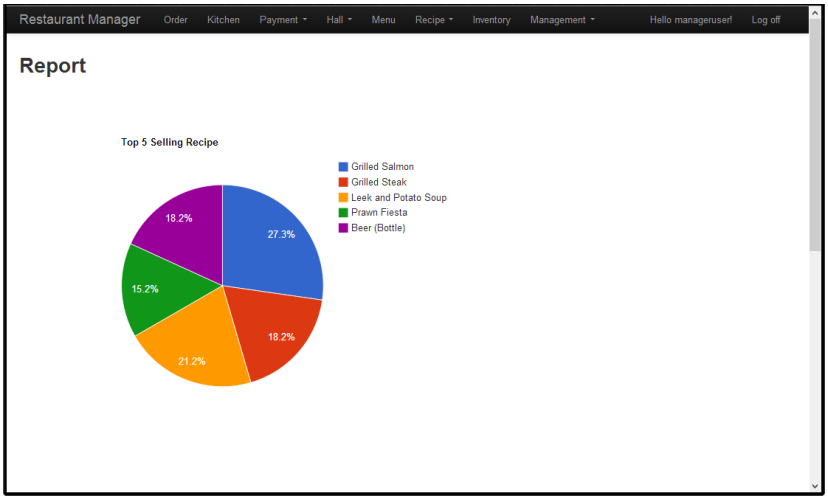



Figure O.39: A screenshot of top selling recipe report.


Figure O.40: A screenshot of sale and cost report.