Restaurant Management System

Third Year Project Report

BSc. Computer Science and Mathematics

School of Computer Science

University of Manchester

Author:
Carl Abernethy

Supervisor:
Prof. Chris Taylor

5th May 2010

Restaurant Management System

Carl Abernethy

Supervised by Prof. Chris Taylor

5th May 2010

Abstract

This report documents the process of designing, developing and testing a software system to be used in a restaurant; usually given the name restaurant management system. The restaurant management system is there to help communication between all teams within a restaurant by minimising the probability of human errors. This report was written by Carl Abernethy as part of his 3rd year project and was published on May 5, 2010.

Acknowledgements

I would like to thank my project supervisor, Prof. Chris Taylor, for providing an awful amount of guidance and input throughout the writing of this report. In addition, I'd like to thank my family for the support throughout my final year at university, and for checking over my report.

Contents

Chapter 1

Introduction

1.1 Chapter Overview

This chapter gives an introduction to the project by defining the problems encountered by restaurants, the main objectives that the system expects to achieve and a brief introduction to existing solutions.

1.2 The Problem

According to a research article written by Horizons [7], in 2006 within the UK there was just over 26,000 restaurants with 734 million meals served that year. As this restaurant sector was worth £7.61 billion, any restaurant generating a good business reputation could lead to the making of a very successful and profitable business. The problem for many businesses is to ensure that they not only attract new customers but to ensure they maintain their existing clientele. It has been argued many times that an existing customer is worth more to a business than a new customer as the cost to attract a new customer can be up to five times the cost to retain an old customer. An online article by Paul Lemberg [9], discusses the pros and cons of this argument.

Within the restaurant sector, a customer is likely to return to the restaurant in the future if they received an excellent customer service as well as appetising food. However, if they had to wait for an unreasonable amount of time or there was a mistake in the order, it's very unlikely the customer would return.

Therefore a solution to this problem would be to minimise mistakes within the order and bill, and help eradicate delays as well as encouraging team work and communication within the team.

The next section will go into the objectives of the proposed solution.

1.3 Project Objective

The objective of this project is to build an electronic restaurant management system using all of the skills and techniques from the field ensuring that no common development mistakes are reproduced. Project management is critical to all software engineering projects and keeping to a project plan will be of similar importance.

One of the main objectives of any business is to maximize profit by increasing efficiency and decreasing overheads[1] without compromising customer satisfaction. Currently, many restaurants use a paper-based system to communicate between the restaurant and kitchen which can be shown to be one

[1] Ongoing expense of operating a business also known as operating costs.

of the least efficient approaches. Even though this approach is implemented in successful profitable restaurants, there are several problems which could be seen as reducing the restaurant's efficiency:

- Miscommunication caused by handwriting.
- Unmanageable order logging.
- Inefficient restaurant-kitchen communication.
- Difficult order tracking and time management.
- Difficult stock management.
- Limited statistical output.

By introducing an electronic restaurant management system these problems can be avoided or improved leading to an increase in profits.

The initial project plan drafted at the beginning of the project can be found in Appendix A.

1.4 Existing Solutions

There are many computerised restaurant management systems available but for each system there exist disadvantages or missing features. The most common type of restaurant management system contains a static order entry computer system usually in the shape of a desktop computer with a touch screen. Typically this common approach is adequate to the restaurants requirements but still requires handwritten orders to be relayed to the order entry computer system. A table comparing features of existing solutions will be presented in Section 2.3.

A slightly different approach was implemented in a restaurant in Nuremberg, Germany, named s Baggers [16]. The restaurant utilises a roller coaster approach to serving the food and an order entry system fully operated by the customer. As reviewed by the BBC [15], there is no need for any waiters as the customers use touch-screen monitors to browse the menu. This new invention can save on operating costs, but the initial injection of cash required is substantial as every table requires the necessary hardware.

The next section will introduce the project proposal listing the proposed features of the system.

1.5 Project Proposal

The aim of this project is to create a restaurant management system that can incorporate the benefits of all the existing solutions but without any of the drawbacks as well as including many new features. A list of proposed features can be found in table 1.1.

Many of the existing solutions to POS (Point-of-Sale) systems are sold with the required expensive hardware so for any business looking to work to a budget, the more enriched software solutions are just out of their range.

[2] An order that has been taken but not yet paid for.

Table 1.1: A table showing the proposed features of the system and the motivation behind the features.

Feature	Motivation
Automated stock control.	Real-time view of ingredient stock levels so only the meals with enough ingredient stock can be sold.
Meal option and preference selection.	Flexible meal options available for the customer.
Wireless order system.	Waiters no longer required to walk to and from the central order computer system.
Advanced discount function.	Calculating the best price for the customer.
Order alerts.	Kitchen and bar staff in direct communication with waiters allowing the kitchen to notify the waiter that service is required.
Flexible GUI design.	Software capable of being used on any sized screen and so must have a flexible design.
Order logging.	All orders logged for future query generation.
Large kitchen order display.	Easy tracking and viewing of all active[2] orders.

1.6 Summary of Chapters

The rest of this report consists of the following chapters:

- Background: Background investigation into the problem.
- Requirement Analysis: Requirements of the system including stakeholder identification, list of features and tabulated requirements.
- Design: Project design process using several diagrammatic techniques.
- Implementation: Discusses the implementation of the software with the help of diagrams and pseudocode.
- Results: Illustrates the system using screenshots.
- Testing: Documents how the system was tested.
- Conclusion: Project conclusion with future development ideas.

1.7 Commonly Used Words

Throughout this report, there are many commonly used words as detailed in table 1.2.

Table 1.2: A table showing the commonly used words throughout this report.

Word	Definition
Ingredient	Ingredient of a meal.
Prepared ingredient	Collection of ingredients.
Item	A meal or drink.
Menu section	A section of menu for example starters, meats, puddings etc.
Suborder	A collection of customer ordered items.
Order	A set of suborders.

1.8 Closing Remarks

This chapter has introduced the foundations of the project. The next chapter gives some background investigation into the problem.

Chapter 2

Background

2.1 Chapter Overview

This chapter gives an insight to Point of Sale (POS) systems similar in nature to that of the one being developed in this project. It also gives a brief introduction to the importance of requirement gathering, a discussion on the development methodologies available as well as a justification on the platform and software used in this project.

2.2 Point-of-Sale (POS) Systems

According to A. Nutt [12], POS systems first dated back to the 1870s, when James Ritty became frustrated with dishonest employees stealing money from the customers in a saloon in Dayton, Ohio. With the inspiration from a counter on a ship's propeller that counted the number of revolutions, he and his brother in 1879 developed the first cash register called 'Ritty's Incorruptible Cashier'.

By the 1970s, the first computerised cash register was developed that was basically a mainframe but packaged as a store controller that had the ability to control the registers. Then in the 1980's, cash registers began to be PC operated which meant that many of the basic PC functions were now available.

Today, the POS systems are much faster, safer and reliable and with the introduction of credit cards and direct communication to the credit card company, transactions are almost instant.

2.3 Existing Point-of-Sale (POS) Systems

Table 2.1: Comparison of existing POS system solutions.

POS System	Table Service	Stock Control	General Reporting	Advanced Discounts	Kitchen Display	PDA Order Input
Point of Success [13] (Standard)	✓	✗	✓	✗	✗	✗
RPS (HOSPOS) [18]	✗	✓	✓	✗	✓	✗
Abacre POS [14]	✓	✗	✓	✓	✓	✗
eZee Burrp [2]	✓	✓	✓	✗	✓	✓

The proposed features in table 1.1 were generated from research into numerous POS systems. Table 2.1 shows the comparison between some of the proposed features and the features of other POS systems. All these POS systems were found on the first page of a Google search on the 14th October 2009.

2.4 Platform Choice

Choosing a suitable platform normally goes down to the programmers experience and the type of software to be developed. The restaurant management system could be developed as a web application or a standalone application but must also be widely supported and platform-independent. Therefore as the developer has minimal or no experience in web programming, the decision was taken to develop a standalone application.

The next decision was to decide on a programming language, with the developer having previous experience in C, C++ and Java. This decision was fairly easy and Java was the selected programming language as the developer has great knowledge in the Java Database Connectivity (JDBC) API[1] that allows database-independent connectivity between the Java programming language and numerous databases [8].

2.5 Software Choice

In software development, the use of integrated development environments (IDE) can increase the efficiency of a programmer. An IDE is a software application that consists of a source code editor, compiler and debugging tools with its main aim to assist the programmer. Simple notepads are not strictly IDEs but can do the same job with the assistance of a compiler.

The two most popular IDEs available for Java programming are Netbeans and Eclipse as they are free, support multiple platforms and offer many features including integrated version control and debugging tools. The two main problems with IDE's are that due to the wealth of features available and plug-in support, there is an associated cost, as their performance is poor and in particular they require more memory and processing power than a standard text editor.

For this project Netbeans was the chosen IDE, as the developer has more experience and knowledge of the Netbeans graphical user interface.

2.6 UML

Diagrammatic techniques are used to visualise, construct and document software systems under development. The most general modelling language to describe both the structure and behaviour of a software system is Unified Modelling language (UML) created by the Object management group. The diagrams one can create using UML can be shown by a class diagram (Figure 2.1). Throughout this report, numerous models defined within the UML class diagram (Figure 2.1) will be used to graphically represent the system.

2.7 Requirement Gathering

Requirement gathering is a very important step in software engineering. According to an article written by Craig Murphy [11], Boehm's cost model discusses how discovering errors at an early stage of software development can reduce overall costs. Figure 2.2 is an example of Boehm's cost model and uses the waterfall technique to give a visual insight into how important gathering accurate requirements can be

[1] Application programming interface.

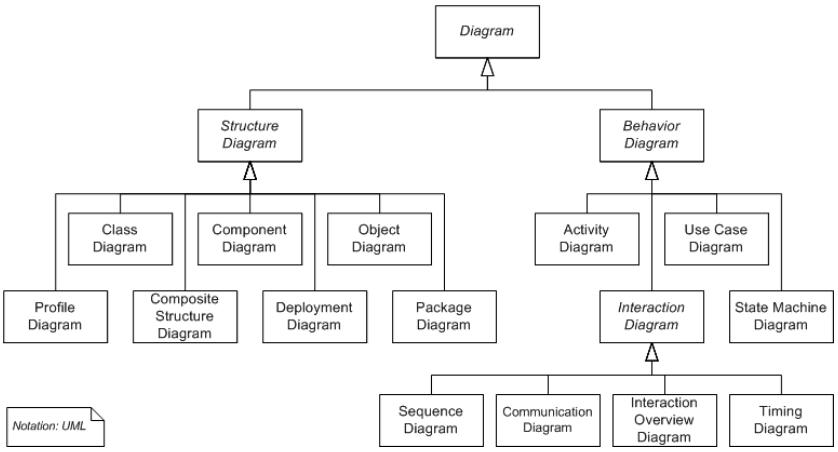

Figure 2.1: UML class model.

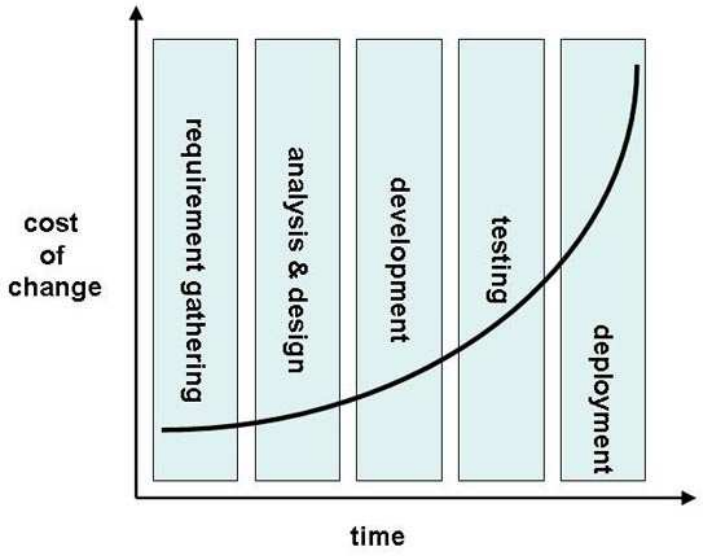

Figure 2.2: Boehm's cost model [11].

when the project has a strict deadline to work to. Roughly a £1 cost of change in the requirement gathering can cost up to £100 in the testing stage and £1000 in the deployment stage to fix.

2.8 Development Methodology

A software development methodology is a framework used to plan the design of a software system controlling the process of development. According to Geoffrey Elliott [5], software development methodologies first emerged in the 1960's with systems development life cycles (SDLC) being considered the first formalised methodology. Since then there have been numerous popular software development approaches including the waterfall model, prototyping, incremental, spiral and agile.

The agile methodology refers to a collection of different agile methods. This project will be based on Extreme programming (XP) which is one of these agile methodologies using an iterative based framework. Each iteration has a development cycle very similar to the waterfall model consisting of planning, requirements analysis, design, development and testing. At each iteration, the business representative sometimes referred to as the 'customer' is given a demonstration to promote useful feedback. This type of methodology reduces risk and lets the project adapt to requested changes quickly with minimum cost.

According to the Agile Manifesto [10], some of the main principles behind the agile methodology are:

1. Customer satisfaction by rapid, continuous delivery of useful software.
2. Working software that is delivered frequently.
3. A welcomed late change in requirements.
4. Simplicity.
5. Regular adaptation to changing circumstances.

Given these principles and the nature of the project, extreme programming was the best choice methodology to use as we expected frequent customer communication and constant requirement changes post customer feedback. Therefore for this project, the supervisor acted as the customer regularly reviewing the work. Any errors or unthoughtful designs were then picked up and fixed relatively early in the process helping to minimise costly errors as shown in Boehm's cost model (Figure 2.2).

2.9 Chapter Summary

This chapter has given examples of POS systems that are directly related to this project as well as general information about requirement gathering and development methodologies. The next chapter starts the process of the development methodology by generating the requirements of the system.

Chapter 3

Requirement Analysis

3.1 Chapter Overview

This chapter will look at the stakeholders of the system and then discuss the required features using a use case diagram to illustrate.

3.2 Stakeholder Identification

As defined in the Business Dictionary [4],

> a stakeholder could be a person, group, organisation that has direct or indirect stake in an organization because it can affect or be affected by the organization's actions, objectives and policies.

Hence, stakeholders can be split into two groups; internal and external with each stakeholder contributing directly or indirectly towards the business activities.

As an example, any system were there exists customer communication, that customer will be a non-financial beneficiary stakeholder. According to an article written by Ian Alexander [1], the person, group, organisation, or system is a stakeholder if they can be defined by any one of the following four questions:

> Who needs to be consulted on the scope of this project?
> Who has an input to the budget of this project?
> Who can support or harm this project politically?
> Who can provide guidance on the usability, functionality, and required qualities (reliability, safety, lifetime, maintainability, ...) of the system under development?

Therefore using these 4 questions as a guide, we can generate a list (Table 3.1) of the stakeholders in this project.

3.3 Use Cases

A use case diagram that is part of the Unified Modelling Language (UML) which was introduced in Section 2.6 gives a graphical overview of the functionality of a system. A use case diagram consists of actors that are normally the stakeholders of the system and their use cases commonly defined as goals.

Table 3.1: A table showing the stakeholders of the project.

Stakeholder	Role
Carl Abernethy	Project developer.
Prof. Chris Taylor	Project supervisor.
Management	User of the management application.
Waiters	User of the restaurant application.
Kitchen Staff	User of the kitchen application.
Bar Staff	User of the restaurant application.
Customer	Indirect user of the system.
Cashier	External stakeholder; accepts payment.

Figure 3.1 shows several use case scenarios of the system that convey how the stakeholders interact with the features to achieve a business requirement. The use case scenarios from figure 3.1 can be found in Appendix B.

The use case diagram is designed to be sequential so by following the use cases down the spine, one can follow the major steps of an order and several post features to query the data.

3.4 Features

An important part of requirements gathering is the production of a list of system features that categorises on priority.

Table 3.2: A table showing the proposed system features and allocated priorities.

Feature	Priority
Communication of data between each application.	1
Minimum click touch screen GUI design for efficient ordering.	1
Meal ingredient and cooking preference options.	1
Interface to view active orders in the kitchen.	1
Ability to add flexible discounts; calculating best price for the customer.	1
Interface to maintain and manage the menus and associated meals.	1
Stock control for all ingredients; reducing/increasing stock automatically.	1
Ability to define groups of ingredients that may be used in numerous meals.	2
Flexible meal grid design to fit any screen size.	2
Real time waiter status alerts.	2
User login functionality.	3
Interface for table management and selection.	3
Figure generation; management can view statistics in numerous forms.	3
Automatic daily stock level alerts.	3
Ability to define meals by images as well as text.	3

3.5 Measureable Goals and Requirements

The measurable goals and requirements of the system are a list of manageable[1] requirements and goals for each application that can be prioritised and 'ticked off'. The software requirements specification

[1]Requirements that are realistic and can be completed within the allocated time.

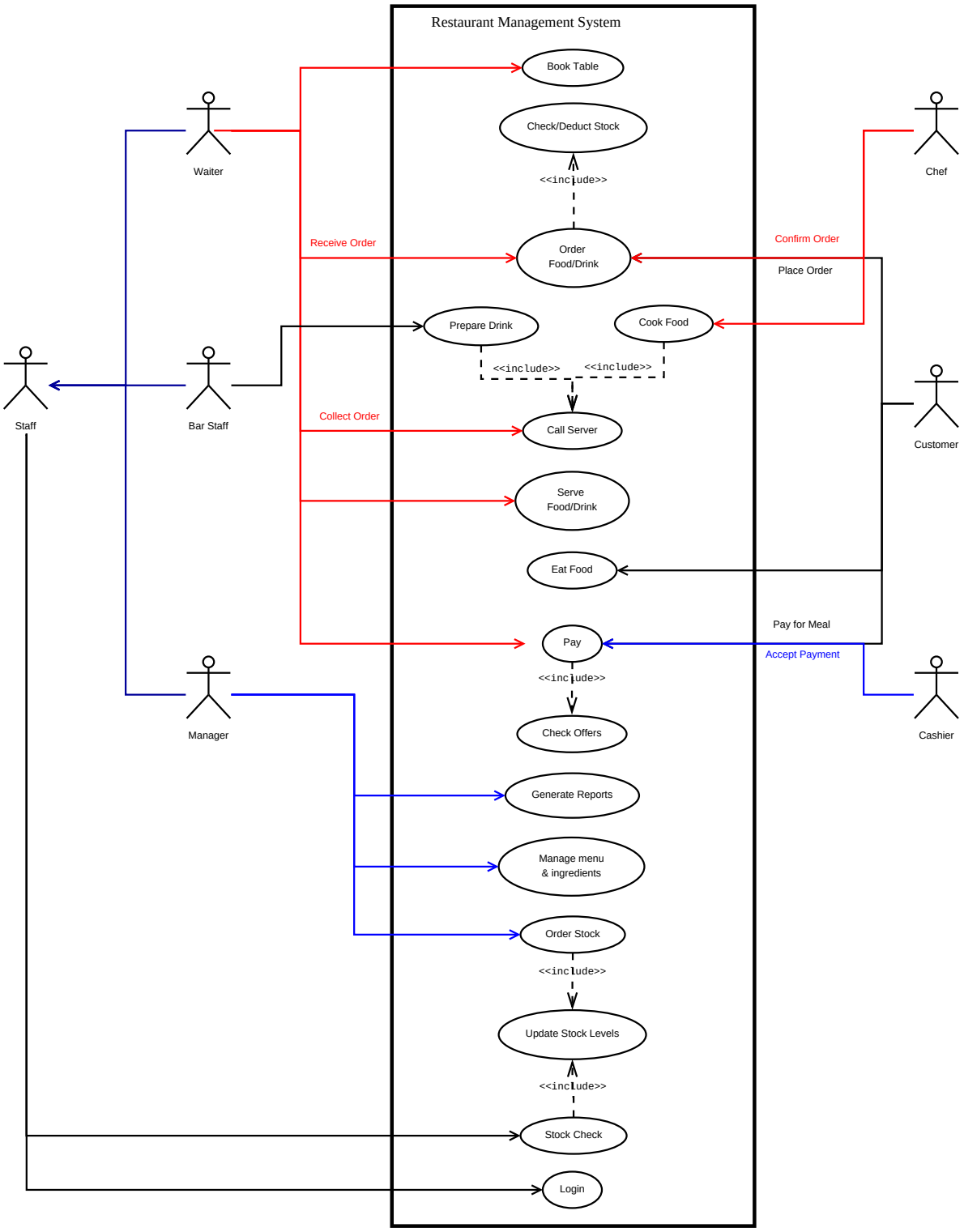

Figure 3.1: Use case diagram showing some of the major features within the restaurant management system.

(SRS) is a complete list of requirements to be designed and developed and can take the form of functional or non-functional requirements.

3.5.1 Functional Requirements

A functional requirement is a requirement that, when satisfied, will allow the user to perform some kind of function. The functional requirements of the kitchen, management and order application can be found in tables 3.3, 3.4 and 3.5 respectively. The priority value is in the range from 1 to 3 where 1 is high priority and 3 is low priority.

Table 3.3: Kitchen application functional requirements.

Requirements	Priority
Organised display of active orders.	1
View preferences and optional choices of every meal.	1
Inform waiter; update order to 'in progress'.	2
Inform waiter; update order to 'ready'.	2
Display elapsed time and progress[2] of each order.	3

Table 3.4: Management application functional requirements.

Requirements	Priority
Ability to display the real time stock levels.	1
Ability to add ingredients, meals and menus.	1
Ability to edit ingredients, meals and menus.	1
Ability to remove ingredients, meals and menus.	1
Ability to display statistics of complete orders.	3
Ability to add new waiters to the system.	3
Ability to add, edit and remove offers.	1
Ability to add time intervals[3] to menus and offers.	1
Ability to alter the grid size of the meals, menu sections and menus.	1
Ability to alter the seating layout.	3
Ability to alter the price of ingredients.	1
(Note: Ingredient might still be in stock at a different price)	
Ability to change transparency of an image that represents a meal.	3
Means of stock checking.	1
Means of updating the minimum stock level.	1

[2]How many items within the order have been cooked.
[3]Duration of time defined by a beginning and end.

Table 3.5: Order application functional requirements.

Requirements	Priority
Ability to view the active suborder details.	2
Ability to add a new order to a table.	1
Ability to add a new order without defining a table.	1
Ability to add a suborder to an existing order.	1
Ability to delete a suborder that has not yet been confirmed within the order system.	1
Ability to view optional ingredients in a meal.	1
Ability to view cooking preferences of ingredients that can be cooked different ways.	1
Ability to only view the active[4] menus.	1
Ability to view the status of all active suborders.	2
Ability to print customer receipts on order completion.	3
Ability to view transaction list of current order.	3
Ability to alert the waiter when the drinks are complete.	3
Ability to delete an item or clear the transaction list.	2

3.5.2 Non-Functional Requirements

Non-functional requirements are usually some form of constraint or restriction that must be considered when designing the solution. The non-functional requirements of the kitchen, management and order application can be found in tables 3.6, 3.7 and 3.8 respectively. The priority value is in the range from 1 to 3 where 1 is high priority and 3 is low priority.

Table 3.6: Kitchen application non-functional requirements.

Requirements	Priority
Ability to interact with the MySQL database.	1
Means of refreshing the orders within the ordered list.	1
Means of refreshing the status of orders.	3
(How many items in the suborder have been cooked)	
Means of refreshing the status of orders.	3
(How many items in the suborder have not been cooked)	
Means of refreshing the menu section list.	2
(How many items in the menu section have been cooked)	
Means of refreshing the menu section list.	2
(How many items in the menu section have not been cooked)	
Means of refreshing the meal colours within the ordered list:	3
==> White: not started	
==> Yellow: in progress	
==> Green: complete	
Means of displaying only the food items so drink items are omitted.	1
Means of clearing the suborder from the kitchen display.	1

[4] A menu is active when the current time is within the specified time interval of the menu.

Table 3.7: Management application non-functional requirements.

Requirements	Priority
Ability to interact with the MySQL database.	1
Means of refreshing menus, menu sections and meals live.	3
Only accessible by management staff.	3
Means of displaying the links therefore being able to view the ingredients in a meal etc.	1
Means of searching the database using wild cards for easier data input.	2

Table 3.8: Order application non-functional requirements.

Requirements	Priority
Ability to interact with the MySQL database.	1
Ability to update the suborder status every 5 seconds.	1
Ability to run on a PDA by defining smaller grid and font sizes.	2
Ability to use the application using a touch screen without the need of a keyboard or mouse.	1
Colour co-ordinated meal buttons to visually warn the user of low stock.	2
Ability to disable meals that have run out of compulsory stock items.	1
Ability to only show an ingredient as an optional ingredient choice if there exists enough ingredient stock.	1
Ability to calculate stock ingredient price, order price and discounted price.	1
Live refreshing of the menus depending on the starting time of the order.	1
Ability to apply offers depending on the starting time of order even if the offer is inactive on completion of the order.	1
Minimum clicks from the beginning to the end of an order.	1

3.6 Chapter Summary

This chapter has discussed the requirements of the system and the development methodology that will be used throughout this project. The design of the system is documented in the next chapter.

Chapter 4

Design

4.1 Chapter Overview

This chapter will focus on the design of the system using diagrams to illustrate graphically certain sections of the software system.

4.2 Introduction

This project has been designed using numerous diagrammatic techniques. Recall from Section 2.6, that the most general modelling language to describe both the structure and behaviour of a software system is Unified Modelling language (UML).

Use case diagrams have already been used in the requirements analysis as a way to graphically overview the order process within the system. Other diagrams from the UML family are used in the design stage to show the structure and behaviour of numerous sophisticated design features.

4.3 Component Diagram

A component diagram is part of UML and its main purpose is to show the structural relationship between components in the system. A component diagram is useful for this system as it shows the higher architecture, as shown in figure 4.1.

4.4 Data Storage

The restaurant management system will be built around the data storage technique therefore choosing the most appropriate persistent[1] data storage is critical to a successful project and we can assume a flat file storage approach is inadequate. The two most popular types of persistent data storage available are relational database management system (RDBMS) and extensible markup language (XML).

4.4.1 Relational Database Management System (RDBMS)

A relational database management system (RDBMS) is a database managed system based around a relational model and are the corner stone's to many software systems including web based systems.

[1] Data structure that preserves the original data when changed.

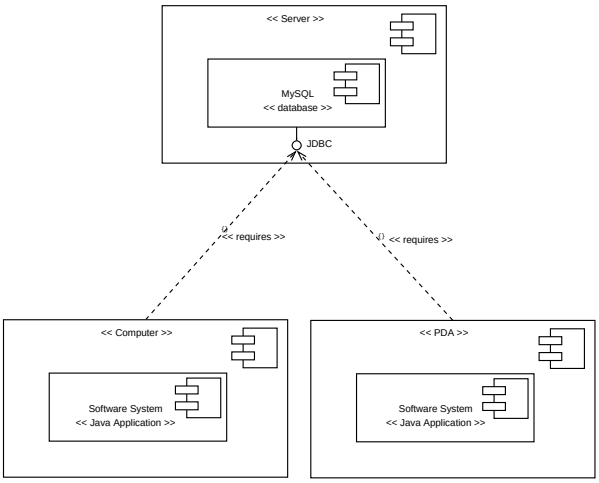

Figure 4.1: Component diagram showing the higher level architecture of the system.

RDBMS are one of the most popular data storage methods out in the market and offer many advantages including:

- Fast data extraction using structured query language (SQL).
- Good management of data and security through the management system.
- Good level of data consistency.
- Advanced features including functions and triggers.
- Requirement of a data model to be developed; leading to long term cost effectiveness.

In industry, there are numerous expensive highly functional RDMBSs including Oracle and SQLServer that are very popular and offer technical support. However, there are also numerous open-source solutions with many adjudged to be as good or better and are becoming even more popular with small scale software systems.

4.4.2 Extensible Markup Language (XML)

XML is a markup language that was designed to transport and store data and is another example of a persistent data storage technique. However, it is not a predefined language thus all tags must be defined and due to its hierarchically data structure all elements must be promoted or demoted.

XML could be used in two different ways in data storage; storing the XML documents within a database or having the XML documents as the fundamental unit of storage. In both cases the XML can be queried using either XPATH or XQUERY which are query languages for extracting data from XML documents.

4.4.3 Storage Method Chosen

The main difference between XML and RDBMS is that XML is hierarchical and RDBMS is relational. As restaurant data can be best represented in a hierarchical way one would believe that XML would be the best approach but it's not always that straight forward. SQL is an extremely flexible and robust

querying language and for the queries required and the type of software system begin designed, it was concluded that RDBMS would be the best storage method.

The next choice was to decide on the type of RDBMS to use. As discussed there are many open source RDBMSs available for us to choose from and for the main reason of experience, MySQL was the preferred option. Figure 4.2 shows just how competitive the performances of different RDBMSs are.

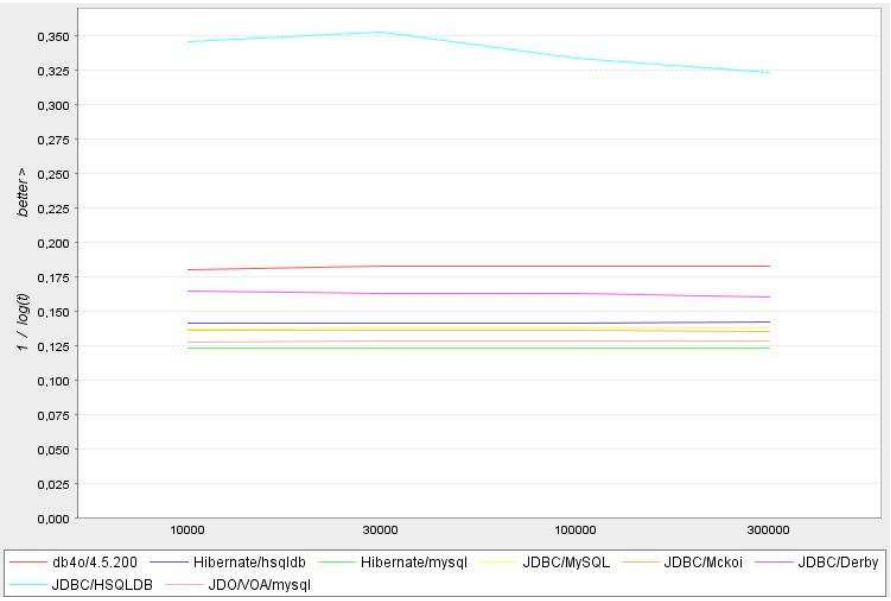

Figure 4.2: Database comparison diagram [3].

4.4.4 Normalisation

Normalisation comes in many forms ranging from first normal form to sixth normal form. The normalisation of a database is a systematic way to free the database of undesirable characteristics where inserts, updates and deletions of data could lead to the loss of data integrity. The greater the normal form, the greater the data integrity of the database.

The database in this system was designed to be in Boyce-Codd normal form which is a slightly stronger version of the third normal form. For the database to be in Boyce-Codd normal form, it had to pass for all previous normal forms as well as Boyce-Codd normal form.

A well designed database will normally abide by the first, second and third normal forms as they are the basics to a well structured relational database. According to Horsforth School [17], the first three normal forms can be defined as:

1. **First norm**: Every attribute is atomic or single valued therefore there are no repeating fields.
2. **Second norm**: All attributes not part of the primary key must be dependent on the full key.
3. **Third norm**: There must be no transitive determinants, or each attribute that is not part of the key must be determined only by the key.

Finally for the database to be in the desired Boyce-Codd normal form, all tables must abide by the first, second and third normal forms and must not have any determinants that are not candidate keys for the table.

4.4.5 Entity Relationship Diagram

An entity relationship (ER) diagram is a modelling language used to represent a type of semantic data model of a system. The ER diagrams are often used to represent a relational database and its requirements in a top-down fashion usually defined as the database schema. The database schema for this database has been split into two ER diagrams (Figures 4.3 and 4.4).

Figure 4.3 graphically shows the objects and their relationships that are contained within a meal. The meal object will be made of at least two ingredients that can be either a normal ingredient or a prepared ingredient. Note, a prepared ingredient is a collection of ingredients used to either group commonly used ingredients or to group optional ingredients. Each ingredient will have a default and manual measurement with the default measurement entered on input of the ingredient and the manual measurement entered if the meal ingredient link requires a different amount.

Also, each ingredient will be part of a generic ingredient object as there are many ingredients that are the same item but packaged in a different way at a different price. This allows the database to be in Boyce-Codd normalised state. An example of this would be the drink Coca-Cola which can be bought by bottle, can or draught, thus are the same item but packaged differently at a different price and amount. Finally each ingredient and prepared ingredient can be part of a category allowing optional ingredients to be interchanged with other ingredients in the same category.

Figure 4.4 graphically shows the relationships for the menu, order and offer objects. The menu consists of a date time relationship that provides the intervals to when the menu is active and a menu section relationship that contains the colour variables and items under that particular menu section.

The order consists of one to many suborders with the suborder consisting of one to many items. The order stores all the ingredients within each item and also the replaced ingredient if that optional ingredient was replaced.

The offer consists of a date time relationship that provides the intervals to when the offer is active and a offer section relationship that contains the sets required by the offer.

4.4.6 Database Design Concepts

The database was the backbone to some highly important functional requirements, therefore the schema (Figures 4.3 and 4.4) needed to have the structure to deal with them.

Some important design concepts of the database are:

- Ability to store prepared ingredients to reduce the size of the meal ingredient list. Recall, that a prepared ingredient is a collection of ingredients.
- Ability to allow numerous options for optional ingredients within meals so that every ingredient (or prepared ingredient) is part of a category. If an ingredient is optional, then that ingredient should be able to be removed or swapped with any other ingredient in the same category.
- Ability to control the stock levels by allocating a variable to all prepared ingredients and meals with the variable reacting in real time to the status of the items ingredient stock level.
- Ability to cope with new supplies where the price differs to the current price within the database.

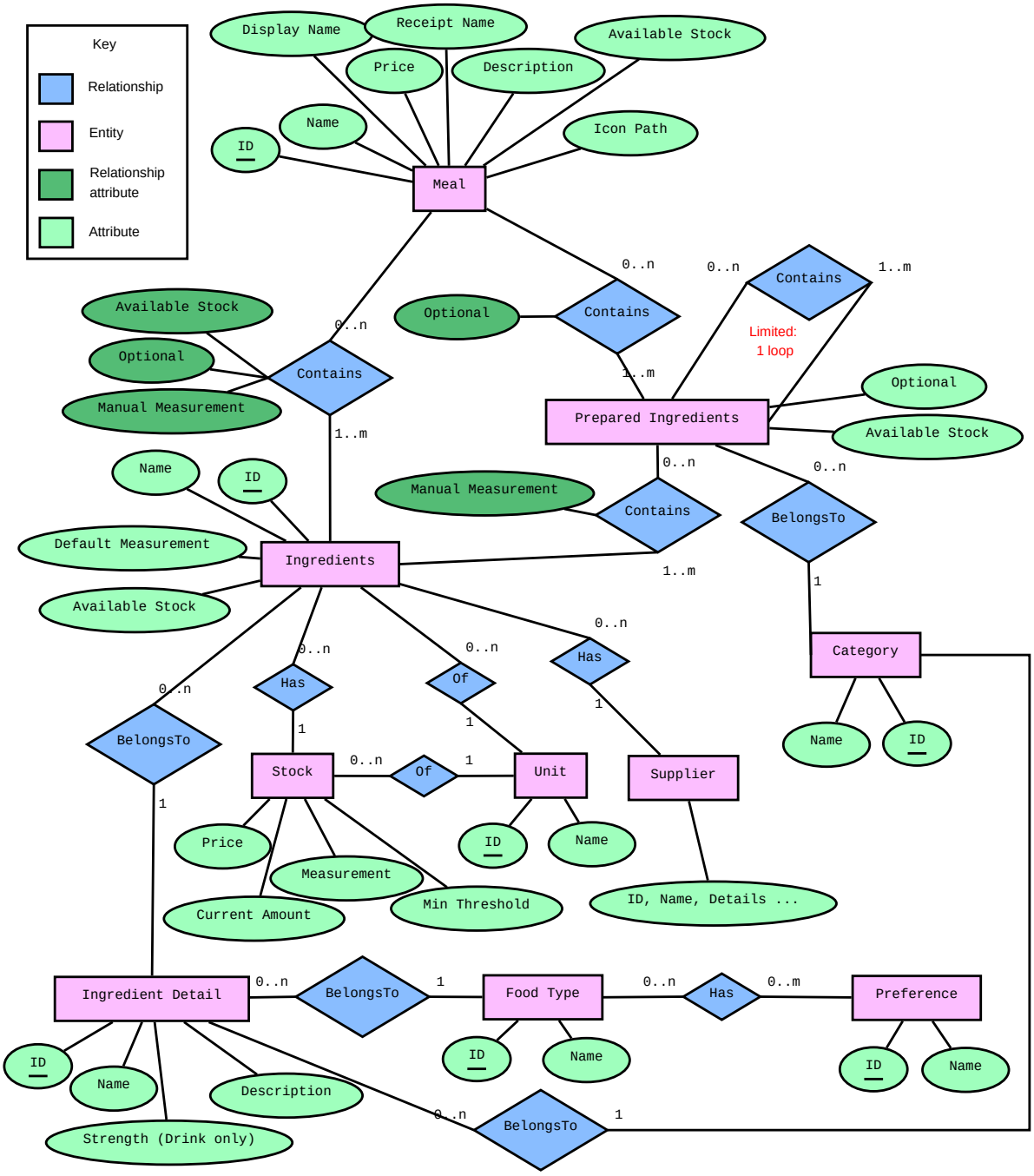

Figure 4.3: Entity Relationship diagram of a meal.

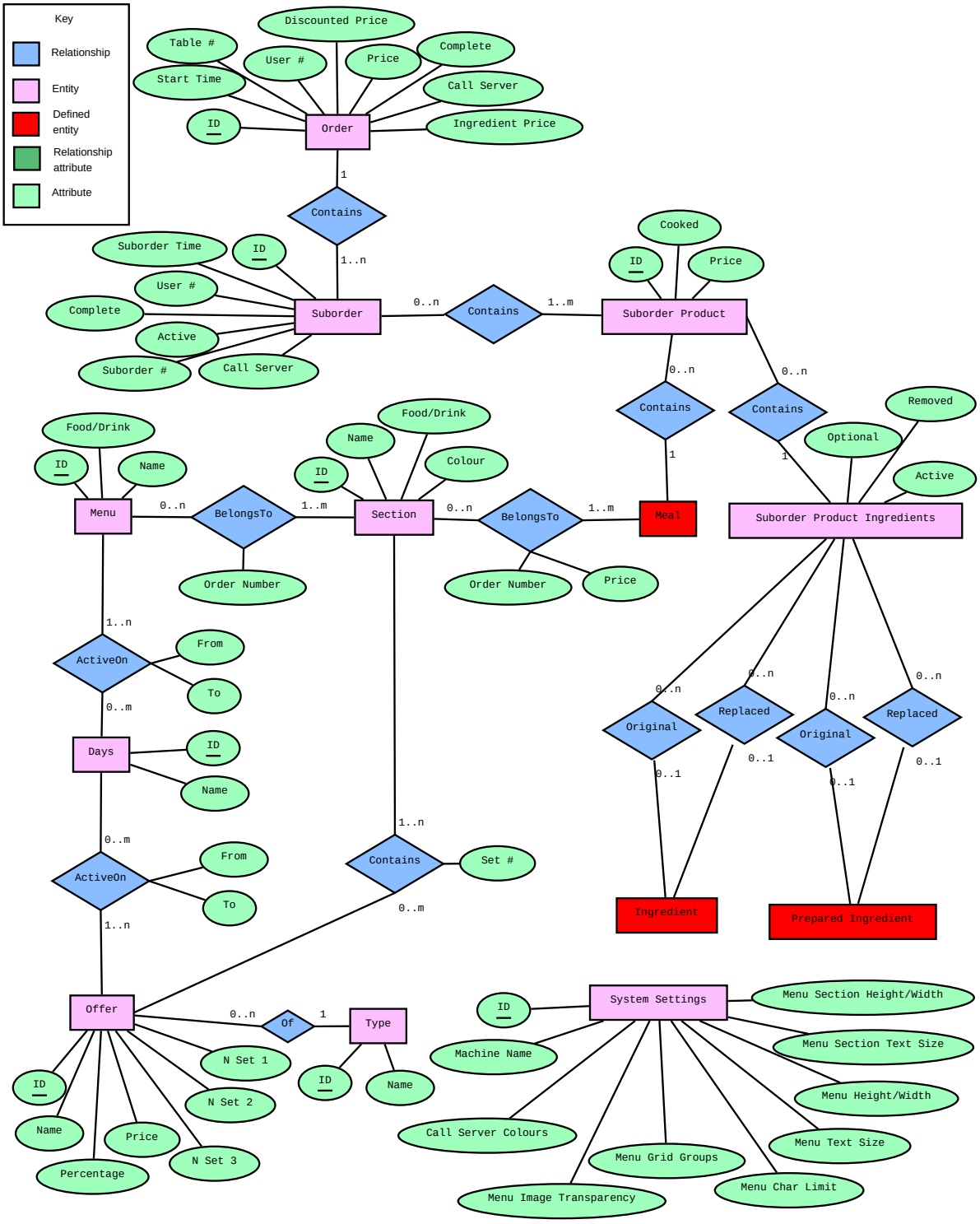

Figure 4.4: Entity Relationship of the menu, order, offer and system settings.

4.5 Graphical User Interface

The graphical user interface (GUI) is the only component of the system that the user interacts with therefore is of great importance. The design had to be simple, clear and concise but whilst also showing all of the required features. The main objective was to create a GUI that allowed the user to get the order completed in the minimum time possible. This was judged by the time taken to complete the order as well as the amount of clicks required to get from the start to the finish. In total, there are 3 different GUIs within the restaurant management system with each GUI requiring a different design specification.

4.5.1 Order GUI

The order system GUI has the most user interaction through the means of a touch screen. Hence usability and user-friendliness of the GUI was of the highest priority. Therefore the specification was as follows:

- Chronological order of steps; either from left to right, top to bottom or a mixture of both.
- Minimum clicks from start to finish.
- Meal items with optional images.
- Ability to fit on any monitor size.
- Maximum space usage with readable font.
- Colour coordinated meal buttons to display the stock status.

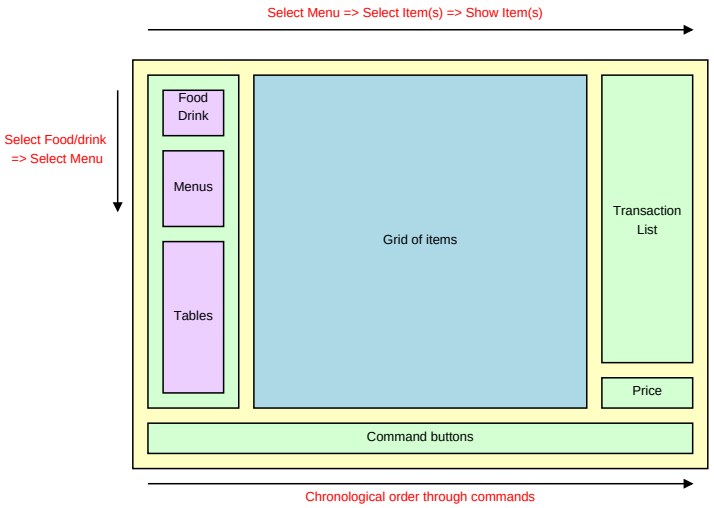

Figure 4.5: Design of the order GUI for a monitor.

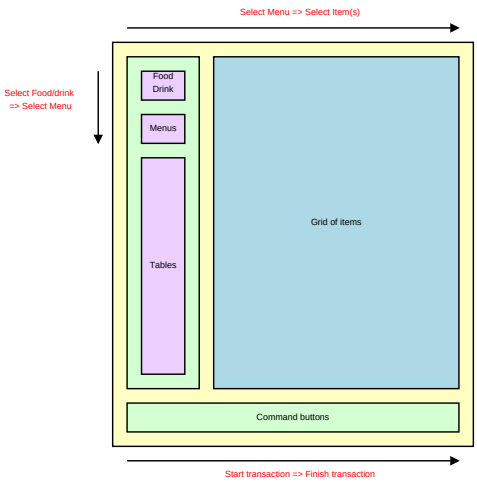

Figure 4.6: Design of the order GUI for a PDA.

Figures 4.5 shows a design of the order GUI that abides by the specification. The design keeps the sequence of steps in a chronological order to minimise hand movement from step to step. By following the design top to bottom and left to right, the order starts from the top left corner and eventually finishes at the bottom right corner. This not only helps keep the GUI structured, but allows the user to use their initiative if unfamiliar hence decreasing the learning curve.

The order process is controlled by the command buttons at the bottom. These allow the user to create an order, add a suborder to an existing order and complete an order. The three purple grids down the left hand side contains the different menus and current active suborders. The blue grid in the centre contains the different sections of the selected menu and the associated meals and drinks within the sections. The 'Transaction List' and 'Price' boxes show the current items within the suborder and the current price respectively.

The shape of the design in Figure 4.5 accommodates all normal size monitors with the grid size for the food/drink, menus, tables and items all customisable. Hence with little or no change the basic components in the order GUI can be designed to fit a PDA screen with a resolution of around 480 by 320 as shown in Figure 4.6. In the designs, Figure 4.5 would use, as default, the grid sizes 8x8 (items), 2x1 (food/drink), 2x3 (menus) and 3x8 (tables) where as the PDA GUI Figure 4.6 would use, as default, the grid sizes 4x4 (items), 1x2 (food/drink), 1x3 (menus) and 1x8 (tables). Note, these dimension would be the preset default values and would be changeable in the system settings.

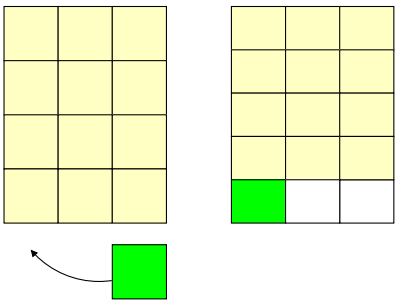

Figure 4.7: GUI grid expansion.

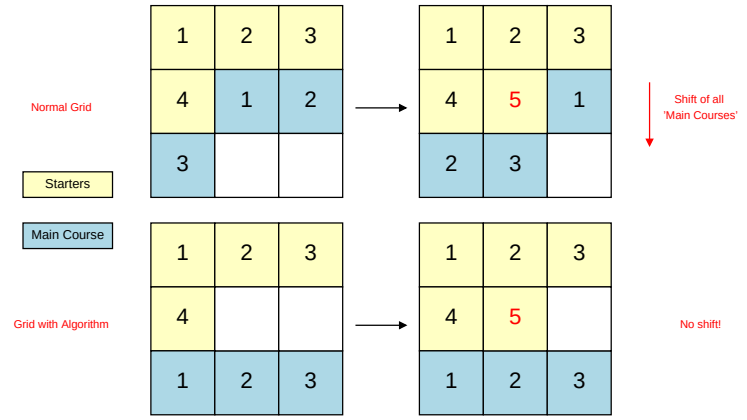

Figure 4.8: GUI grid items algorithm.

As we can never assume the number of items, menus and tables active at any one point, all the grids should be able to automatically resize when the grid becomes full adding an extra row to the bottom. Figure 4.7 visually shows how a full grid resizes when an additional item is added.

As well as the general layout, individual components required specific design details incorporated. As new meals and drinks could be added to the menu at any point, then if a normal grid was used, the addition of a new item that wasn't at the end of the grid would shift all items after the newly inserted item by one. At sight this is a problem, as with frequent use of the system, the user becomes accustomed to the exact position of certain functions. Hence a change to the layout could lead to an increase in human errors and a decrease in the user transaction speed.

Figure 4.8 shows a solution that assigns the starting item in every menu section a new line therefore the shifting of items would only occur when a section requires an extra line.

Finally, a well known saying is that a 'picture tells a thousand words', therefore by allowing the system to use transparent images as well as the text, the end user should locate the particular item faster.

4.5.2 Kitchen GUI

The kitchen system requires a GUI to display all active orders. This GUI requires little user input, therefore the design of the GUI and the way the data was displayed was the most important part. The specification of this GUI was as follows:

- View of all sub orders with elapsed time and status.
- Separation of ordered items into their specific menu section.
- Colour coordinated item list.
- Ability to view on numerous sized monitors.

Figure 4.9 shows a design of the kitchen GUI that abides by the specification. The design utilises as much space as possible, providing up to four sections. These sections give the kitchen staff the option to separate the orders down to any menu section, for example deserts, starters, meats etc. The suborder list on the right hand side of the design gives details about the status of active suborders including elapsed time and a count of how many items are ready.

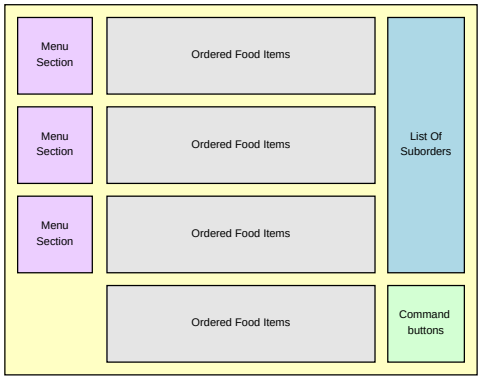

Figure 4.9: Design of the kitchen GUI.

4.5.3 Management GUI

The management system requires a GUI to input data therefore requires an appropriate input design. Figure 4.10 shows the design of the management application with the different functions appearing down the left hand side as a tab. Each tab contains a subform with Figure 4.9 containing the structure of the generic data input form.

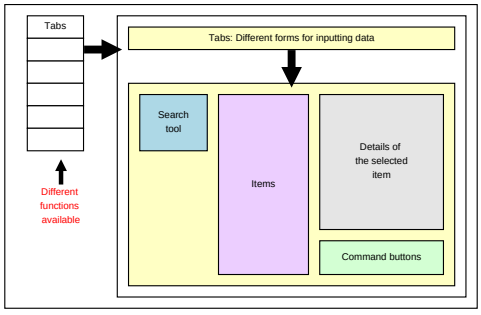

Figure 4.10: Design of the management GUI.

4.6 Pricing Algorithm

An algorithm is a finite set of instructions for solving a particular problem. In our case, an algorithm will be required to generate the best price for the customer given a set of special offers that are available. The system will allow the input of two types of offer:

1. Buy X from A, Y from B and Z from C for £S
2. Buy X from A and get S% of Y from B,

where X,Y and Z are positive integers, A,B and C are sets associated with the offer and S is a positive real number. A few examples of offers that can be expressed using the above generic structure are:

- Buy one get one free.
- Buy three get the cheapest free.
- Buy two get the third half price.
- Buy a starter, a main course and a pudding for £15.00.
- Buy a main course and a starter or pudding for £12.00.

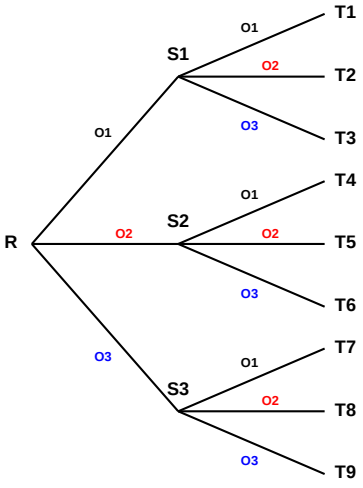

Figure 4.11: Visual example of the price algorithm problem.

There are many types of classifications of algorithms such as exhaustive, divide and conquer and the greedy method but for any algorithm, the performance is always measured by the time and space complexity.

The first step is to model the problem so one can start to rule out different classifications of algorithms. Figure 4.11 shows an exhaustive approach by applying every offer to the starting set of meals within the order with the offers denoted by O# and the sets denoted by R#,S# and T#. This process is repeated until the set is empty or zero offers apply. By calculating the saving at each node, the total cost can be calculated at the end of each path starting at node R and ending at any of the T nodes. The path which saves the most will then be the optimal solution. Note that any meal that is used within an offer is then removed from the set and cannot be used again.

The exhaustive approach always generates the correct result but will not offer the best in time and space complexity. Therefore we need to investigate whether any of the other classifications of algorithms will generate a more efficient algorithm.

The greedy model algorithm follows the strategy of making the logical optimal choice at each stage. Therefore in our case, the greedy model would take the path from node R to the node S with the greatest saving. This is repeated until the set is empty or zero offers apply. The time complexity of the greedy model algorithm would be significantly less than the exhaustive algorithm but does not always produce the optimal solution. Consider the following example:

```
Set_A:  Meal_1 = £3     Offer_1: Buy 3 from Set_A for £4
        Meal_2 = £3     Offer_2: Buy 1 from Set_A get 1 from Set_A free (100%)

Order:  Meal_1  (Order set defined as Set_R)
        Meal_1
        Meal_2
        Meal_2
```

```
    GREEDY APPROACH SOLUTION

Applying Offer_2 to Set_R: Saving £5
Set S1: Meal_1 (Removed)
        Meal_2 (Removed)
        Meal_3 (Removed)
        Meal_4

Unable to apply any offer!

Total saving: £5
```

```
    OPTIMAL SOLUTION

Applying Offer_1 to Set_R: Saving £3
Set_S2: Meal_1 (Removed)
        Meal_1 (Removed)
        Meal_2
        Meal_2

Applying Offer_1 to Set_S2: Saving £3
Set_T1: Meal_1 (Removed)
        Meal_2 (Removed)
        Meal_3 (Removed)
        Meal_4 (Removed)

Unable to apply any offer!

Total saving: £6
```

From this example we notice that the greedy method finds a solution but is not the optimal solution. Therefore we can rule out this classification of algorithm.

Another approach could be to try and redraw the problem into a graph with one source and one sink, then use an optimisation algorithm such as the max-flow min-cut theorem. In order to apply an optimisation theorem we need to develop a graph that either has an edge or vertex weight. In our case, the edge weight would be the saving made when that specific offer is applied to the set. Once again, there is a drawback to this type of algorithm. As an offer is applied the set decreases so to draw the problem as a graph would just be drawing the exhaustive algorithm out, hence this attempt would lead back to the same complexity.

After numerous attempts at other approaches, the only algorithm that guarantees the optimal solution is the exhaustive approach. Therefore we can calculate the best and worst case scenario. The best case scenario is when there are n items in the order, and m amount of offers but zero offers apply to the order. The worst case scenario is when there are n items and m offers and with each offer removing one item from the order each time. Hence in the worst case the time complexity will be $O(n^m)$ which is perfectly workable for likely values of n and m.

4.7 Flow Charts

A flow chart is a diagram used to represent the process flow of an algorithm, problem or some transaction within a business. Therefore a flow chart (Figure 4.12) was used to graphically represent the process flow of an order.

4.8 Chapter Summary

This chapter has displayed many graphical representations of the design of the system. The implementation of the system is documented in the next chapter.

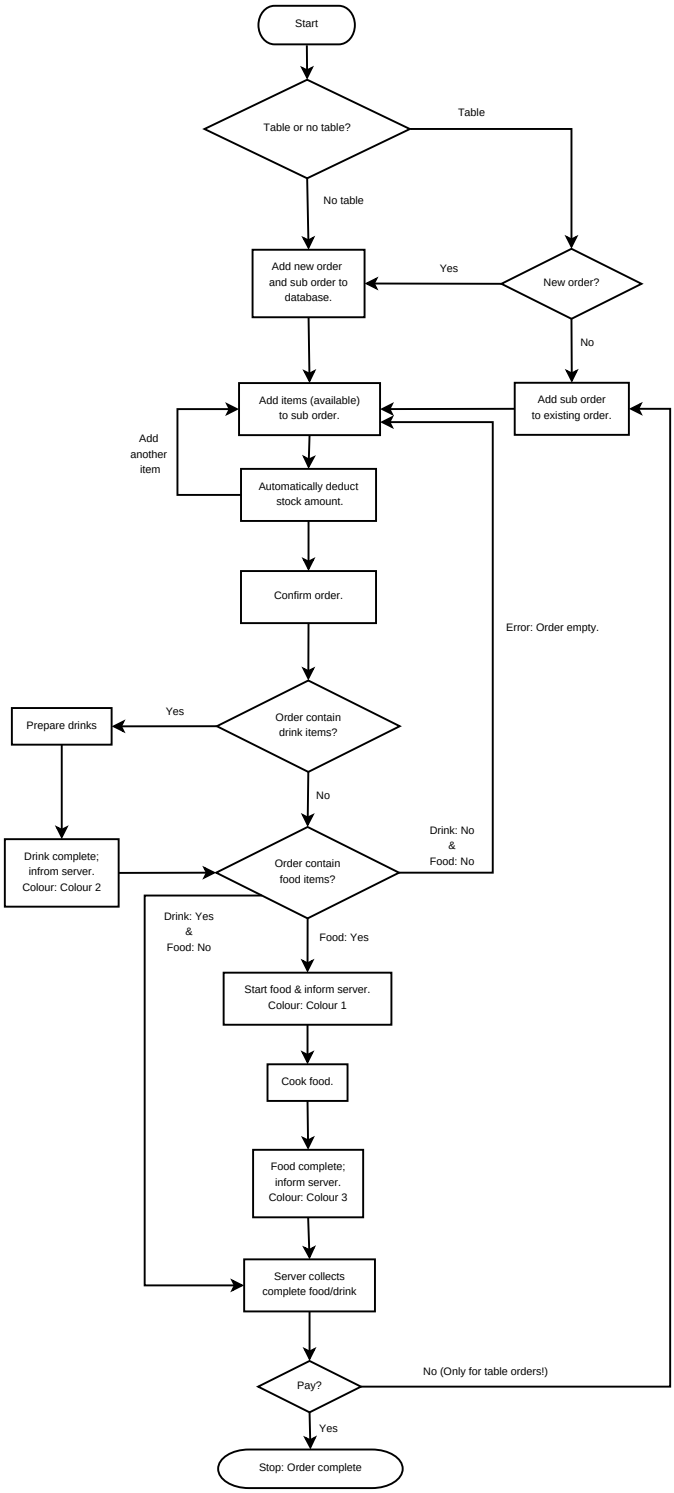

Figure 4.12: Flow chart to show the flow of events of an order.

Chapter 5

Implementation

5.1 Chapter Overview

This chapter discusses the theory behind the implementation of interesting and complex features using pseudocode and diagrams for clear explanations. The implementation stage of any project can be the most enjoyable part and with a well structured plan and appropriate development methodology can be a trouble free stage.

5.2 Implementing Extreme Programming (XP)

As discussed in section 2.8, this project used an agile methodology called Extreme Programming (XP). This type of methodology utilises an iterative process involving customer communication and feedback. The iterations were short with a new software version built at the end of each phase. Version control helped with debugging as the versions could be rolled back to find the iteration that caused the bug.

5.3 Data Storage and Retrieval

As discussed in Section 4.4, a RDBMS was the preferred storage method chosen for this project. The two ER diagrams (Figure 4.3 and 4.4) that were designed in Section 4.4.5 helped with the implementation of the database. A screenshot of the actual implemented database structure can be found in Appendix C.

The structure of the database was just the first step of many with data retrieval being high in priority. Recall from the background on requirement gathering (Section 2.7), most software systems abide by Boehm's cost model (Figure 2.2) so a £1 cost of change in the requirement gathering can cost up to £1000 in the deployment stage to fix. Therefore when implementing the code to retrieve data from the database, we have to consider the case that in the future this system will need maintenance and added features to keep the product competitive within the market. The problem is we cannot predict the future; all we can do is rely on a well structured code base written using all the skills and techniques from the field to ensure easy readability of the code.

To implement this requirement for database retrieval, the code used the object oriented advantages of `Java` to create a database object that provided the database connection and data retrievals. Along with this database object, inheritance played an important role as a generic class created the SQL statements using an array of table field names and input values. Any object that required to send or

retrieve data to the database could extend this generic class giving that object the functionality to insert, delete and update any table within the database.

5.4 Stock Control

This section focuses on how stock control was dealt with by using triggers in the MySQL database. A trigger is some procedural code that is executed in response to data changes within a table. The trigger can react upon inserts, deletes and updates of data within a table and are normally used to code business rules or data integrity checks.

This project used these responsive triggers to keep a stock check field within certain database tables upto date. The stock check field holds an integer depending on the stock level and would update when:

1. An item was selected even before optional ingredient confirmation.
2. An optional ingredient change, therefore a decrease of stock for the ingredient selected and an increase in the original ingredient stock.
3. An optional ingredient removed, therefore an increase in the removed ingredient stock.
4. An order cancelled, therefore an increase in the stock of all the ingredients within that order.
5. An item removed from the order, therefore an increase in the stock of all the ingredients within that item.

When the stock levels changed in any of these circumstances, a database function updated the available stock field to a value between 0 and 4 inclusive. Every ingredient and prepared ingredient link table

Table 5.1: A table showing the available ingredient stock status.

Value	Meaning
0	Compulsory ingredient out of stock.
1	Compulsory ingredient low on stock.
2	Optional ingredient out of stock.
3	Optional ingredient low on stock.
4	No stock issues.

would hold a similar stock status between 0 and 2 inclusive as the ingredient is only defined compulsory or optional within the link between the ingredient and item.

The database functions that updated the available stock fields were:

- `Reduce ingredient stock`: When an item is selected, this function deducted the specified amount from the ingredient stock level.
- `Increase ingredient stock`: When an optional ingredient was removed or swapped, this function increased the specified amount from the ingredient stock level.
- `Reduce prepared ingredient stock`: When an item was selected, this function deducted the specified amount for all the ingredients within the prepared ingredient. This function also accommodated the requirement were a prepared ingredient could be an ingredient of another prepared ingredient.

- `Increase prepared ingredient stock`: When an optional prepared ingredient was removed or swapped, this function would increase the specified amount for all the ingredients within the prepared ingredient. This function also accommodated the requirement were a prepared ingredient could be an ingredient of another prepared ingredient.
- `Update available stock`: When an ingredients stock level changed, this function updated the available stock field within the meal and prepared ingredient table.
- `Update available stock (meal only)`: When a meal added or removed a prepared ingredient to its ingredients list, this function updated the available stock field within the meal table.

Listing 5.1 gives an example of a function within the MySQL database for increasing ingredient stock. Two more coded examples can be found in Appendix D.

Listing 5.1: Database function for increasing ingredient stock.

```
/* MySQL Database procedure: Executed by reactive triggers */
CREATE PROCEDURE `increaseIngredientStock` (IN ingredientID int, IN subOrderProductID int)
BEGIN
  /* Declaring variables */
  DECLARE amount double;
  DECLARE stockID int;

  /* Default measurement specified in the ingredient table */
  if(subOrderProductID = 0) then
    SELECT DISTINCT ingrd.Default_Measurement, Ingrd_Stock_ID into amount, stockID
    FROM ingrd
    WHERE ingredient_id = ingredientID;
  /* Manuaul measurement specified in the meal ingredient table */
  else
    /* Get the manual measurement and ingredient stock ID */
    SELECT DISTINCT meal_ingrd.Manual_Measurement, ingrd.Ingrd_Stock_ID into amount, stockID
    FROM (sub_order_products
      INNER JOIN meal_ingrd ON sub_order_products.Product_ID = meal_ingrd.Product_ID)
      INNER JOIN ingrd ON meal_ingrd.Ingredient_ID = ingrd.Ingredient_ID
    WHERE (((sub_order_products.Sub_Order_Product_ID)=subOrderProductID)
      AND ((meal_ingrd.Ingredient_ID)=ingredientID));
  end if;
  /* Subtract the amount from the ingredient stock table */
  UPDATE ingrd_stock
  SET current_amount = current_amount + amount
  WHERE ingrd_Stock_ID = stockID;
END
```

5.5 GUI Implementation

The GUI implementation follows on from the GUI designs that were discussed in Section 4.5. Recall, that the GUI is one of the most important components of the system as it's the only component the user will ever interact with. Hence by creating an application that has amazing functionality but in a poor command line interface would be enough for the user to dislike the software.

The three GUIs have been coded in Java using the `Swing` classes that are part of the `Javax` package. These classes allow applications to be built using similar controls to that found in many applications therefore first impressions should be recognition of the familiarity of the controls.

The following sub-sections show how some important design concepts were implemented in the GUI. Pseudocode, code snippets and screenshots will all be used to give a clear explanation of the implementation.

5.5.1 Order GUI

The order GUI was by far the most challenging part of the implementation. The basic layout does not need to be explained but just mentioned, with every order containing one to many suborders stored within a table in the database.

Certain components within the GUI required numerous algorithms and specific design features to make the GUI as practical and user friendly as possible.

Synchronized GUI

A feature specified within the order GUI design specification was to keep the GUI in real-time and therefore only show the active menus and only apply the active offers. Hence, the database schema was designed to allow both the menus and algorithms to be time based so that they were only valid at the specified time periods.

This specific specification was implemented by using SQL queries to only extract out the menus currently active and Java threads to keep the GUI synchronised. Listing 5.2 gives an example of an SQL query used within the thread to extract out the active menus. This query compares the menu time against both the order start time and the current time. The reason behind this was that if a customer entered the restaurant when a menu was active but then ordered from the menu when the menu was inactive, the system wouldn't allow this by just comparing the current and menu time.

Listing 5.2: SQL query to extract out available menus.

```
SELECT DISTINCT menu.name
FROM menu
    INNER JOIN availability
        ON menu.Menu_ID = availability.Menu_ID
WHERE   (((availability.Day_ID)=DayOfWeek(Now()))
            AND ((availability.From_time)<=Now())
            AND ((availability.To_time)>=Now())
            AND ((menu.Food)=" + isFoodSection + "))
    OR  (((availability.Day_ID)=DayOfWeek('"+orderDate+"'))
            AND ((availability.From_time)<='"+orderDate+"')
            AND ((availability.To_time)>='"+orderDate+"')
            AND ((menu.Food)=" + isFoodSection + "))
```

The thread constantly called an update function every x amount of seconds with x being the refresh rate specified by the user. Within the same thread, the table grid, menu section and meal item stock colour were updated using similar SQL queries with the latter explained later in this section.

Ingredient choices and cooking preferences

One of the system features gathered in the requirements analysis was to enable the GUI to allow meals to contain optional ingredients that could be interchanged and also to provide the option to select the cooking preference of an ingredient. This requirement was given a priority 1 and the database schema was built to allow this behaviour by allocating both ingredients and prepared ingredients a category. For any ingredient that was classed as an optional ingredient within a meal, could then be interchanged with any ingredient within the same category. The cooking preference was slightly easier as every ingredient could contain a preference and the GUI would only ask for input if the preferences contained two or more options.

The component of the GUI was split into two sides with the optional ingredients on the left hand side and the ingredients that required a cooking preference choice on the right hand side. The two grids could auto extend down but kept a static grid width with numerous colours used to represent the different states. Tables 5.2 and 5.3 defines the meaning of the different colours used in the following four situations:

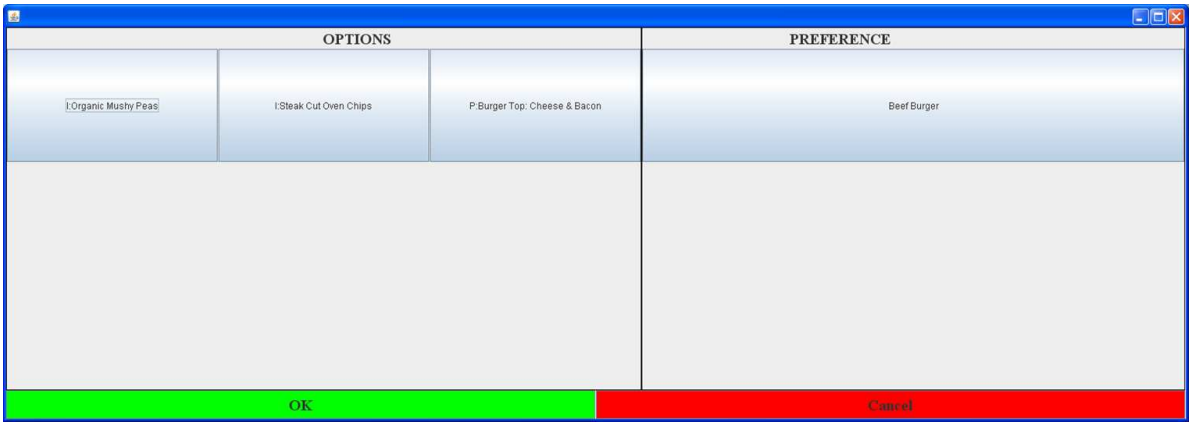

Figure 5.1: A screenshot of the implemented optional and preference GUI component.

1. Optional grid showing the meal ingredients.
2. Optional grid showing the options of the meal ingredients.
3. Preference grid showing the ingredients within the meal that requires a cooking preference.
4. Preference grid showing the preference options of an ingredient within the meal.

Table 5.2: Available states within the ingredient option grid.

Grid	Colour		Illustration
Ingredients	Yellow	Original optional ingredient replaced.	Figure 5.2
	Red	Original optional ingredient removed.	Figure 5.3
Options	Green	Current selected ingredient.	Figure 5.4
	Red	Button to removed the ingredient.	Figure 5.4

Table 5.3: Available states within the preference grid.

Grid	Colour		Illustration
Ingredients	Yellow	Preference selected for item.	Figure 5.5
Preferences	Green	Current selected preference.	Figure 5.7

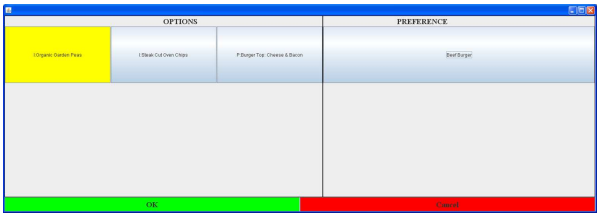

Figure 5.2: Screenshot showing a replaced optional ingredient.

Figure 5.3: Screenshot showing an optional ingredient removed.

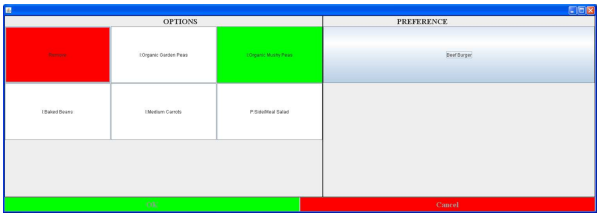

Figure 5.4: Screenshot showing the optional ingredient options with the current selected ingredient in green.

Figure 5.5: Screenshot showing the selected ingredient preference.

Figure 5.6: Screenshot showing the list of preferences available for the ingredient.

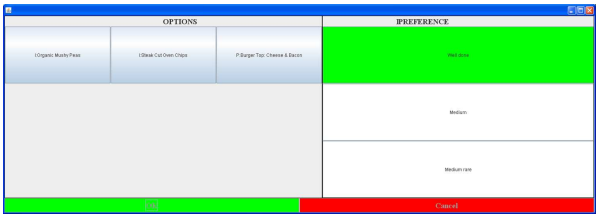

Figure 5.7: Screenshot showing the list of preferences available for the ingredient with the current selected preference in green.

Grid Implementation

Many of the components within the order GUI implemented a grid algorithm. This grid algorithm used a basic layout, where each component appeared one after the other, but with the added functionality of auto expanding rows. Listing 5.3 shows the generic psuedocode that implemented this simple grid expansion.

Listing 5.3: Grid algorithm pseudocode.

```
% Static Column Size
rowSize = 0;
count = calculateItemCount();
if( count >= (width * height))
    newHeight = count/width;
if(count % width > 0)
    newHeight ++;
    buttonGrid = new ButtonArray [width] [newHeight];
else
    buttonGrid = new ButtonArray [width] [height]
end if
initGrid(buttonGrid);
```

However the main grid implemented a different algorithm as explained in the design Section 4.5.1. Recall, that this grid would show the meals and drinks, but would be separated into blocks of sections with each section starting on a new row. The added problem was that the grid would not be a basic grid but column based. An example would be, an 8 x 8 grid could represent as a 16 x 4 grid. Listing 5.4 shows the pseudocode of the algorithm used to create and fill this type of grid.

Listing 5.4: Item grid algorithm psuedocode.

```
Define 2D buttonArray;
for (int i = 0; i< (heightAmount/columnGroups); i++){
  // Loop through each component on that row
  for (int j = 0; j < (widthAmount*columnGroups); j++){
    // x and y variables of the array of JButtons
    int x;int y;
    // If the modulus of button in the ith row is greater than width
    // => then the button is in the second row
    if ((j % (widthAmount*columnGroups)) >= widthAmount){
      // X will be the number subtract width,
      //as the array is from 0=>width-1 and
      //we are dealing with width+1=>width*columnGroups
      x = j - widthAmount;
      // increment by height/columngroups
      y = i + (heightAmount/columnGroups);
    }else {
      // Otherwise just x and y
      x = j;
      y = i;
    } // else
    Create tempButton;
    Set tempButton invisible;
    Assign button into buttons x,y element in buttonArray;
  } // for
} // for
```

Stock control

The stock control was implemented using triggers within the database as discussed in Section 5.4. This sub-section explains how the system would visually display the meals.

Recall that an integer in the range of zero to five was applied to all meals with the integer referring to the ingredient stock statuses defined in table 6.1 in Section 5.4. The Java thread that kept the GUI

synchronised with the available menus and tables also implemented a function to highlight the specific meals that appeared under any status where an ingredient was low or out of stock.

Each button that represented a meal or drink would have a specific coloured border. The border would be updated depending on the index of that meal within the database. Table 5.4 defines the five different statuses available and the colours allocated for that status.

Table 5.4: A table showing the available ingredient stock status and the allocated colour.

Value	Defining	Colour
0	Compulsory ingredient out of stock.	Red
1	Compulsory ingredient low on stock.	Orange
2	Optional ingredient out of stock.	Yellow
3	Optional ingredient low on stock.	Blue
4	No stock issues.	Black

5.5.2 Kitchen GUI

The kitchen GUI was far simpler to implement than the order GUI as the GUI only required to display data without much human input. The orders stored in the database, which were created using the order application, would be accessed by the kitchen GUI.

The GUI used four Java `Swing JTables` with three of the `JTables` showing the items within the user specified menu sections (i.e. starters, fish, puddings etc) and the final `JTable` showing any other ordered item not currently displayed.

An order start button would update the status of the order, giving the waiters feedback on the order status. Each food item within the order contained a completed field within the order database table and would be checked once complete. Once all items within the order were complete the waiter would be automatically informed and the order removed from the kitchen display once collected.

To help the readability of the kitchen GUI, row colours within the table were implemented. The tables would also display statistics about the order including elapsed kitchen time, order progress[1] and section progress[2].

Finally, a Java thread was used to update the GUI every x seconds where x is an integer specified by the user thus keeping the GUI constantly synchronised.

5.5.3 Management GUI

The management GUI was the easiest GUI to implement, as the GUI was only used to add, update and delete records from the database as well as displaying bulk data sets. The GUI used numerous techniques to make the input of data as easy as possible including the use of wildcard character searching, consistent design layout and validation checks with user prompts.

5.6 Pricing Algorithm

Applying numerous offers to an order to generate the best price for the customer was discussed in the design chapter (Section 4.6). This section will now go into the implementation details of the pricing algorithm using pseudocode to show the techniques used.

[1] How many items within the order were complete.
[2] How many items within the section were complete

As discussed in the design, the exhaustive algorithm approach was chosen to find the best price. The algorithm can be drawn into a tree structure where the root is the initial set of items from the order and each child node is a subset of its parent's node after the application of an offer.

As with most tree structure algorithmic solutions, the code solution used recursion.

Listing 5.5: Basic pseudocode of the price algorithm.

```
Array offerArray
Array orderArray
/* Active:  1) Available at present time
            2) At least 1 item in order exists in the offer */
SQL Query: Get all active offers
if no active offers
    No reduction in price
else
    Add order items to order array
    loop through each active offer
        Create offer object
        Only add the items within both the offer and order to the offer object sets
        Add offer object to offerArray
    end loop
    call recursive method with offerArray and orderArray
end if

/*****************************************************************************/
Recursive method (offerArray, 2dOrderArray)

/*  2dOrderArray holds database item ID and price
    Price calculated along the way */

for (i = 0; i < offerArray size; i++)
    boolean success = applyOffer to 2dOrderArray /* Changes prices within 2dOrderArray */
    if (success = true)
        call recursive method(offerArray, new2dOrderSet)
    else
        cheapestPrice(2dOrderArray)
    end if
end for
```

The pseudocode in Listing 5.5 gives a basic idea as to how the algorithm was created, but in fact the coded version was much more complicated.

To increase the efficiency of this exhaustive algorithm, SQL was used in the generation of the offer objects, so that the sets within the offer objects only contained items currently within the order, hence a reduction in the amount of comparisons.

The fully coded version of the algorithm can be found in Appendix E.

5.7 Code Documentation

Code documentation is an important part of any software engineering project. Throughout the implementation, a JavaDoc tool was used to generate HTML API documentation of the project. The JavaDoc could then be used to provide assistance to any future developer.

5.8 Version Control

Due to the type of development methodology used for this project, incremental backups of the system were required. Version control systems (also known as Revision control) such as Mercurial manage the changes to documents storing each backup in its own revision with the ability to restore back to a particular version in the event of debugging.

For this project, each iteration within the development methodology utilised the version control system by archiving and saving each prototype.

5.9 Error Logging

Within this project, error logging was implemented to log all thrown exceptions within the code. This gave the developer a better understanding of any errors that occurred in run-time leading to efficient debugging.

The errors were logged in a flat text file. Listing 5.6 shows an example of the output within the log but do note that the first column would have stored the user that was currently logged in, but the user feature was never implemented due to time constraints.

Listing 5.6: Error logging sample.

```
USER! Tue Mar 23 13:22:49 2010  NullPointerException: Required values not filled in!
USER! Tue Mar 23 13:45:03 2010  NullPointerException: No value selected
USER! Tue Mar 23 14:19:03 2010  Exception: Insert Error: Duplicate entry 'TEST' for key 'name'
USER! Tue Mar 23 14:43:41 2010  NullPointerException: All required fields need to be filled in
```

5.10 Chapter Summary

This chapter has discussed the interesting aspects from the implementation stage. The next chapter documents the results by demonstrating the working system.

Chapter 6

Results

6.1 Chapter Overview

This chapter demonstrates the working system using screenshots. The layout is set out to show the chronological order in how you would use the system similar to a walk through.

6.2 Management Application

The first stage of setting up this system was to enter all the required data. The data inputted into the system was real data from two menus used by a Beefeater Grill [6] pub in Ormskirk, Lancashire.

The management application consisted of 5 sections:

- Data entry (excluding offer entry).
- Stock management.
- Offer entry.
- System Settings.
- Statistics.

6.2.1 Data Entry

All the data input forms were contained within the management application. The following screenshots show some of the data input forms for inputting the menus and associated ingredients. Please note that validation will not been shown within the results as the database design took care of the majority of the validation.

All the forms kept a consistent layout, with add, save and delete command buttons performing the operations. The add command button would open a new form with blank fields, the save command would save any changes the user would make to a selected item and the delete command button would delete the selected item. For any list or table that held more than a few items, a searching facility was provided.

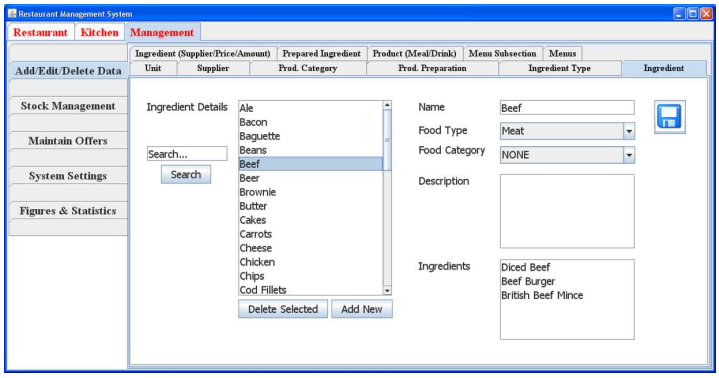

Figure 6.1: A screenshot of the generic ingredient form.

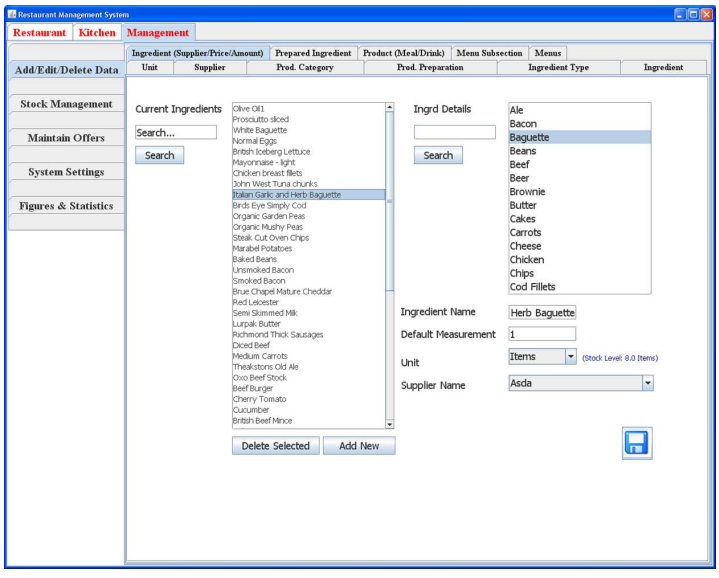

Figure 6.2: A screenshot of the ingredient form.

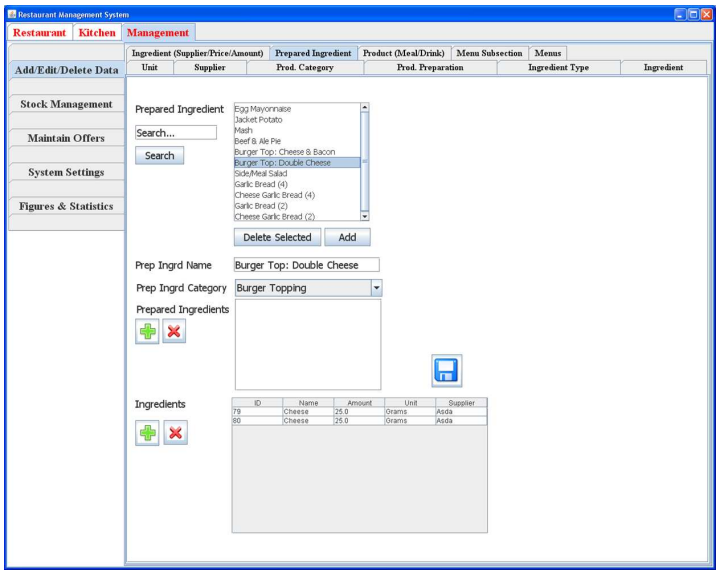

Figure 6.3: A screenshot of the prepared ingredient form.

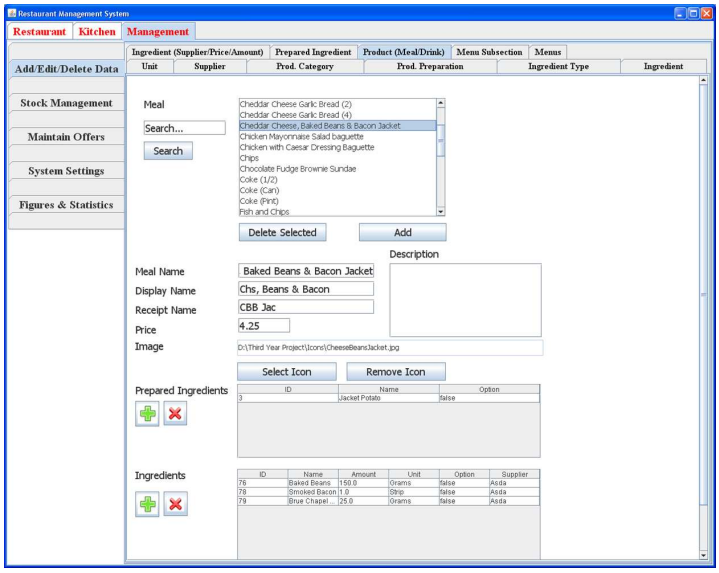

Figure 6.4: A screenshot of the product form.

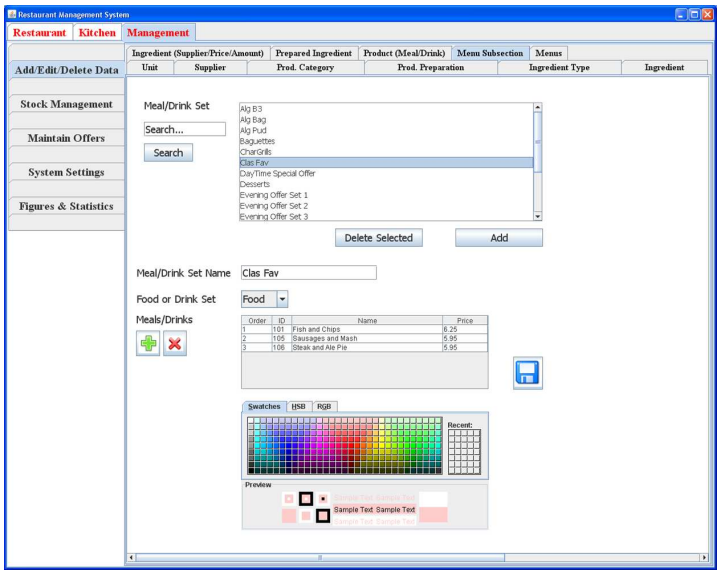

Figure 6.5: A screenshot of the menu section form.

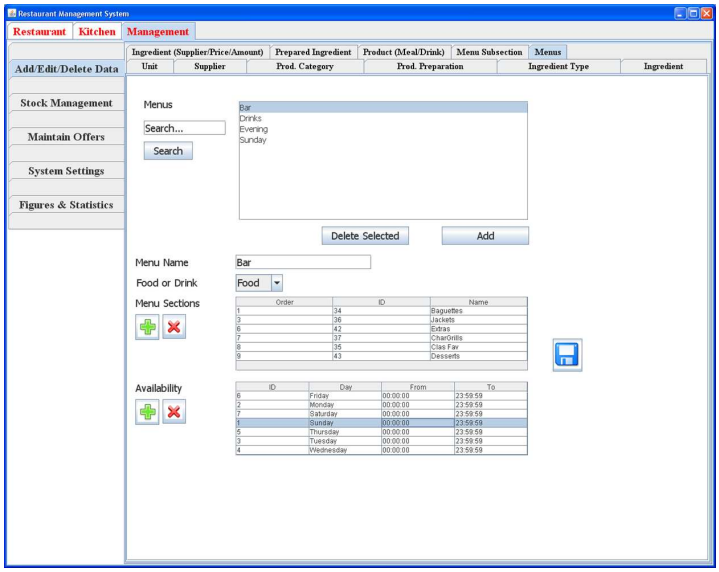

Figure 6.6: A screenshot of the menu form.

6.2.2 Stock Management

Stock control was a vital part of the system with the stock management form allowing the management to view the current stock levels. Figure 6.7 shows a screenshot of the stock management form which allowed the following operations:

- Save (Stock Check): The restaurant could perform stock checks.
- Add Stock: On delivery of stock, the management could add the new stock to the system even if the prices were different. Figures 6.8 and 6.9 show screenshots of the required input to add new stock to the database.
- Refresh: Function to refresh the table.

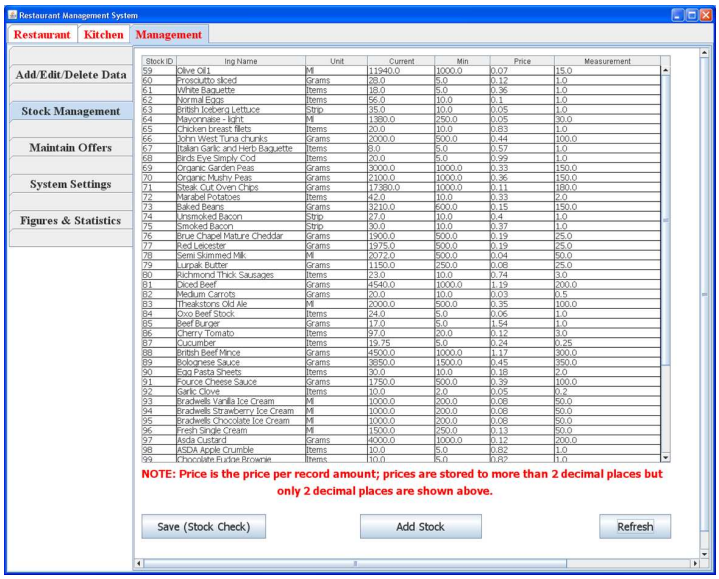

Figure 6.7: A screenshot of the stock management form.

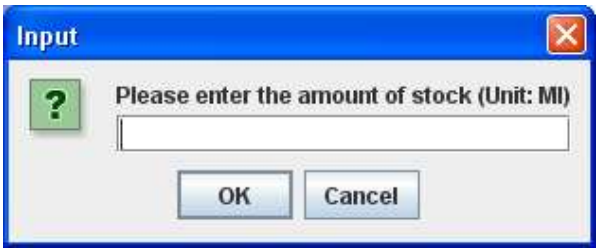

Figure 6.8: A screenshot of the first input value required when adding stock.

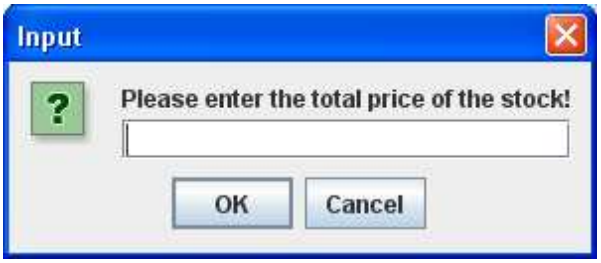

Figure 6.9: A screenshot of the second input value required when adding stock.

6.2.3 Offer Entry

Recall from Section 4.6 that the pricing algorithm could take the form of the following 2 different generic offer types:

1. Buy X from A, Y from B and Z from C for £S
2. Buy X from A and get S% of Y from B,

where X,Y and Z are positive integers, A,B and C are sets associated with the offer and S is a positive real number. Also recall that an offer is bounded within its specified time period and hence only becomes active in the times specified.

Within the system, the two algorithms were separated into two different forms to avoid confusion. Screenshots of the two offer input forms can be found in figures 6.10 and 6.11.

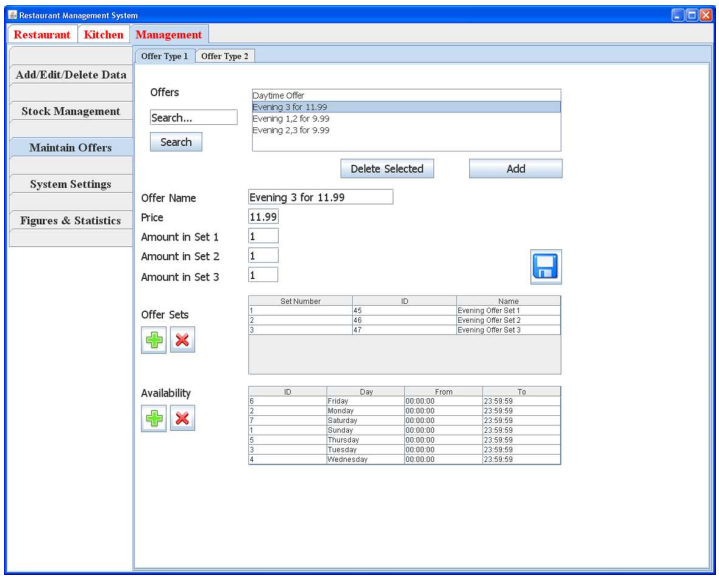

Figure 6.10: A screenshot of the form used to input the type 1 offer.

In Figure 6.10, the example within the screenshot demonstrates an offer of buy 1 item from set 'Evening Offer Set 1', 1 item from set 'Evening Offer Set 2' and 1 item from set 'Evening Offer Set 3' for £11.99. The offer is available every day.

In Figure 6.11, the example within the screenshot demonstrates an offer of buy 1 item from set 'Alg B3' and get 1 item from set 'Alg Pud' for free. The offer is only available on Wednesday.

Figure 6.12 and 6.13 shows screenshots of the input forms used to add offer sets and availability to either offer one or offer two.

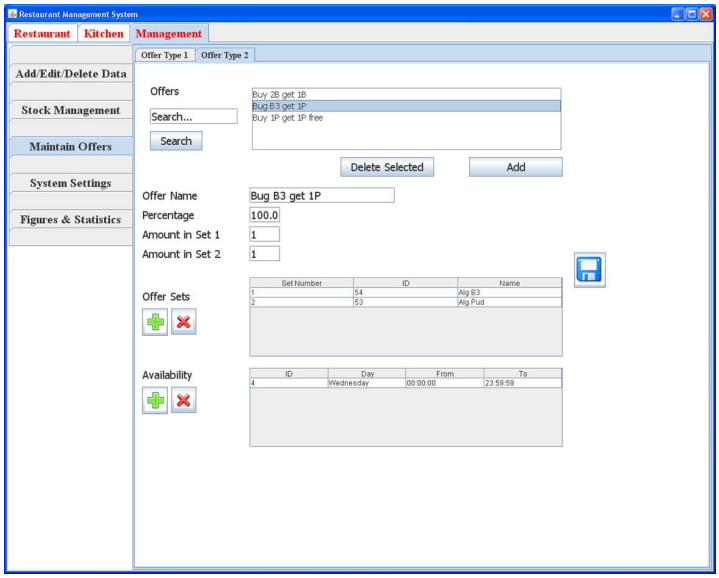

Figure 6.11: A screenshot of the form used to input the type 2 offer.

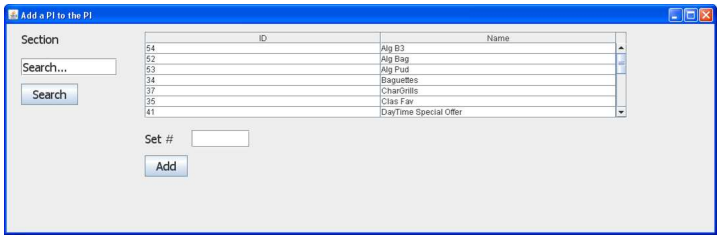

Figure 6.12: A screenshot of the form used to add the sets within an offer.

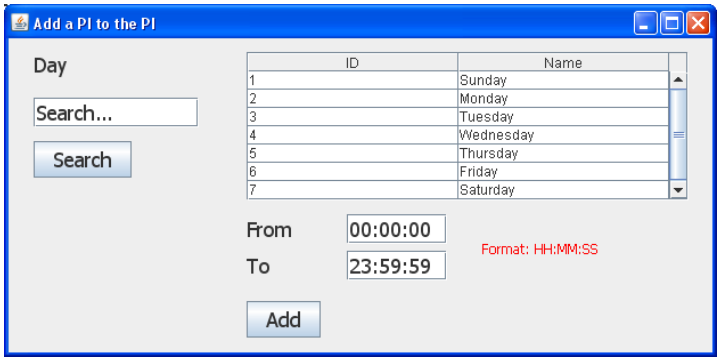

Figure 6.13: A screenshot of the form used to add the availability of an offer.

6.2.4 System Settings

The system settings defined the grid sizes and other properties for the different desktop computers and PDAs connected. The following table explains how each property changed the GUI's look.

Table 6.1: A table defining the menu properties within the system settings form.

Property	Result
Menu: Height	The height of a sub grid within the main grid.
Menu: Width	The width of a sub grid within the main grid.
Menu: Text Size	Text size of the text within the main grid.
Menu: Column Groups	The amount of sub grids from left to right within the main grid.
Menu: Character Limit (per line)	The maximum amount of characters per line for the item buttons.
Menu: Transparency ([0,1])	How transparent the image behind the text is for the item buttons.

The form also contains similar properties for the menu section and colour selectors for the waiter calls. Figure 6.14 gives an example of settings for a desktop with the resulted layout in figure 6.16. Figure 6.15 gives an example of settings for a PDA with the resulted layout in figure 6.17.

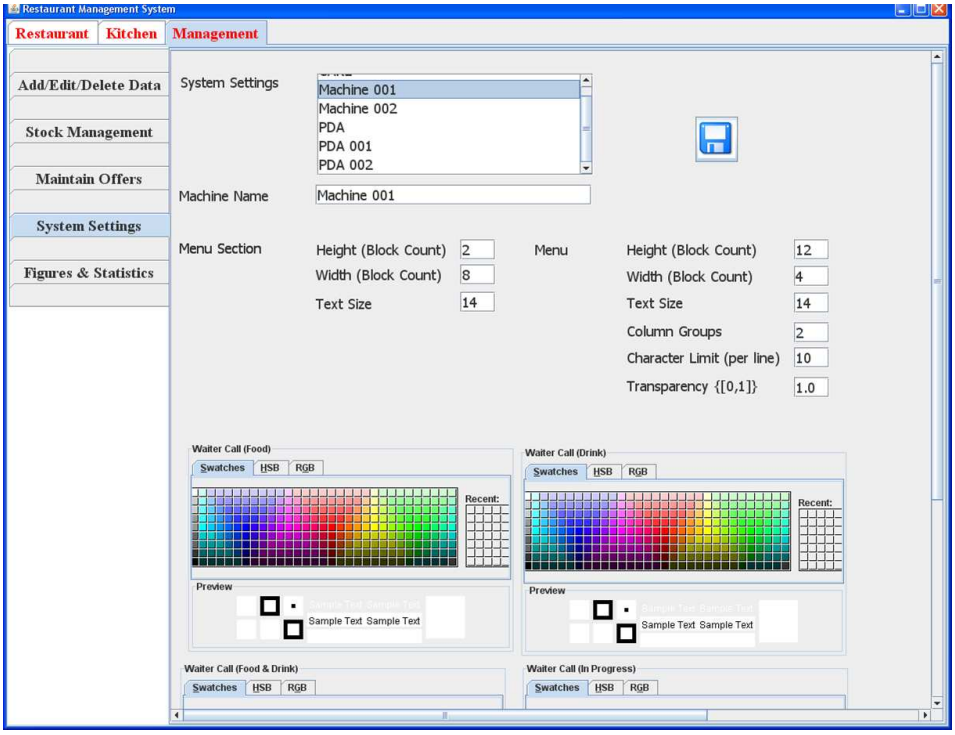

Figure 6.14: A screenshot of the system settings for a desktop computer.

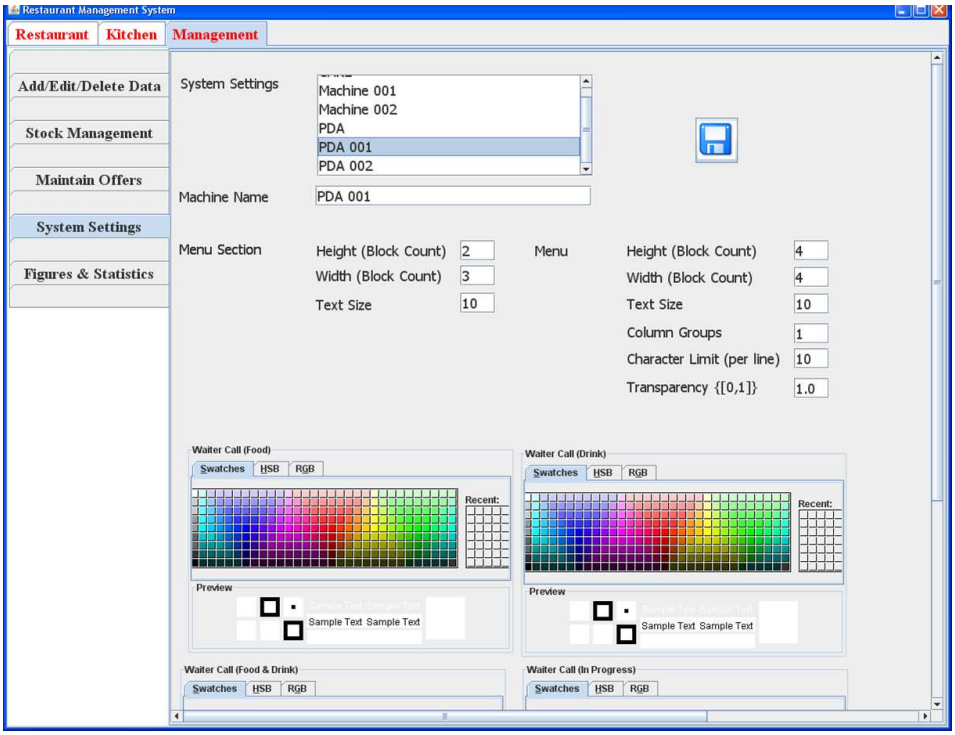

Figure 6.15: A screenshot of the system settings for a PDA.

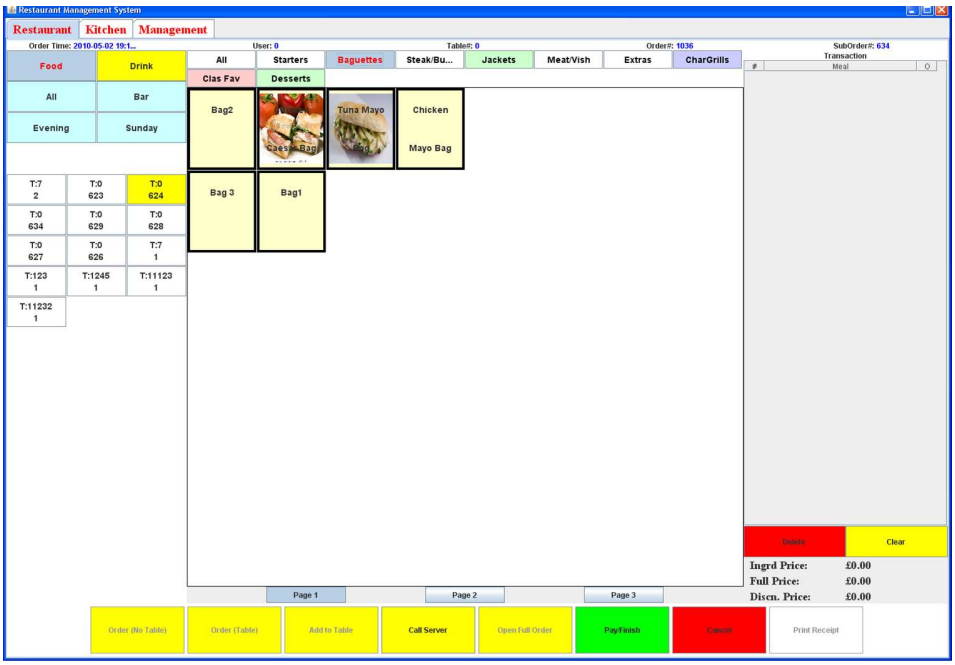

Figure 6.16: A screenshot showing the result of using the system settings in figure 6.14.

Figure 6.17: A screenshot showing the result of using the system settings in figure 6.15.

6.2.5 Statistics

The final section within the management application was to show daily, monthly and year statistics. Unfortunately as this was a priority 3 requirement, only a couple of graphs were implemented.

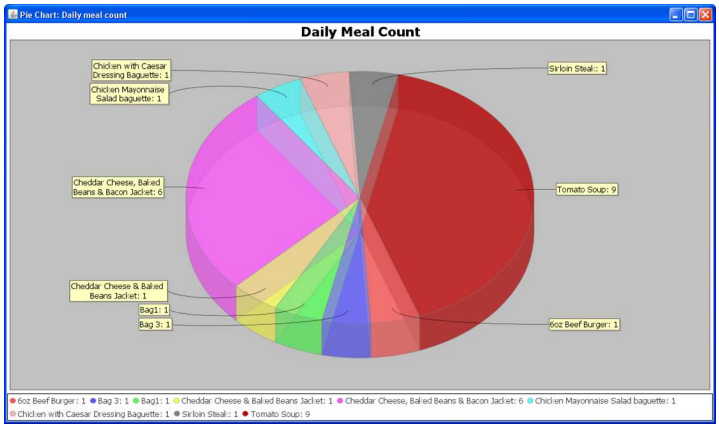

Figure 6.18: A screenshot showing a pie chart of the daily meal count within the restaurant.

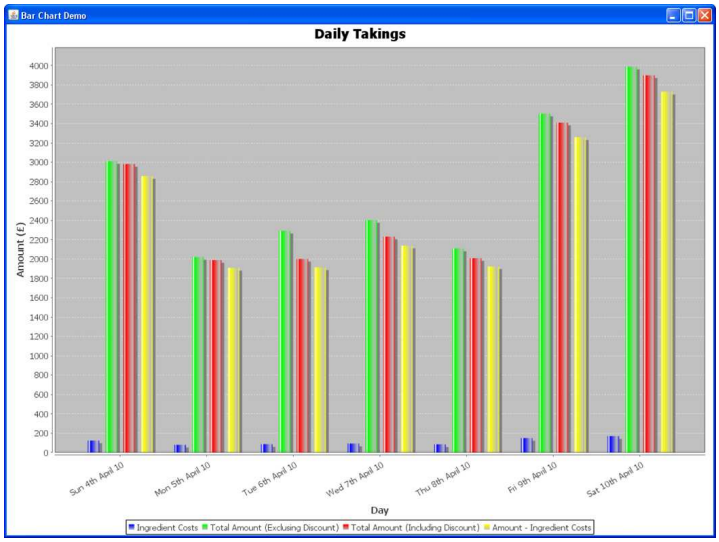

Figure 6.19: A screenshot showing a bar chart of the weekly takings within the restaurant.

6.3 Order Application

The following annotated screenshots show the functionality of the order application. The screenshots are ordered to give the reader an idea of the steps involved in order.

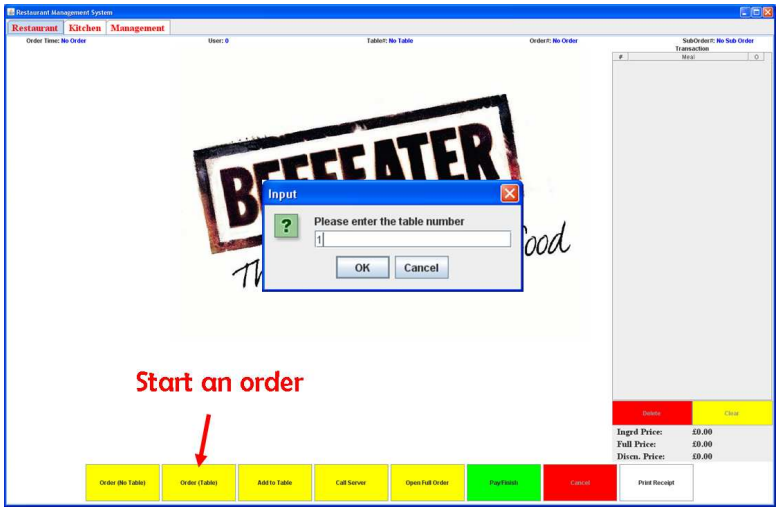

Figure 6.20: A screenshot showing the start of order.

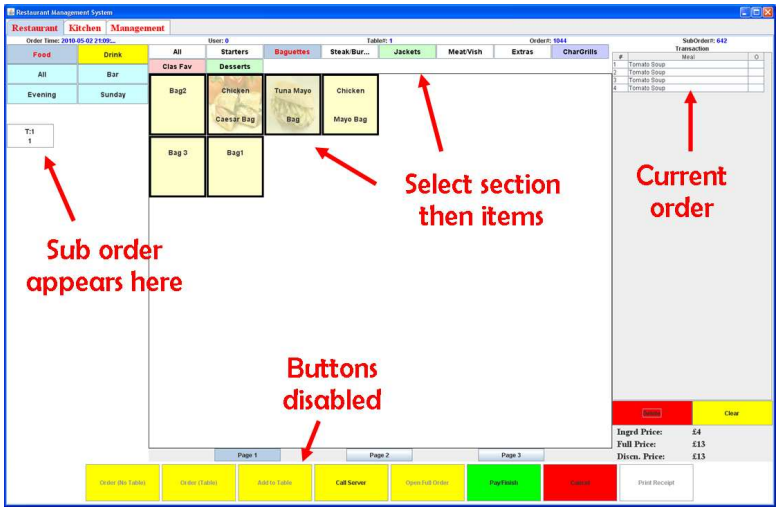

Figure 6.21: A screenshot showing the different components within the GUI.

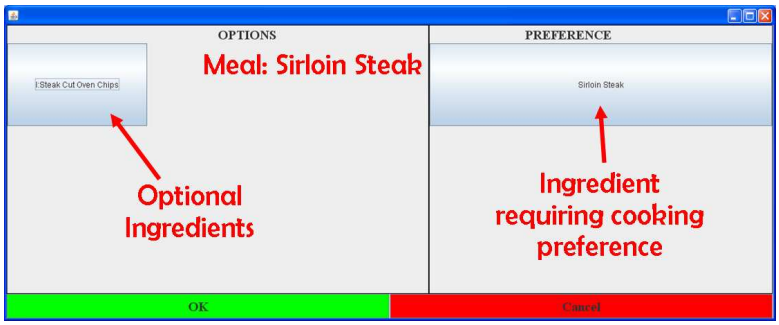

Figure 6.22: A screenshot showing the options sub-form when selecting 'Sirloin Steak'.

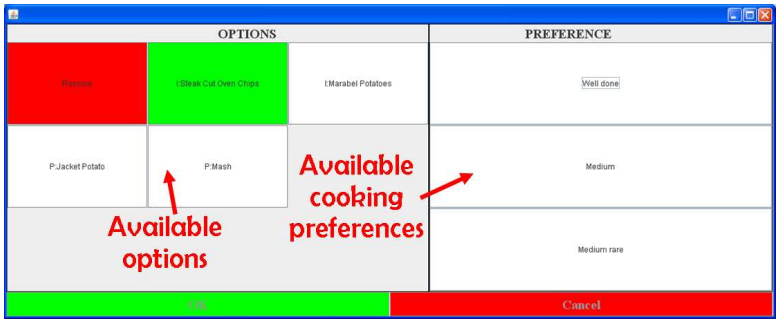

Figure 6.23: A screenshot showing the available options and cooking preferences.

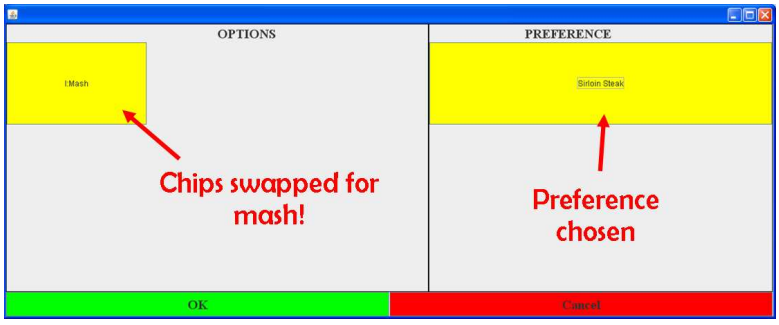

Figure 6.24: A screenshot showing the optional ingredients being swapped.

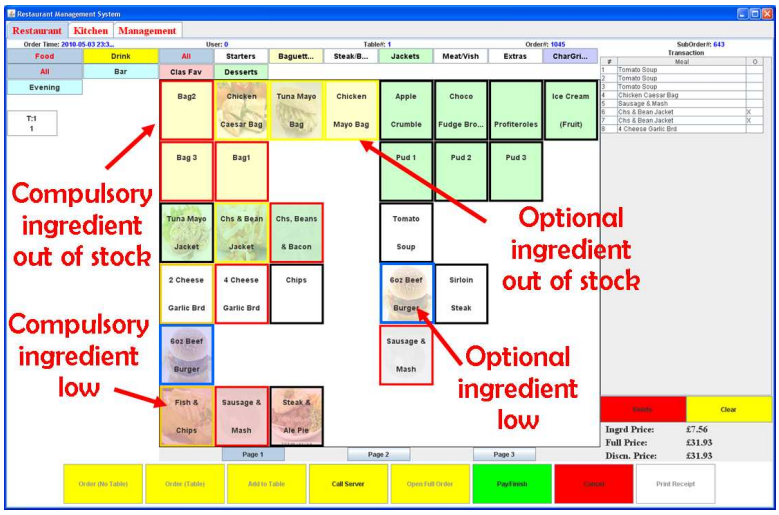

Figure 6.25: A screenshot showing how the GUI displays meals with low or no stock.

Figure 6.26: A screenshot showing the status of a suborder once confirmed by the chef.

Figure 6.27: A screenshot showing the details of an order.

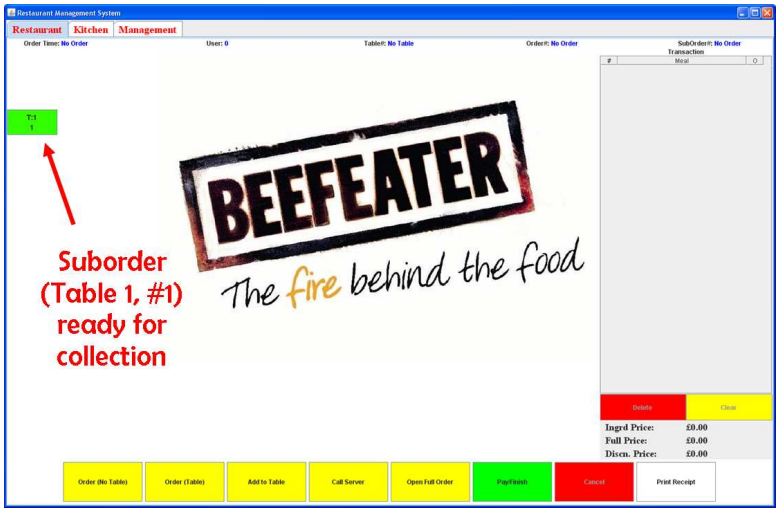

Figure 6.28: A screenshot showing the status of a suborder when ready.

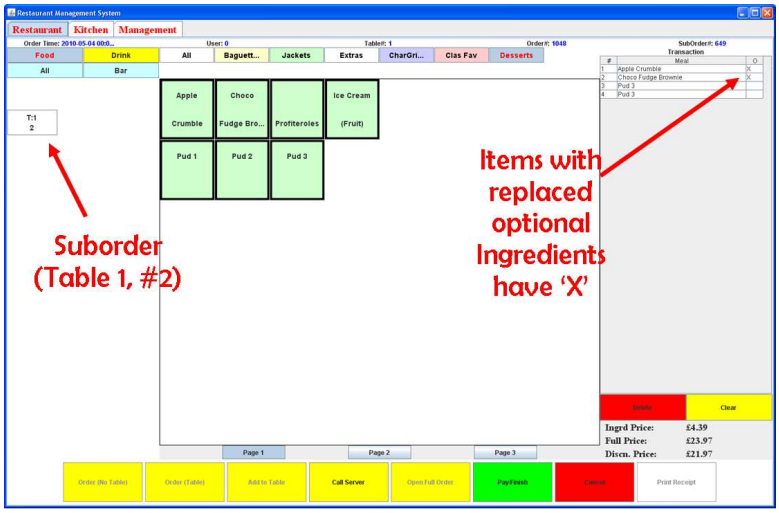

Figure 6.29: A screenshot showing the second suborder within an order.

Figure 6.30: A screenshot showing suborder complete; removed from suborder grid.

Figure 6.31: A screenshot of the receipt with the discount applied.

6.4 Kitchen Application

The following three annotated screenshots show the functionality of the kitchen display.

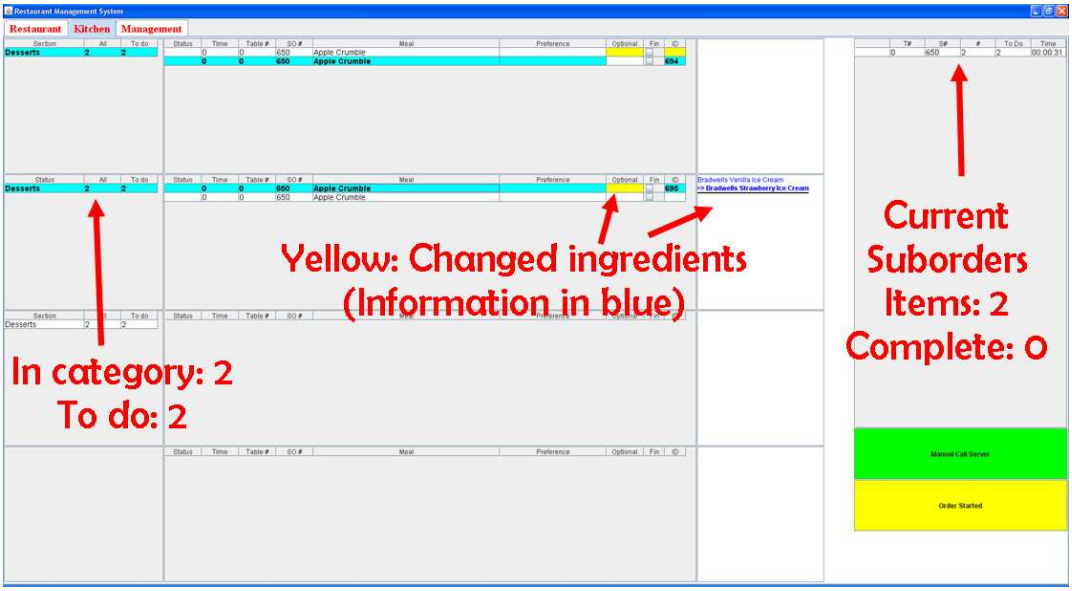

Figure 6.32: A screenshot of the kitchen application.

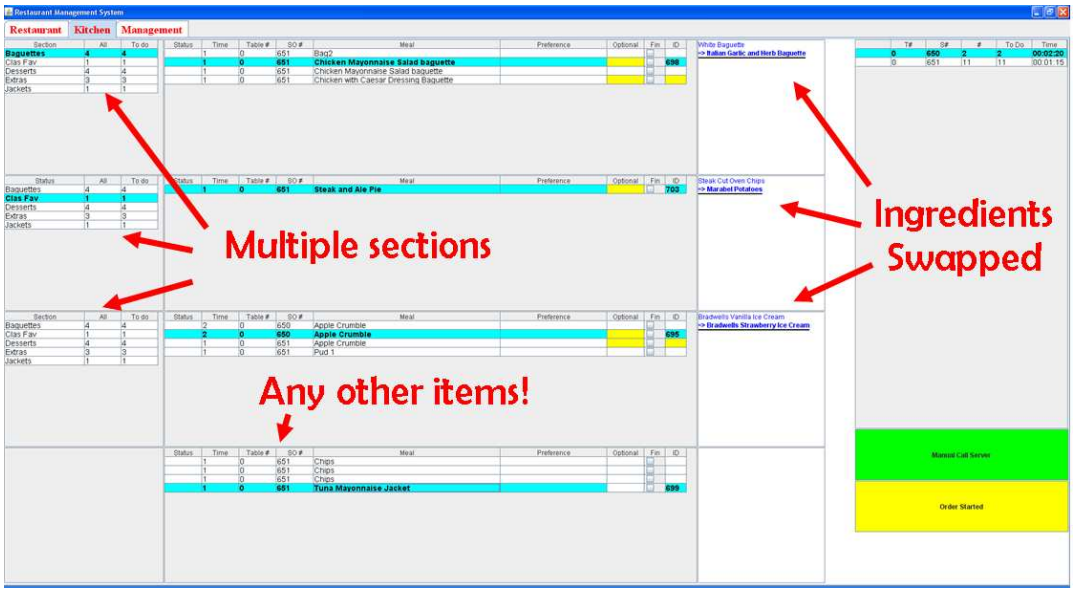

Figure 6.33: A screenshot of the kitchen application.

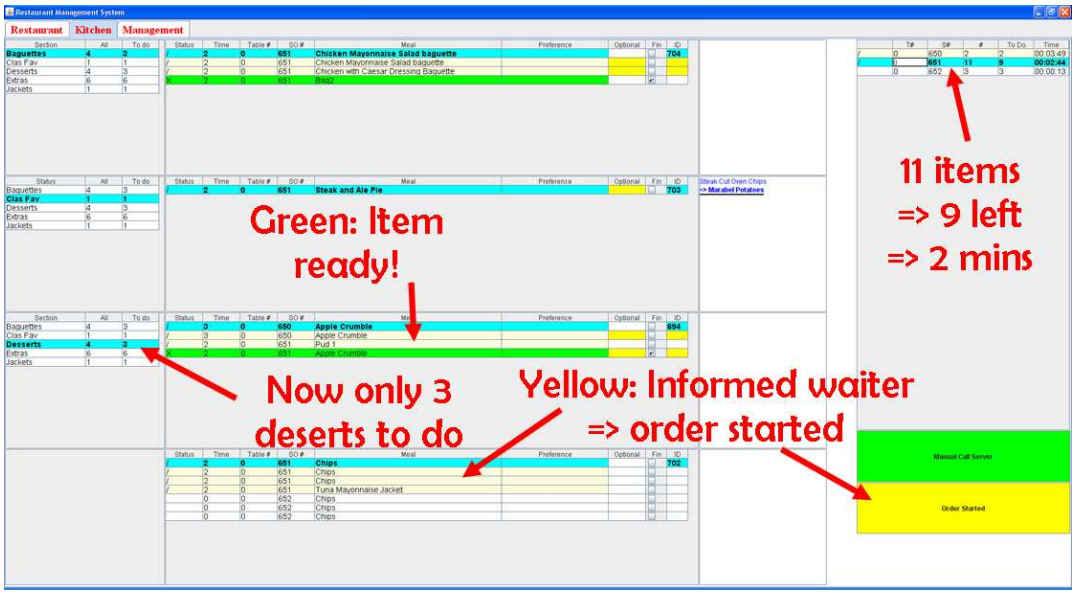

Figure 6.34: A screenshot of the kitchen application.

6.5 Chapter Summary

This chapter has demonstrated the system using screenshots. The next chapter will discuss how the software system was tested.

Chapter 7

Testing

7.1 Chapter Overview

This chapter describes the different testing techniques used to test the system including the results of some of the testing techniques applied. The tests did not just test the code directly but also tested some of the non-functional aspects of the system

7.2 Testing Techniques

This section explains the different testing techniques that were used when implementing the extreme programming (XP) software development methodology. Recall, that with this development methodology, testing is carried out at the end of each iteration.

7.2.1 Unit Testing

Unit testing is a testing technique to verify and validate individual units of code. According to Don Wells [19], unit tests are one of the corner stone's to XP. With the generation of the unit test cases early within the development cycle, the test cases were repeatedly used at the end of each iteration to check the new functionality added does not have any unintended side effects.

Netbeans, the IDE used for this project, contained a tool that could generate `JUnit`[1] test templates for any source file. Within the project, all source file functions were tested, with simple pass/fail output and necessary warning messages. Listing 7.1 shows an example test case that tests the insert, update, delete and extraction of units[2] within the database with the output shown in Figure 7.1.

Listing 7.1: Java unit test case for a 'unit' object.

```
import java.sql.ResultSet;
import junit.framework.TestCase;

public class UnitTest extends TestCase {
    int databaseID = 90;                        // Specified database ID

    public UnitTest(String testName) {super(testName);}
    @Override
    protected void setUp() throws Exception {super.setUp();}
    @Override
```

[1] Unit testing framework for the Java programming language.
[2] A unit is how the ingredient was measured (i.e. grams, litres etc)

```
    protected void tearDown() throws Exception {super.tearDown();}
    /** Test of insert method, of class Unit.*/
    public void testInsert() throws Exception {
        System.out.println("insert");
        String unit = "UNIT_TEST_CASE";
        Unit instance = new Unit();                    // Creates unit object
        int expResult = databaseID;                    // Expected result specified above
        int result = instance.insert(unit);            // Insert method
        assertEquals(expResult, result);               // Compare expected ID to returned ID
    }
    /**Test of editRecord method, of class Unit.*/
    public void testEditRecord() throws Exception {
        System.out.println("editRecord");
        String name = "UNIT_TEST_CASE";                // Old name
        String newName = "UNIT_TEST_CASE_NEW_NAME"; // New name
        Unit instance = new Unit();                    // Creates unit object
        instance.editRecord(name, newName);            // Edit method
        instance.getUnit("UNIT_TEST_CASE_NEW_NAME");// Gets the new unit object
        String expName = instance.getUnitName();
        assertEquals(expName, newName);
    }
     /**Test of getUnitName method, of class Unit.*/
    public void testGetUnitName() throws Exception {
        System.out.println("getUnitName");
        Unit instance = new Unit();                    // Creates unit object
        String expResult = "UNIT_TEST_CASE_NEW_NAME";
        instance.getUnit(expResult);                   // Gets the new unit object
        String result = instance.getUnitName();       // Returns unit name
        assertEquals(expResult, result);
    }
    /**Test of deleteRecord method, of class Unit.*/
    public void testDeleteRecord_String() throws Exception {
        System.out.println("deleteRecord");
        String name = "UNIT_TEST_CASE_NEW_NAME";
        Unit instance = new Unit();                    // Creates unit object
        boolean expResult = true;                      // Expected result
        boolean result = instance.deleteRecord(name);// Returned result
        assertEquals(expResult,result);
    }
}
```

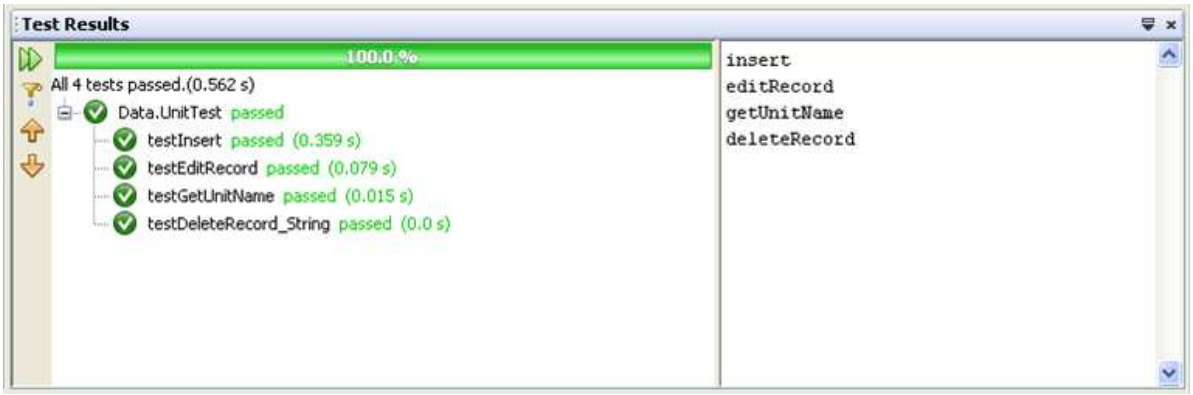

Figure 7.1: Screenshot of the result produced using `JUnit` in Netbeans.

7.2.2 User Acceptance Testing

User acceptance testing is a black-box[3] approach to testing the functionality of the system.

[3]An approach were the tester has no knowledge of the internal structure of the function they are testing.

In this project, at numerous iterations, the customer sometimes defined as the subject matter expert (SME) trialled and reviewed the current prototype of the system. The customer then gave feedback and confirmed whether the system met the agreed-upon requirements. This type of testing only appeared in the last few iterations, as the GUI was one of the latter features to be developed and without a GUI, user acceptance testing would gain no benefit.

7.2.3 Usability Testing and Usability Inspection

Usability testing is similar to user acceptance testing where the product is evaluated by the end user and gives direct feedback on how real users use the system.

This testing technique was performed at the same time as user acceptance testing with the feedback used to alter any important concerns.

An example where user feedback was pivotal was in the initial design where the GUI implemented lists that displayed the menus, sections and tables. When the end user tested the GUI, the lists appeared to be a stumbling block and the users were finding it difficult to select the option they wanted. This problem could have been fixed by increasing the size of the text within the list but would have resulted in the addition of scroll bars. However, this brought around the design decision to change these lists into grids which the users found to be much easier to use.

Usability inspection is where the developer instead of the end user inspects the user interface. This testing technique was mainly used to inspect whether the GUI would be adequate, with the speed of an order input critical to the restaurants efficiency.

Therefore the developer tested the time complexity of the amount of clicks required to create an order.

The worst case scenario:

1. Select a table (2 clicks).
2. Select a meal (1 click).
3. Meal has o amount of optional ingredients with all optional ingredients being removed or swapped ($2o$ clicks).
4. Meal also has p amount of ingredients each requiring a preference choice ($2p$ clicks).
5. Repeat steps 2,3 and 4 for m amount of meals ($m(1 + 2o + 2p)$ clicks).
6. Confirm the suborder (1 click).
7. The order contains s amount of suborders so repeat steps $1 - 6$, s number of times.
8. Confirm the suborder (2 clicks).

Therefore in the worst case scenario, the time complexity would be $O(s(3 + m(1 + 2o + 2p)) + 2)$.

The best case scenario:

1. No table selected[4](1 click).
2. Select a meal (1 click).
3. Meal has 0 optional ingredients and 0 preference choices.
4. Repeat step 2 for m amount of meals (m clicks).

[4]An order when a table is not specified such as a drinks order in a pub.

5. Confirm the suborder (1 click).
6. The order contains only 1 suborder.
7. Confirm the suborder (2 clicks).

Therefore in the best case scenario, the time complexity would be $O(1 + m + 1 + 2)$.

Hence the time complexity is of polynomial complexity in the worst case scenario and linear complexity in the best case scenario.

7.3 Testing Analysis

Throughout the design and implementation, the look and feel of the GUI was of the highest importance. However it became apparent within the testing stage that numerous design decisions didn't quite work.

The grid that displayed the items in the centre of the GUI, used an algorithm to try and keep a consistent grid layout, even on the addition of new items. However the disadvantage was the amount of blank cells within the grid that were unusable due to this algorithm. Hence a more appropriate algorithm should have been applied that didn't leave blanks cells but still kept a consistent layout.

Another problem was the use of both images and text on the item buttons. Instead of making the buttons easier to find, the end-users actually struggled to find the item they wanted. Hence either images or text should be chosen but not a mixture of the two.

7.4 Chapter Summary

This chapter has documented numerous testing techniques applied to this project. The next chapter concludes the report.

Chapter 8

Conclusion

8.1 Chapter Overview

This chapters draws the project report to a close and reflects on the design decisions made throughout. It also discusses possible future development ideas.

8.2 Project Overview

The system achieved all of its proposed priority 1 and priority 2 functional requirements and even some priority 3 outlined in Section 3.5.

However, the initial project plan and gannt chart had to be modified as the project became about a month behind due to underestimations on the time to implement some desired features. This meant that some of the lower priority requirements had to be scrapped.

8.3 Further Development

This project was developed under time constraints of 120 hours. Therefore the proposed features specified in the requirements were what the developer thought to be realistic targets.

However, if more time became available the following could be implemented.

8.3.1 Graphical User Interface (GUI)

Currently the GUI was adequate to do the job but maybe the GUI could have had a more appealing look and feel. This all beckons on whether or not Java was the best programming language to use to generate the best looking GUI. Maybe a web developed GUI could have been a better alternative but with the developer having little experience in web development this would not have been ideal.

8.3.2 Table Management

A feature that was thought of as a possibility but never documented past the design stage was the use of a table management feature. This would give the system the ability to reserve and allocate tables. The table data could then be used to help predict how busy the restaurant may be and help prepare the staff rota.

8.3.3 Cooking Instructions

Currently the kitchen display is not as functional as it could have been and could be developed to include the cooking instructions for each meal. Obviously the database schema does not currently support the storage of the cooking instructions so there are added implications. The time to cook the individual ingredients could also be included within this function helping the chef with the organisation of the cooking.

8.3.4 Online Management

Currently the statistics and data management are controlled from a computer within the restaurant premises. The generation of a management web application, could allow access to the statistics and data from anywhere in the world thus allowing the managers to keep check on the business.

8.4 Reflection

On reflection, even though the majority of the proposed features were completed and the project was deemed a huge success, the author felt that he could have been more disciplined in keeping to the plan. He also felt that the proposed features were slightly unrealistic and some even unnecessary.

For the general project, the author felt that important aspects of research were not undertaken including interviews with restaurant owners and user questionnaires. This would have provided good insight into existing solutions.

8.5 Skills Attained

This project has helped the author to attain new skills as well as develop existing skills. The skills attained have been both technical and individual with the main individual skill being project management which required good time keeping and management of the workload. Some technical skills that have been developed include:

- Advanced coding using the Java `Swing` interface.
- Relational database schema design and trigger coding.
- Advanced coding using Java threads.

8.6 Chapter Summary

This chapter has concluded the project report and provided an insight into possible future development.

Bibliography

[1] Ian Alexander. Computing & control engineering. Volume 14(Issue 1):22–26, April 2003.

[2] Ezee Burrp. *Software Features.* Accessed on 14 October, 2009. `http://www.ezeeburrp.com/features.html`.

[3] HSQL db. *Database comparison.* Accessed on 13 September, 2009. `http://hsqldb.org/images/imola_retrieve.jpg`.

[4] Business Dictionary. *Stakeholder.* Accessed on 2 April, 2010. `http://www.businessdictionary.com/definition/stakeholder.html`.

[5] Geoffrey Elliot. *Global business information technology: an integrated systems approach.* Pearson Education, 2004.

[6] Beefeater Grill. *Beefeater Grille.* Accessed on 24 September, 2009. `http://www.beefeater.co.uk/ourFoodAndDrink`.

[7] Horizon. *Restaurant Sector Overview.* Accessed on 4 May, 2010. `http://www.catererresearch.com/Articles/2006/05/12/306737/Restaurant-sector-overview.htm`.

[8] Java JDBC. *Java SE Technologies - Database.* Accessed on 3 May, 2010. `http://java.sun.com/javase/technologies/database/`.

[9] Paul Lemberg. *Which is Better: New customers or repeat business?.* Accessed on 12 October, 2009. `http://www.businessknowhow.com/marketing/new-customers.htm`.

[10] Agile Manifesto. *Principles behind Agile Manifesto.* Accessed on 20 September, 2009. `http://www.agilemanifesto.org/principles.html`.

[11] Craig Murphy. *Improving Application Quality Using Test-Driven Development (TDD).* Accessed on 16 October, 2009. `http://www.methodsandtools.com/archive/archive.php?id=20`.

[12] A Nutt. History of pos equipment. *Computers and Internet community*, 1:1, March 2009.

[13] Point of Success. *Products and Features.* Accessed on 14 October, 2009. `http://www.pointofsuccess.com/softwarefeatures.htm`.

[14] Abacre POS. *Major Features.* Accessed on 14 October, 2009. `http://www.abacre.com/restaurantpos/features.htm`.

[15] Steve Rosenberg. *Fast food, German-style.* Accessed on 15 October, 2009. `http://news.bbc.co.uk/1/hi/7335351.stm`.

[16] s Baggers restaurant. *Rollercoaster style restaurant.* Accessed on 13 October, 2009. `http://www.sbaggers.de/main-ger/?sid=home&lang=en`.

[17] Horsforth School. *Normalisation.* Accessed on 1 December, 2009. `http://www.horsforth.leeds.sch.uk/subjects/comp/alevel/module5/unit59/unit59d.asp`.

[18] Restaurant Pos Software. *HOSPOS.* Accessed on 14 October, 2009. `http://www.restaurant-pos-software.com/Products/HospitalitySoftware.aspx`.

[19] Don Wells. *Unit Tests.* Accessed on 24 September, 2009. `http://www.extremeprogramming.org/rules/unittests.html`.

List of Figures

List of Tables

List of Listings

Appendix A

Initial Project Plan

For my third year project I have to design a restaurant management system. This will involve designing a system for the restaurant, kitchen, and bar.

A.1 Aims & Objectives

The aims of the restaurant management system will be to:-

1. Increase efficiency of the restaurant by decreasing process time. This will speed up table turning which will result in an increase in profit.
2. Simplify communication between the restaurant and kitchen staff. All orders taken by the staff will result in a request on the monitors in the kitchen with all orders being separated by table and cooking time.
3. Provide planning for the future. Ingredient usage will be taken into account providing approximation into ingredient demand. This will help keep stock levels to the minimum increasing the average freshness of the produce.
4. Ease of setup for the end user – adding ingredients, costs, menus etc to the system.
5. Ease of figure generating: revenue, turnover, profit, and operating costs.
6. Design a user friendly front end whilst still allowing all combinations of options and extras for every meal.
7.

A.2 Proposed System Features

The proposed system features can be split into two categories: essential features and wish-list.
The essential features:-

1. The system will require a very simple user interface with touch screen capability.
2. Colour co-ordinated menu is a necessity. Advanced options incorporated into the simple front end to reduce the potential limitations. Allow options for every meal including extras that can be added at an extra cost.

3. User credentials to track the transaction history and average table turning time of every member in the team.
4. SQL Server/Oracle relational database to hold all the ingredients and courses with a hierarchy so ingredients can be allocated into food types. Creating food types will simplify the front end with respect to showing extras and options of a meal.
5. Administrator options for voiding and applying discounts
6. Management features for viewing sales report, recipe costing etc.
7. Booking tool for allocating tables with table turn estimation.

The wish-list:-

1. Bluetooth PDA's for the restaurant staff so the orders will be immediately sent to the kitchen staff decreasing process time.
2. User training built into the system.
3. Table management map – Easy allocation of staff to tables and the progress of the current customers.

A.3 Task, Milestone and Deliverable Summary

The Gantt chart has been allocated two different design and implementation phases to accommodate exam revision. The structure and the design of the database and the class diagram of the system will need to be complete before the first design phase can begin. The first major task will come in the first design phase. The first design phase will look into designing the hierarchy of the ingredients and how they will be displayed within the system.

The second design phase will start at the beginning of week 45. This is mainly down to the two deadlines in week 46 and 47. I have decided to give myself plenty of time to practice the seminar due to lack of confidence in presenting slides. The first main implementation stage will begin once the design is complete. Obviously there could be some design tweaks whilst in the implementation or testing stage and this will have to be accommodated when the issue arises.

The second implementation phase will begin after the exams at the beginning of the second semester. This will lead right up to the end with the final 2 weeks overlapping with testing.

I have included all the deliverables within the Gantt chart and the hours required to prepare for each deliverable.

Appendix B

Use Case Scenarios

The following tables document the use cases from Figure 3.1.

Use case ID:	UC2
Use case name:	Order Food/Drink
Description:	Order creation and food/drink selection.
Pre conditions:	UC1.
Standard flow:	1. Order object created. 2. Item selected from the menu in the order system implementing UC3. 3. Order confirmed and sent to kitchen application.
Post conditions:	Stock used deducted from the database.

Table B.1: Use case 2: Order Food/Drink.

Use case ID:	UC3
Use case name:	Check/Deduct Stock
Description:	Method to deduct stock on selection of menu item.
Pre conditions:	Implemented by UC2.
Standard flow:	1. Stock checked before any item selected; options disabled if empty. 2. Item selected from the menu in the order system.
Post conditions:	Stock used deducted from the database.

Table B.2: Use case 3: Check/Deduct Stock.

Use case ID:	UC4
Use case name:	Prepare Drink
Description:	Drink prepared by bar staff.
Pre conditions:	UC3.
Standard flow:	1. Order retrieved. 2. Drinks prepared.
Post conditions:	Server called; implementing UC6.

Table B.3: Use case 4: Prepare Drink.

Use case ID:	UC5
Use case name:	Prepare Food
Description:	Food prepared by Chef.
Pre conditions:	UC3.
Standard flow:	1. Order retrieved. 2. Chef confirms order has started. 3. Food is cooked.
Post conditions:	Server called; implementing UC6.

Table B.4: Use case 5: Prepare Food.

Use case ID:	UC6
Use case name:	Call Server
Description:	Server called when food or drink is complete.
Pre conditions:	Implemented by UC4 and UC5.
Standard flow:	1. Server called to inform either food or drink is ready.
Post conditions:	Server collects.

Table B.5: Use case 6: Call Server.

Use case ID:	UC7
Use case name:	Serve Food/Drink
Description:	Server serves food/drink.
Pre conditions:	UC6.
Standard flow:	1. Server serves food/drink once waiter called.
Post conditions:	No post conditions.

Table B.6: Use case 7: Serve Food/Drink.

Use case ID:	UC8
Use case name:	Eat Food
Description:	Customer eats the food.
Pre conditions:	No pre conditions.
Standard flow:	1. Customer eats/drinks the food/drink.
Post conditions:	No post conditions.

Table B.7: Use case 8: Eat Food.

Use case ID:	UC9
Use case name:	Pay
Description:	Customer pays for the food.
Pre conditions:	UC10.
Standard flow:	1. Order completed. 2. Price calculated. 3. Discounted price calculated by implementing UC10. 4. Customer pays.
Post conditions:	No post conditions.

Table B.8: Use case 9: Pay.

Use case ID:	UC10
Use case name:	Check Offers
Description:	Method to check all active offers generating best price.
Pre conditions:	No pre conditions.
Standard flow:	1. Generate list of active offers. 2. Apply algorithm to calculate best price.
Post conditions:	UC9.

Table B.9: Use case 10: Check Offers.

Appendix C

Database Structure

Due to the size of the database (amount of tables), the following diagram only shows the table names and relationships.

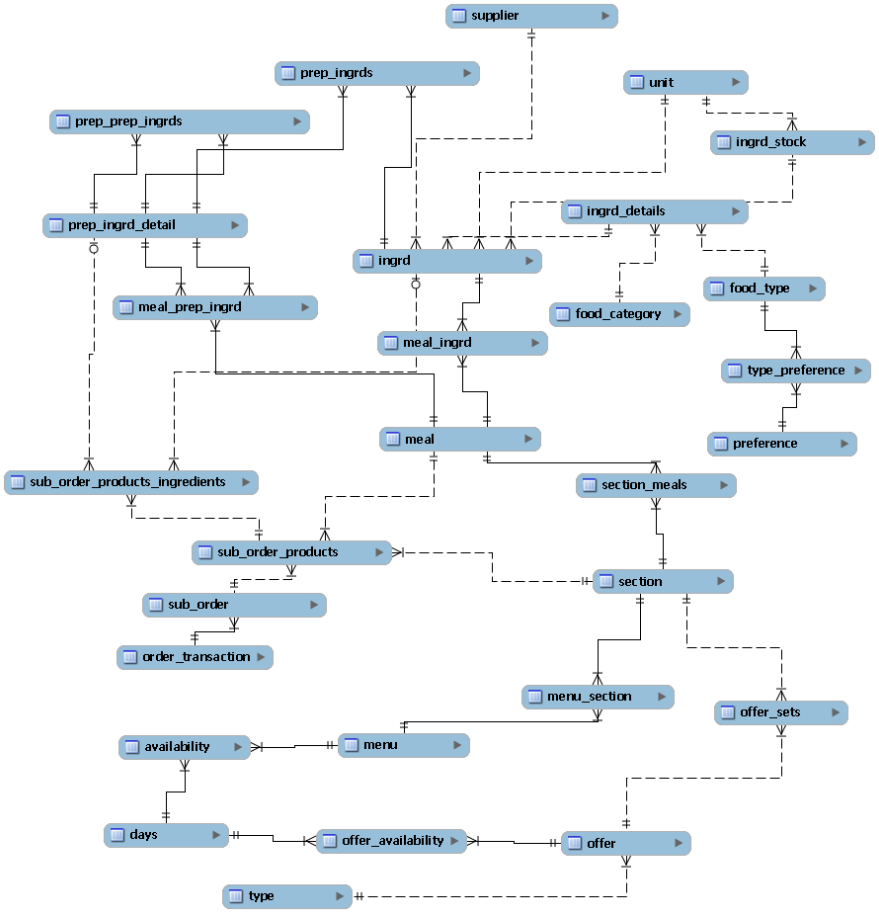

Figure C.1: MySQL database structure.

Appendix D

Database Stock Control Functions

Listing D.1: Database function for decreasing ingredient stock.

```
CREATE PROCEDURE `reduceIngredientStock` (IN ingredientID int, IN subOrderProductID int)
BEGIN
  DECLARE amount double;
  DECLARE stockID int;

  if(subOrderProductID = 0) then
    SELECT DISTINCT ingrd.Default_Measurement, Ingrd_Stock_ID into amount, stockID
    FROM ingrd
        WHERE ingredient_id = ingredientID;
  else
    /* Get the manual measurement and ingredient stock ID */
    SELECT DISTINCT meal_ingrd.Manual_Measurement, ingrd.Ingrd_Stock_ID into amount, stockID
    FROM (sub_order_products
      INNER JOIN meal_ingrd ON sub_order_products.Product_ID = meal_ingrd.Product_ID)
      INNER JOIN ingrd ON meal_ingrd.Ingredient_ID = ingrd.Ingredient_ID
    WHERE (((sub_order_products.Sub_Order_Product_ID)=subOrderProductID)
      AND ((meal_ingrd.Ingredient_ID)=ingredientID));
  end if;
  /* Subtract the amount from the ingredient stock table */
  UPDATE ingrd_stock
  SET current_amount = current_amount - amount
  WHERE ingrd_Stock_ID = stockID;
END
```

Listing D.2: Database function for updating the stock field within the meal table.

```
CREATE PROCEDURE `restmngsys`.`updateAvailableStockMealOnly` ()
BEGIN
  /* Update all meals to initial value of 4: Stock level ok! */
  UPDATE meal set available_stock = 4;
  /* Update meals that are optional that are low to stock to 3 */
  Update meal set available_stock = 3 where product_id IN
  (
    SELECT DISTINCT section_meals.Product_ID
    FROM ((((section_meals
      LEFT JOIN meal_ingrd ON section_meals.Product_ID = meal_ingrd.Product_ID)
      LEFT JOIN meal_prep_ingrd ON section_meals.Product_ID = meal_prep_ingrd.Product_ID)
      LEFT JOIN ingrd ON meal_ingrd.Ingredient_ID = ingrd.Ingredient_ID)
      LEFT JOIN prep_ingrd_detail
        ON meal_prep_ingrd.Ingredient_Prepared_ID = prep_ingrd_detail.Prepared_Ingredient_ID)
      LEFT JOIN ingrd_stock ON ingrd.Ingrd_Stock_ID = ingrd_stock.Ingrd_Stock_ID
    WHERE (((meal_ingrd.Optional)=True)
        AND ((meal_ingrd.Manual_Measurement)<=ingrd_Stock.current_amount)
        AND ((ingrd_stock.Current_Amount)<=ingrd_Stock.min_Threshold))
      OR (((meal_prep_ingrd.Optional)=True)
```

```
        AND ((prep_ingrd_detail.Available_Stock)=1))  /* 1: Stock level low */
  );
  /* Update meals that are optional that are out of stock to 2 */
  Update meal set available_stock = 2 where product_id IN
  (
    SELECT DISTINCT section_meals.Product_ID
    FROM ((((section_meals
      LEFT JOIN meal_ingrd ON section_meals.Product_ID = meal_ingrd.Product_ID)
      LEFT JOIN meal_prep_ingrd ON section_meals.Product_ID = meal_prep_ingrd.Product_ID)
      LEFT JOIN ingrd ON meal_ingrd.Ingredient_ID = ingrd.Ingredient_ID)
      LEFT JOIN prep_ingrd_detail
        ON meal_prep_ingrd.Ingredient_Prepared_ID = prep_ingrd_detail.Prepared_Ingredient_ID)
      LEFT JOIN ingrd_stock ON ingrd.Ingrd_Stock_ID = ingrd_stock.Ingrd_Stock_ID
    WHERE (((meal_ingrd.Optional)=True)
        AND ((meal_ingrd.Manual_Measurement)>=ingrd_Stock.current_amount))
      OR (((meal_prep_ingrd.Optional)=True)
        AND ((prep_ingrd_detail.Available_Stock)=0))
  );
  /* Update meals that are compulsory that are low to stock to 1 */
  Update meal set available_stock = 1 where product_id IN
  (
    SELECT DISTINCT section_meals.Product_ID
    FROM ((((section_meals
      LEFT JOIN meal_ingrd ON section_meals.Product_ID = meal_ingrd.Product_ID)
      LEFT JOIN meal_prep_ingrd ON section_meals.Product_ID = meal_prep_ingrd.Product_ID)
      LEFT JOIN ingrd ON meal_ingrd.Ingredient_ID = ingrd.Ingredient_ID)
      LEFT JOIN prep_ingrd_detail
        ON meal_prep_ingrd.Ingredient_Prepared_ID = prep_ingrd_detail.Prepared_Ingredient_ID)
      LEFT JOIN ingrd_stock ON ingrd.Ingrd_Stock_ID = ingrd_stock.Ingrd_Stock_ID
        WHERE (((meal_ingrd.Optional)=False)
                AND ((meal_ingrd.Manual_Measurement)<=ingrd_Stock.current_amount)
                AND ((ingrd_stock.Current_Amount)<ingrd_stock.min_threshold))
            OR (((meal_prep_ingrd.Optional)=False)
                AND ((prep_ingrd_detail.Available_Stock)=1))
  );
  /* Update meals that are compulsory that are out of stock to 0 */
  Update meal set available_stock = 0 where product_id IN
  (
    SELECT DISTINCT section_meals.Product_ID
    FROM ((((section_meals
      LEFT JOIN meal_ingrd ON section_meals.Product_ID = meal_ingrd.Product_ID)
      LEFT JOIN meal_prep_ingrd ON section_meals.Product_ID = meal_prep_ingrd.Product_ID)
      LEFT JOIN ingrd ON meal_ingrd.Ingredient_ID = ingrd.Ingredient_ID)
      LEFT JOIN prep_ingrd_detail
        ON meal_prep_ingrd.Ingredient_Prepared_ID = prep_ingrd_detail.Prepared_Ingredient_ID)
      LEFT JOIN ingrd_stock ON ingrd.Ingrd_Stock_ID = ingrd_stock.Ingrd_Stock_ID
    WHERE (((meal_ingrd.Optional)=False)
        AND ((meal_ingrd.Manual_Measurement)>=ingrd_stock.current_amount))
      OR (((meal_prep_ingrd.Optional)=False)
        AND ((prep_ingrd_detail.Available_Stock)=0))
  );
END
```

Appendix E

Pricing Algorithm

Listing E.1: Recursive pricing algorithm function.

```
public void recursiveMethod(OfferObject [] offerArray, int [][] product){
    // looping through each offer
    int [][] nextProduct;
    for (int index = 0; index<offerArray.length; index++){
        // Apply Offer: Returns new set as array
        nextProduct = offerArray[index].applyOffer(product);
        if(nextProduct != null)
            // Recursive call
            recursiveMethod(offerArray, nextProduct);
        else{
            // End of path, compare cheapest price
            cheapestPrice(product);
        } // else
    } // for
} // recursiveMethod
```

Listing E.2: Initial function to create required variables for the recursive function (Listing E.1).

```
public double calculateDiscountedPrice(int aOrderID,java.util.Date orderDate){
  // VARIABLES DECLARED: REMOVED TO REDUCE SIZE
  try{
    db = new Database();
    conn = db.getConnection();
    // NOW CALCULATE REDUCTION
    // First find all the available offers
    //  -   Checking the order date and availability of the offer
    //  -   Checking to see that at least 1 product exists in offer set from order
    availableOffers = db.executeSelectQuery("" +
      "SELECT DISTINCT offer.Offer_ID " +
      "FROM (sub_order_products " +
        "INNER JOIN sub_order ON sub_order_products.Sub_Order_ID = sub_order.Sub_Order_ID) " +
        "INNER JOIN (((offer INNER JOIN offer_availability " +
          "ON offer.Offer_ID = offer_availability.Offer_ID) " +
        "INNER JOIN offer_sets ON offer.Offer_ID = offer_sets.Offer_ID) " +
        "INNER JOIN section_meals ON offer_sets.Set_ID = section_meals.Section_ID) " +
          "ON sub_order_products.Product_ID = section_meals.Product_ID " +
      "WHERE (((offer_availability.From_time)<='"+orderDate+"') " +
        "AND ((offer_availability.To_time)>='"+orderDate+"') " +
        "AND ((offer_availability.Day_ID)=DayOfWeek('"+orderDate+"')) " +
        "AND ((sub_order.Order_ID)="+aOrderID+"))");
    // Now create the offer objects
    // 2 types of offers    1)      Buy n from A and get m from B % off
    //                      2)      Buy 1 from A, n from B and m from C for £x
    int offerCount = 0;
    // Check to see how many offers there are
```

```
while (availableOffers.next()) offerCount++;
// Go back to beginning
availableOffers.beforeFirst();
// Return if no offers valid
if(offerCount == 0){
  ...
}else{
  // Get a list of all the sets in the particular offer id under the order
  pstmt = conn.prepareStatement("" +
    "SELECT DISTINCT type.Name, offer_sets.Set_ID " +
    "FROM (sub_order " +
      "INNER JOIN sub_order_products
        ON sub_order.Sub_Order_ID = sub_order_products.Sub_Order_ID) " +
      "INNER JOIN (section_meals " +
      "INNER JOIN ((offer " +
      "INNER JOIN type ON offer.Type_ID = type.Type_ID) " +
      "INNER JOIN offer_sets ON offer.Offer_ID = offer_sets.Offer_ID) " +
        "ON section_meals.Section_ID = offer_sets.Set_ID) " +
        "ON sub_order_products.Product_ID = section_meals.Product_ID " +
    "WHERE (((sub_order.Order_ID)="+aOrderID+") AND ((offer_sets.Offer_ID)=?))");

  // An array of offers to apply
  //(The offer count will be the max number of children for each node)
  OfferObject [] offerArray = new OfferObject[offerCount];
  // Set initial set count to 0
  int setCount = 0;
  // Set initial offer Number to 0
  int offerNumber = 0;
  // Loop through each offer 1 by 1
  int currentOfferID = 0;
  offer = new Offer();
  while (availableOffers.next()){
    // Clear parameters in prepared statement
    pstmt.clearParameters();
    // Set first parameter to the offer id
    currentOfferID = availableOffers.getInt("Offer_ID");
    offer.getOffer(currentOfferID);
    pstmt.setInt(1, currentOfferID);
    rsetOffer = pstmt.executeQuery();
    setCount = 0;
    while (rsetOffer.next()) setCount++;
      rsetOffer.beforeFirst();
      rsetOffer.next();
      if (rsetOffer.getString("Name").compareTo("Offer1") == 0 && setCount == 1){
        int setID1 = rsetOffer.getInt("Set_ID");
        // Example: Buy 1 get 1 free
        offerArray[offerNumber] = new Offer1Object(aOrderID,setID1,setID1,
          offer.getNSet1(),offer.getNSet2(),offer.getPercent(),currentOfferID);
        offerNumber++;
      }else if(rsetOffer.getString("Name").compareTo("Offer1") == 0 && setCount == 2){
        ... // SIMILAR TO FIRST IF
      }else if(rsetOffer.getString("Name").compareTo("Offer2") == 0 & setCount == 1){
        ... // SIMILAR TO FIRST IF
      }else if(rsetOffer.getString("Name").compareTo("Offer2") == 0 & setCount == 2){
        ... // SIMILAR TO FIRST IF
      }else if(rsetOffer.getString("Name").compareTo("Offer2") == 0 & setCount == 3){
        ... // SIMILAR TO FIRST IF
      }
      //offerNumber++;
    } // while
    // Once while loop complete
    // Have an arry of Offers need to apply them offers in numerous ways
    orderRset = db.executeSelectQuery("" +
    "SELECT Product_ID,Sub_Order_Product_ID, price " +
      "FROM sub_order INNER JOIN sub_order_products
        ON sub_order.Sub_Order_ID = sub_order_products.Sub_Order_ID " +
      "WHERE (((sub_order.Order_ID)="+aOrderID+")) " +
      "ORDER BY sub_order_products.price DESC");
      int count = 0;
    while (orderRset.next()) count++;
    orderRset.beforeFirst();
    // Each product with price
    // The extra one will be the price
    int [][] product = new int [count+1][3];
```

```
        int index = 0;
        while(orderRset.next()){
          product[index][0] = orderRset.getInt("Product_ID");
          product[index][1] = (int)(orderRset.getDouble("Price")*100.0);
          product[index][2] = orderRset.getInt("Sub_Order_Product_ID");
          index++;
        } // while
        product[product.length-1][0] = 0;product[product.length-1][1] = 0;
        discountedPriceInt = 0;
        recursiveMethod(offerArray, product);
      } // while
      return ((double)discountedPriceInt/100.0);
  }catch (Exception e){
      throw e;
  }finally{
      ...
  } // finally
}
```