Application and Evaluation of Use Case Maps with UML in Object-Oriented Systems Development by Case Study

By

William S Blackley

A project presented to the
School of Computing
University of Paisley
for the degree of
MSc in Information Technology with Software Development

September 2005

Supervisor: Dr Ying Liang
Moderator: Mr Michael McCready

Contents

Acknowledgement

I wish to thank Dr Ying Liang, Mr Michael McCready and Mr Alistair McMonnies (all School of Computing, University of Paisley, Scotland) for their guidance and support throughout this project.

I would also like to express my gratitude to Dr Daniel Amyot (School of Information Technology and Engineering (SITE), University of Ottawa, Canada) for his guidance; in particular, the information he provided on the history of Use Case Maps.

I am very much obliged to Mrs C Baldwin, Senior Systems Analyst, HSBC Bank plc, Dr Bill Kay, Principle Consultant, Financial Services, LogicaCMG and Mr Mike Costello, Head of Corporate Systems, Abbey National plc for their comments.

Many thanks, to my friend Mr David MacIntosh of Newell and Budge Ltd, for his encouragement and invaluable advice regarding the general architecture of a bank accounts management system.

I also wish to thank all my other friends for the encouragement they gave me while I was undertaking my MSc.

Lastly, I would like to express my deepest gratitude to my mother, Margaret, without whose faith and support I would never have had the opportunity to undertake my MSc.

Abstract

The project evaluates the benefits of adding a Use Case Map view to existing Unified Modeling Language views of an object-oriented software system. This includes ascertaining if adding a Use Case Map view to existing Unified Modeling Language views helps bridge the gap between requirements and use cases and more detailed views, such as interaction diagrams (sequence diagrams and collaboration diagrams) and statechart diagrams. Also, a comparison of Use Case Maps and Use Case Diagrams is given.

.

In order to do this, a prototype of an application for managing a hypothetical bank's customers' accounts is developed. The application is to be used by the bank's branch staff and is connected to the bank's database. A prototype of the database is also developed.

Chapter 1

Introduction

1.1 History

Use Case Maps (UCMs) started life in the late 1980's/early 1990's. At this time they were called *Program Slices* and the first formal publication which used them was Dr Mark Vigder's Ph.D. Thesis[1]. Over the next few years Professor Raymond J A Buhr and his colleagues (in particular Professor Murray C Woodside) at Carleton University in Ottawa, Canada developed the ideas into a notation called *Timethreads*, which focused on high-level design of real-time and distributed systems. Dr Daniel Amyot (School of Information Technology and Engineering (SITE), University of Ottawa, Canada) studied and formalized Timethreads (using LOTOS) in his MSc Thesis[2]. Timethreads were renamed UCMs around 1995 when Professor Buhr and R S Casselman wrote the UCMs book [18].

Unified Modeling Language (UML) was developed in order to try and consolidate and simplify the many object-oriented development methods which had emerged since the first object-oriented methods were published in the late 1980's. The *Object Modelling Technique* (OMT) was developed by James Rumbaugh et al at General Electric in 1991. One year later (1992) the *Objectory* book[3] was published by Ivar Jacobson et al and in 1994 Grady Booch et al suggested an approach which he calls *round-trip gestalt design.* Rumbaugh joined Booch at Rational Software Corporation in 1994 and this led to the first successful attempt to combine and replace existing development methods. They began integrating the concepts from OMT and round-trip gestalt design and this resulted in a first proposal in 1995. Ivar Jacobson joined Rational Software Corporation that year (1995) and began working with Booch and Rumbaugh. Their joint work was called the Unified Modeling Language (UML). A request for proposals for a standard approach to object-oriented modelling was issued by the Object Management Group (OMG) in 1996. A number of proposals were developed and eventually they all came together in the final UML proposal which was submitted to the OMG in September 1997. In November 1997 the membership of the OMG unanimously adopted the UML as a standard and the OMG assumed responsibility for the further development of the UML standard.

The relationship between UCMs and UML has been considered in, for example, [2], [3], [5] and [16]. However, at the end of 1999, Dr Daniel Amyot and his colleagues, together with a team at Nortel (who were involved in ITU-T/TIA[4] standardization activities) initiated the idea of creating a standard for UCMs. The ITU-T accepted the idea as part of a new "question for study" on *User Requirements Notation* (URN) in 2000. According to Dr Amyot, their work in this area has progressed slowly since then, due to the loss of many industrial partners and contributors from the telecommunications sector. However, at the time of writing, they do expect to have the UCMs specification (ITU-T Recommendation No. Z.152, [25]) approved within a year. All this has led to little work being done on the interesting potential of UCMs being added to UML.

1.2 Overview

1.2.1 UCMs Overview

UCMs help a person describe and understand the large-grained behaviour patterns of complex and dynamic (computer) systems. In this context, systems are understood to mean

---

[1] Vigder, M, (1992) *Applying Formal Techniques to the Design of Concurrent Systems.* Ph.D. Thesis, Carleton University, Canada.

[2] Amyot, D, (1994) *Formalization of Timethreads Using LOTOS.* MSc Thesis, Dept. of Computer Science, University of Ottawa, Canada.

[3] Jacobson, I, et al, *Object-Oriented Software Engineering: A Use Case Driven Approach.* Wokingham: Addison-Wesley, 1992.

[4] ITU-T:- International Telecommunications Union Standardization Sector, TIA:- Telecommunications Industry Association (USA)

sets of collaborating components which work together to achieve some overall objective. UCMs are composed of causal paths called *use case paths* (shown as sets of wiggly lines) which traverse organisational structures of abstract *components*. Both use case paths and components may have *responsibilities* (there will always be at least one responsibility in any particular diagram, although it may not be visible). Responsibilities are short, prose descriptions of actions, activities, operations, etc that the system must perform and are represented by crosses in the diagrams (*dynamic responsibilities* are represented differently – see section 2.1). The term components means software entities such as objects, processes, databases, servers, etc. as well as non-software entities such as actors. The use case paths are said to be "causal" because they link causes (e.g. a mouse click) to effects (e.g. a window closing) and because they involve concurrency and partial orderings of activities. They have start points (filled circles representing preconditions or triggering causes) and end points (bars representing post-conditions or resulting effects). The use case paths are visual representations of *scenarios*. A scenario is a sequence of actions that a system performs which achieves a notable result of value to a particular user (actor). If one moves along a use case path, point-to-point, from its start to its finish, the path traced is a scenario. This interpretation of use case paths links UCMs to *use cases.* A use case is a set of prose descriptions of scenarios of an actor's interaction with a system [18]. The basic structure of a UCM is shown in Figure 1.1 below (adapted from diagram in [35]).

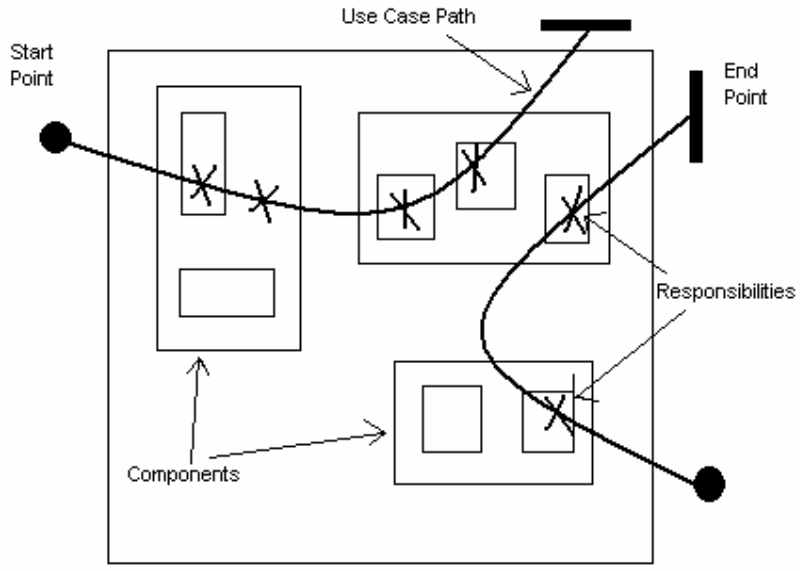


Figure 1.1: Basic Structure of UCMs

UCMs can be used for requirements engineering, design, testing, maintenance, adaptation and evolution. They have already been used in a wide variety of areas such as requirements engineering and design (of real-time systems, object-oriented systems, telecommunications systems, etc.), functional testing, performance analysis and prediction, and synthesis of Message Sequence Charts (MSCs) and formal specifications, to name but a few.

1.2.2 UML Overview

UML is a non-proprietary, general-purpose, visual modelling language which is used for specifying, visualising and documenting the structure and design of software systems. It is not a programming language. UML is general-purpose in that it can model most application domains, including those that involve large, complex, real-time, distributed, etc, systems. It is both independent of implementation language and platform independent. UML is not a complete development method. However, it does support most, if not all, of the existing development processes (lifecycles). One must be aware of the fact that UML is distinct from any particular development process (lifecycle).

UML can also be used to model non-software systems e.g. workflow in an office. However, it should be noted that UML is a discrete modelling language and is therefore not suitable for modelling the continuous systems that one finds in areas such as physics, engineering, etc. It is intended for modelling software-intensive systems and has been successfully used for this in market sectors such as Banking and Financial Services, Telecommunications, Transport, Defence/Aerospace and Retail, to name but a few.

UML is composed of structural, behavioural, grouping and annotational *things*, *relationships* and *diagrams* which group these things and relationships together to show a particular aspect (known as a *view*) of the system being modelled. There are nine types of diagram in UML which between them show five different *views* of the system being modelled.

1.3 Project Objectives

The aim of the project is to answer the following two questions:

- *What are the differences between Use Case Maps and Use Case Diagrams ?;* and

- *Are there significant benefits to be gained by adding a Use Case Map view to existing Unified Modeling Language views of a software system ? This includes ascertaining if adding a Use Case Map view to existing Unified Modeling Language views of a software system helps bridge the gap between requirements and use cases and more detailed views, e.g. interaction diagrams.*

In order to answer these questions a prototype for a software system for managing a hypothetical clearing bank's customers' accounts will be developed. The hypothetical clearing bank is called "*BestBank*" and the software system is for managing the accounts of a new product (type of bank account) which *BestBank* is introducing called the "*FlexiPlus*" account – a current account. The software system will comprise of an object-oriented application at the front-end connected to an operational database (cf. data warehouse) at the back-end. The software development lifecycle used will be based on the traditional *Waterfall* lifecycle model.

Initially, an outline of how to construct UCMs and UML diagrams will be given, along with a synopsis of the literature regarding the possible extension of UML with UCMs. Then, a detailed (prose) specification for a software system to manage *BestBank's FlexiPlus* accounts will be given. This detailed specification is contained in Chapter 3 – The Case Study. The application is called "*FlexiPlus* Accounts Management System" or FAMS for short. The database is called FAMSDB. Based on the detailed specification of the system, three separate analysis & design phases will be carried out. The analysis & design of the FAMSDB database will be carried out in the first phase. The second phase will be an analysis & design of the FAMS application using a number of UML views of the system. The third phase will be an analysis & design of the FAMS application using UCM views of the system as well as the UML views of the system used in the second analysis & design phase. Having carried out the analysis & design stages of the development lifecycle, a prototype of the FAMSDB database will be implemented using Microsoft Jet 4.0 and Microsoft Access 2002 and a prototype of the FAMS application, based on the third analysis & design (UML views and UCM views), will be implemented using Microsoft's Visual Basic .NET. This will be followed by an outline of a testing strategy for the FAMS prototype. All of this work will then be used to carry out an evaluation which will:-

- contrast UCMs and UCDs
- outline the benefits (if any) gained by adding a UCM view to the analysis & design, by reflecting upon how easy it would have been to build the FAMS prototype using the first analysis & design (UML views only) rather than the second analysis & design (UCM view and UML views)
- decide whether or not adding a UCM view helps bridge the gap which separates requirements and use cases and more detailed views.
- consider the possibility of extending UML with UCM concepts.
- critically review the work carried out in the project.

Finally, there will be a brief conclusion of the project's findings and recommendations for further work will be given, if appropriate.

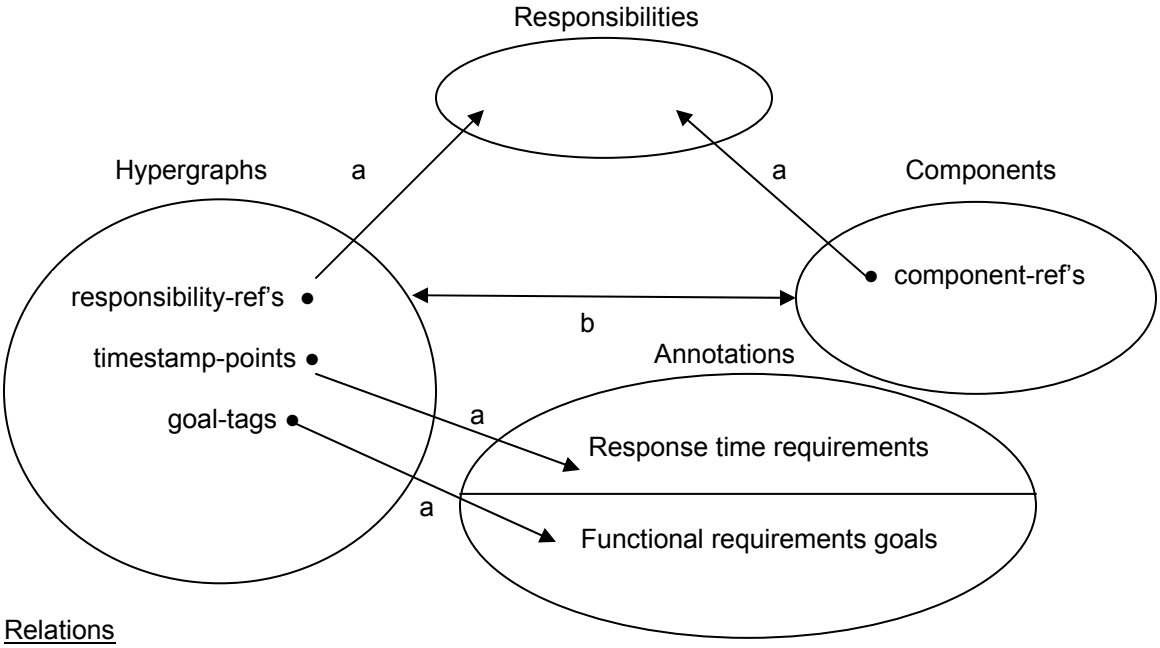
Chapter 2

Use Case Maps and the Unified Modeling Language

2.1 Vocabulary of UCMs - Definitions

All of the definitions in this section are taken or adapted from those given in [17] [18] [25].

The vocabulary (or elements) of the UCM notation is grouped into four types, namely, *hypergraph* elements (called *hyperedges*), *component* elements, *responsibility* elements and *annotational* elements. A hypergraph is a graph structure that represents use case paths (causal scenarios) and specifies the hyperedges that make up the use case paths. There are five types of component element, namely, *teams, processes, objects, agents* and *pools.* Regular components (teams, processes, objects and agents) have *slot, protected, replicated* and, optionally, *processor-id* attributes. Responsibilities are processing tasks that must be carried out by the system and can be referenced by components (via the *component-ref* attribute) and referenced by use case paths (via the *responsibility-ref* hyperedge). Responsibilities may be defined as being dynamic. There are two types of annotational element, namely, *response time requirements* and *functional requirements goals*. Annotations are referenced by use case paths: response time requirements are referenced by two *timestamp-point* hyperedges and functional requirements goals are referenced by two *goal-tags* hyperedges. Every UCM element has an XML[5] definition [25], however, this aspect of UCMs will not be covered in this paper. Most, but not all, UCM elements have a graphical representation. The relationships between hypergraphs, components, responsibilities and annotations are shown if Figure 2.1 below.



Relations
a – reference, eg. responsibility-ref's *reference* Responsibilities
b – superposition, i.e. Hypergraphs (use case paths) can be superposed on Components, and vice versa.

Figure 2.1: Relations on UCM notation

[5] XML:- World Wide Web Consortium's *eXtensible Markup Language*

2.1.1 Use Case Path Elements (and Annotational Elements)

UCM hypergraphs contain the basic use case path constructs (hyperedges), which are outlined below.

Start Points

Start points (or start hyperedges) are where scenarios are caused by one or more triggering events and/or one or more pre-conditions being satisfied. Each start point has a list of triggering events and/or list of pre-conditions associated with it and has an appropriate label. A start hyperedge has no source hyperedge (preceding hyperedge), except when it's triggered by the end point, or an empty segment, of another use case path, and can have only one target hyperedge (subsequent hyperedge). See Figure 2.2 below.

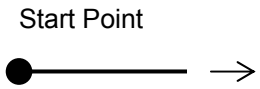


Figure 2.2: Start Point

End Points

End points (or end hyperedges) describe the effects of a scenario. Each end point has a list of resulting events and/or post-conditions associated with it and an appropriate label. An end point can have only one source hyperedge (preceding hyperedge) and no target hyperedges (subsequent hyperedges), except when it's triggering the start point, a waiting place or a timer of another use case path. See Figure 2.3 below.

Figure 2.3: End Point

Empty Segments

Empty segments (see Figure 2.4(a) below) are used to add characteristics to use case paths such as *failure points* where the causal flows stops for some reason (see Figure 2.4(b) below), *shared responsibilities* which indicate that there must be some negotiation between two or more components in order for the (shared) responsibility to be completed (see Figure 2.4(c) and Figure 2.4(d) below), the triggering of another use case path by the empty segment connecting with a start point (see Figure 2.4(e) below) or to indicate the direction of the causal flow of the use case path by superimposing an arrow on it (see Figure 2.4(f) below).

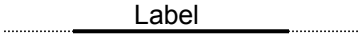


Figure 2.4(a): Empty Segment

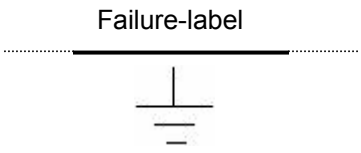


Figure 2.4(b): Empty Segment with Failure Point

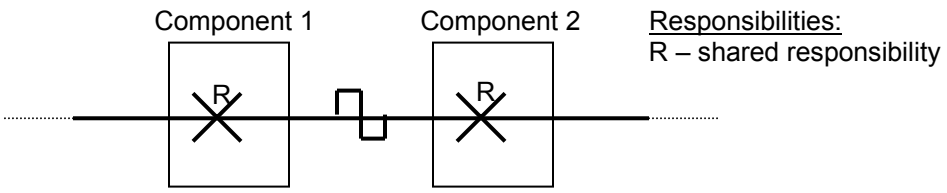


Figure 2.4(c): Empty Segment with (two component) shared responsibility

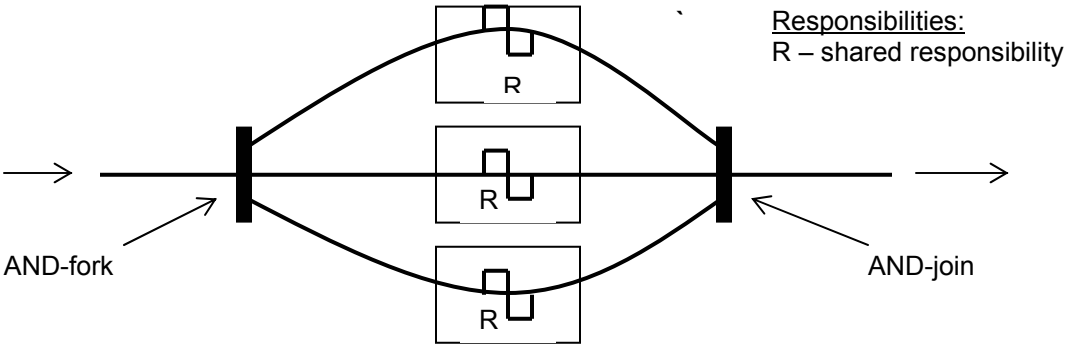


Figure 2.4(d): Empty Segment with (multi-component) shared responsibility

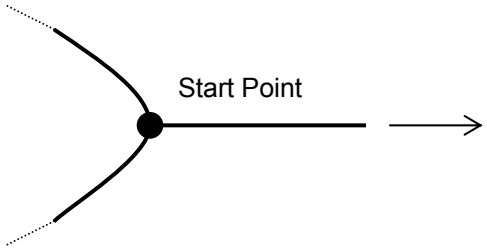


Figure 2.4(e): Empty Segment triggering another Use Case Path

Figure 2.4(f): Empty Segment with path direction shown

Waiting Places and Timers

Waiting places are points on a use case path where the causal flow stops until it is restarted by some triggering event, or the satisfaction of some precondition, that emanates from the environment or from another use case path via a connected end point or connected empty segment. Waiting places have appropriate labels. See Figure 2.5(a) below.

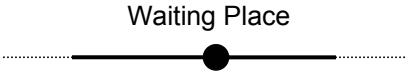


Figure 2.5(a): Waiting Place

Timers are specialised types of waiting place where the causal flow stops and only continues if the triggering event, or satisfaction of a precondition, occurs within a certain period of time. If the event does not occur in the specified time the causal flow continues along a timeout path (an abort hyperedge). See Figure 2.5(b) below.

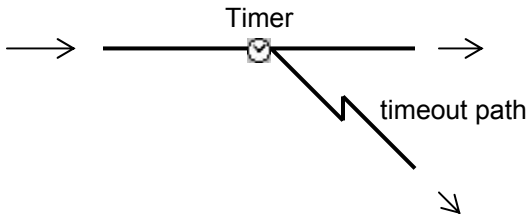


Figure 2.5(b): Timer

OR-forks and OR-joins
OR-forks occur when a use case path splits into two or more alternative paths. Thus, an OR-fork has only one source hyperedge and two, or more, target hyperedges. OR-forks may have an appropriate label. (See Figure 2.6(a) below). OR-joins denote the merging of two or more use case paths. Thus, an OR-join will have two or more source hyperedges but only one target hyperedge. OR-join may have a label. (See Figure 2.6(b) below).

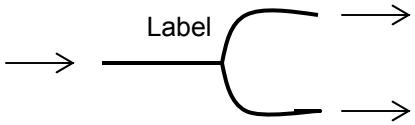


Figure 2.6(a): OR-fork

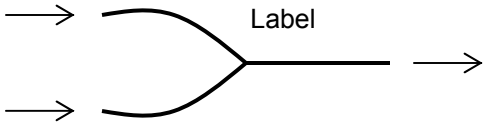


Figure 2.6(b): OR-join

AND-forks and AND-joins (synchronizations)
There are two types of synchronization – AND-forks and AND-joins. AND-forks occur when a use case path splits into two or more concurrent use case paths. AND-forks have one or more source use case path(s) and one or more target use case path(s). See Figure 2.7(a) below. AND-joins occur when two or more use case paths are synchronised. AND-joins can have two or more source use case path(s) and one or more target use case path(s). See Figure 2.7(b) below. Synchronizations can have a cardinality associated with them which specifies the number of source use case paths and the number of target use case paths. Cardinalities are given in the form N:M, where N = number of source use case paths and M = number of target use case paths. AND-forks must have cardinality such that N < M. AND-joins must have cardinality such that N > M.

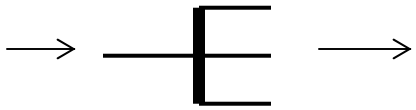


Figure 2.7(a): AND-fork with cardinality 1:3

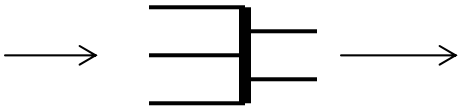


Figure 2.7(b): AND-join with cardinality 3:2

Loops
Loops represent repetition. They have two source hyperedges representing the original use case

path and the end of the loop. They have two target hyperedges representing the original use case path and the beginning of the loop. See Figure 2.8 below.

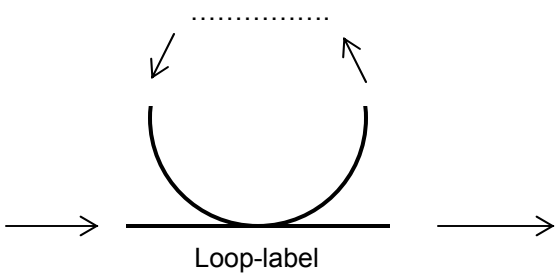


Figure 2.8: Loop

Aborts
Aborts are used to represent the case where one use case path aborts or disables the flow of another use case path. A timer's timeout path (see above) is a type of abort. Aborts can also be used to represent exception-handling in systems. An abort has one source hyperedge and one target hyperedge. An abort can link an empty segment or stub to an other empty segment or stub. See Figure 2.9 below.

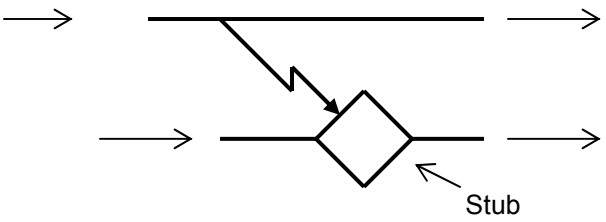


Figure 2.9: Abort (empty segment linked to static stub)

Stubs and Plug-ins
Stubs are containers for plug-ins and allow fine-grained detail to be factored out of the diagram in which the stub appears. Plug-ins are sub-UCMs which describe how the parent use case path behaves. A plug-in could simply be superposed on its parent UCM at the appropriate point, however, this may cause the parent UCM to become very cluttered (busy) at that point.

There are two types of stub: *static stubs* and *dynamic stubs*. Static stubs have only one plug-in associated with them whereas dynamic stubs can have any number of plug-ins associated with them, the selection of which is determined at run-time. If a stub involves negotiations with components other than the one which it is superposed on (allocated to) it is called a *shared stub*. See Figure 2.10(a), Figure 2.10(b), Figure 2.10(c) and Figure 2.10(d) below.

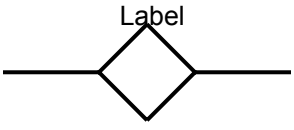


Figure 2.10(a): Static stub

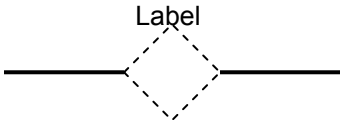


Figure 2.10(b): Dynamic stub

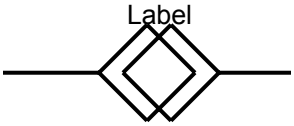


Figure 2.10(c): Static shared stub

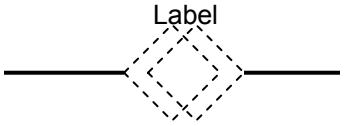


Figure 2.10(d): Dynamic shared stub

Stubs have entry points and exit points which are connected to source and target use case paths, respectively. The use case paths that are connected to the stub are bound to the use case paths of the associated plug-in(s). See Figure 2.10(e) below.

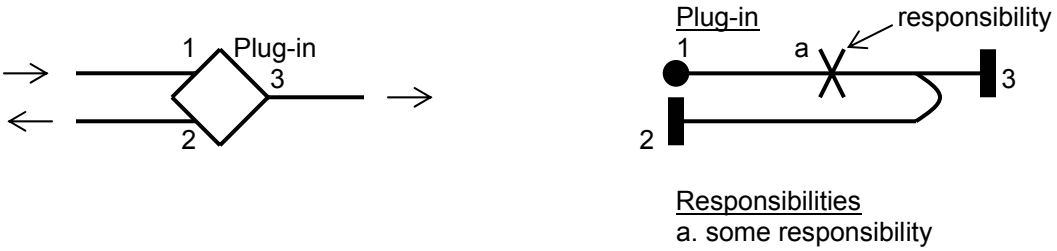


Figure 2.10(e): Static stub with associated plug-in

Responsibility References
A responsibility reference (responsibility-ref hyperedge) refers to a responsibility (either a non-dynamic responsibility or a dynamic responsibility). Thus, responsibility references allow responsibilities to be bound to use case paths. Use case paths which have one or more responsibilities bound to them and are not bound to a *component substrate* are called *unbound UCMs.* See section 2.1.3 below for the graphical notation of responsibility references (and for the graphical notation of the attribute component-ref, which also references responsibilities). Note: responsibilities themselves have no graphical notation.

Annotations
In a UCM, performance (response time) requirements and functional goals can be described with annotations.

A response time requirement is referenced by two timestamp-point hyperedges; one indicating the start of the response time requirement and one indicating the end of the response time requirement. The response time requirement attribute *response-time* specifies the time allowed (in μs) for the system to perform all the procedures, operations, tasks, etc between the two timestamp-points for a certain percentage of the responses. The See Figure 2.11(a) below.

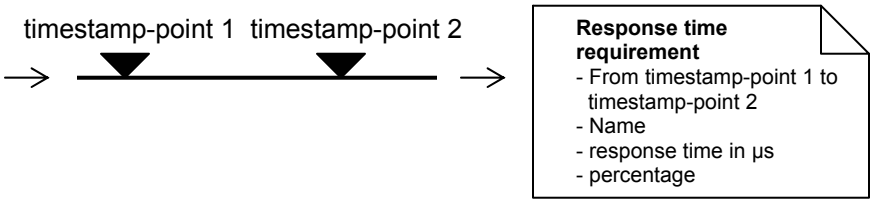


Figure 2.11(a): Response time requirement annotation

Functional requirements goals occur in agent systems. They are referenced by two goal-tag hyperedges; a starting goal-tag and an ending goal-tag. The goal-tags indicate that the procedures, operations, tasks, etc between the start goal-tag and the end goal-tag are intended to achieve the specified functional goal. See Figure 2.11(b) below.

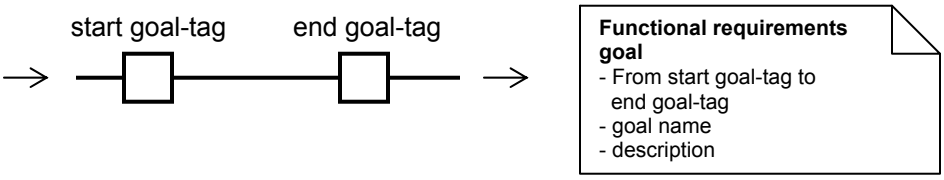


Figure 2.11(b): Functional requirements goal annotation

2.1.2 Component Elements

There are two categories of components: *regular components* and *pools*. Regular components may have both *dynamic responsibilities* and *non-dynamic responsibilities* bound to them. Pools may have only dynamic responsibilities bound to them. (See section 2.1.3.) Organizational structures of components are called component substrates. There are four types of Regular component, as follows:

Teams

A team is coarse-grained component used as a container for sub-components. That is, a team is a grouping of sub-components. The sub-components can be of any type, i.e. they can be objects, processes, agents, pools or even other teams. Teams are non-specific components in that they do not specify any type and give no indication of whether they will later be expanded into a group of sub-components or if they will later be replaced by a single component of a specific type. In this way teams allow for the consideration of details to be deferred until later in the design process. A team that contains multiple processes is said to be an active team; otherwise, it is said to be a passive team. See Figure 2.12 below.

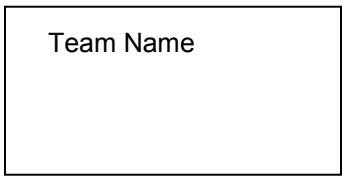


Figure 2.12: Team

Processes

Processes are coarse-grained, autonomous, self-directed, active components that can work concurrently with other processes in active teams. Active teams are teams that contain one or more processes; passive teams contain no processes. Since processes are coarse-grained they may contain other components. However, the internal logic of a process is sequential and so they may not contain other processes, i.e. a process may only contain objects and passive teams. Ultimately, processes control when passive components, such as objects, perform their responsibilities. A process is similar to the mainline programme (eg. main() method in Java console applications, Sub Main() in VB.NET console applications) in sequential programmes. See Figure 2.13 below.

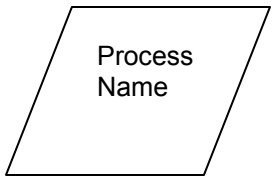


Figure 2.13: Process

Objects

An object is a passive, fine-grained component that supports a data and/or procedural abstraction through an interface. The actual interfaces of objects are not specified in UCMs. However, it is assumed that an object's interface will allow processes, teams and other objects to control when it performs its responsibilities. Since objects are fine-grained, they cannot contain other components, i.e. objects cannot be decomposed to show finer detail. If an "object" requires decomposition, i.e. if it needs to contain other components, a team should be used in place of it. Objects may represent instances of classes in an object-oriented class hierarchy. See Figure 2.14 below.

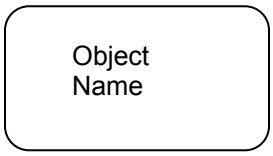


Figure 2.14: Object

Agents
An agent is an autonomous component, which acts on behalf of other components. An agent is a stand-alone computer program that performs tasks on behalf of a controlling entity, but without direct, continuous supervision. The agent may utilize network resources and services, but works relatively independently of other components. See Figure 2.15 below.

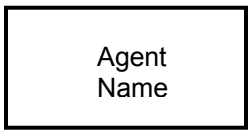


Figure 2.15: Agent

All of the aforementioned regular components possess the following Boolean attributes:

Slots
When “true”, the slot attribute indicates that the component is a static holding-place for active (executing) *dynamic components* of its type. That is, at any particular moment, a slot may be empty or populated by one active dynamic component, of the specified type, which will perform the slot’s responsibilities. Hence, when dynamic components are in slots they are visible for operational purposes.

Dynamic components are never explicitly shown in UCMs. They are components which are held in pools (see below) when they are passive (non-executing) and moved into slots when they are active (executing). When a slot is populated by a dynamic component it is assumed that the dynamic component can perform the slot’s responsibilities.

Slot components are shown with a dotted boarder. See Figure 2.16(a) and Figure 2.16(b) below.

Figure 2.16: Slot for holding dynamic teams

Figure 2.16(b): Slot for holding dynamic objects

Protected
When “true”, the protected attribute indicates that the component has mutual exclusion protection, i.e. the component’s responsibilities must not be performed concurrently by processes. The protected attribute deals with the issue of multiple use case paths traversing passive teams and objects (both of which are not capable of concurrent processing). Note that processes always

have mutual exclusion protection by definition and so their protected attribute is always set to "false" – there is no need to indicate mutual exclusion protection as it is implicit. In regular components other than processes, mutual exclusion protection is indicated by double outlines. See Figure 2.17 below.

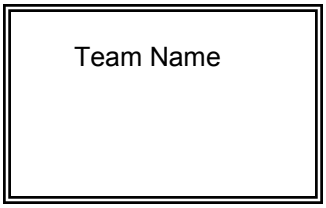


Figure 2.17: Team with mutual exclusion protection

Replicated
When "true", the replicated attribute indicates that multiple instances of the component exist simultaneously. This is shown graphically as a component stack. See Figure 2.18 below.

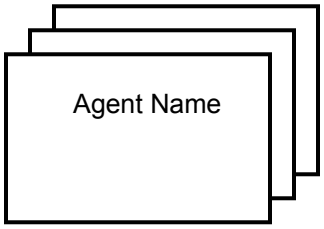


Figure 2.18: Multiple instances of an agent

Processor-id
When "true", this (optional) attribute refers to the processor on which the component executes and can be used for performance analysis. The processor-id attribute appears in components' XML definitions but has no graphical notation.

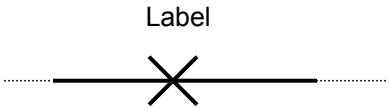
Pools
Pools are places for holding any number of passive (non-executing) dynamic components, until they are required to be moved into slots to perform the slots' responsibilities. They can also hold a list of plug-ins until they are required to be moved into dynamic stubs or dynamic shared stubs. Pools can have dynamic responsibilities bound to them but can not have non-dynamic responsibilities bound to them. See Figure 2.19 below.

Figure 2.19: Pool

2.1.3 Responsibility Elements
Every UCM has at least one responsibility element, however, it may not be visible in the diagram. There are two categories of responsibilities; *non-dynamic responsibilities* and *dynamic responsibilities*. Responsibilities are bound to use case paths by responsibility-ref hyperedges. That is, responsibility-ref hyperedges reference responsibilities. Responsibilities are bound to components by their component-ref attributes. That is, components' component-ref attributes

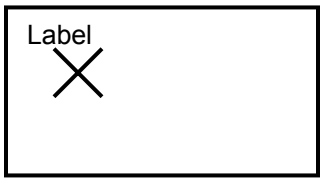
reference responsibilities. Pools may have dynamic responsibilities bound to them but not non-dynamic (designer defined) responsibilities.

Non-dynamic responsibilities are designer defined, i.e. they are short, prose descriptions of processing tasks, eg. actions, activities, operations, etc, that the designer specifies the system must perform. Non-dynamic responsibilities are shown on UCMs as crosses with the responsibility's name (label). The short, prose description of the processing task associated with the cross and label will appear in a list of responsibilities on the UCM or in the UCMs documentation. See Figure 2.20(a) and Figure 2.20(b) below.



Responsibilities
Label. Description of processing task

Figure 2.20(a): Non-dynamic responsibility bound to a use case path



Responsibilities
Label. Description of processing task

Figure 2.20(b): Non-dynamic responsibility bound to an agent

Dynamic responsibilities enable dynamic components to be created, destroyed and moved in, along and out of, use case paths. They also enable plug-ins to be created, destroyed and moved in, along and out of, use case paths. There are five different actions that a dynamic responsibility can perform, as follows:

Create
Creates a new instance of a dynamic component, or a new instance of a plug-in, and moves it into or out of a use case path. In the case of new components, initialisation is assumed to be part of the responsibility. See Figure 2.20(c) below.

Figure 2.20(c): (dynamic responsibility) Create

Destroy
Moves an instance of a dynamic component, or an instance of a plug-in, into or out of a use case path and then deletes it. See Figure 2.20(d) below.

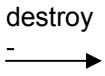


Figure 2.20(d): (dynamic responsibility) Destroy

Move
Moves a component instance, or a plug-in instance, into or out of a use case path. Used for unaliased moves, i.e. the source (use case path, slot, pool, dynamic stub or dynamic shared stub) does not retain a copy of the component instance or plug-in instance. See Figure 2.20(e) below.

Figure 2.20(e): (dynamic responsibility) Move

Move-stay
(Used for aliased moves.) Moves a reference to a component instance or a plug-in instance into or out of a use case path, i.e. the component instance or plug-in instance remains in the source (use case path, slot, pool, dynamic stub or dynamic shared stub) but is visible in the destination (use case path, slot, pool, dynamic stub or dynamic shared stub) via the reference. (This is similar to passing arguments by reference in C++ and VB.NET.) See Figure 2.20(f) below.

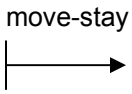


Figure 2.20(f): (dynamic responsibility) Move-stay

Copy
(Used for aliased moves.) Moves a copy of a component instance or plug-in instance into or out of a use case path, i.e. the component instance or plug-in instance remains in the source (use case path, slot, pool, dynamic stub or dynamic shared stub) while an exact replica of it appears in the destination (use case path, slot, pool, dynamic stub or dynamic shared stub). The two distinct copies then develop independently of one another. (This is similar to passing arguments by value in C++ and VB.NET.) See Figure 2.20(g) below.

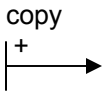
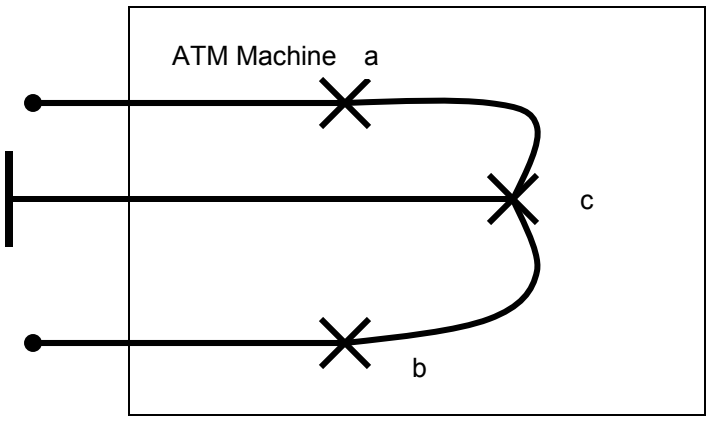


Figure 2.20(g): (dynamic responsibility) Copy

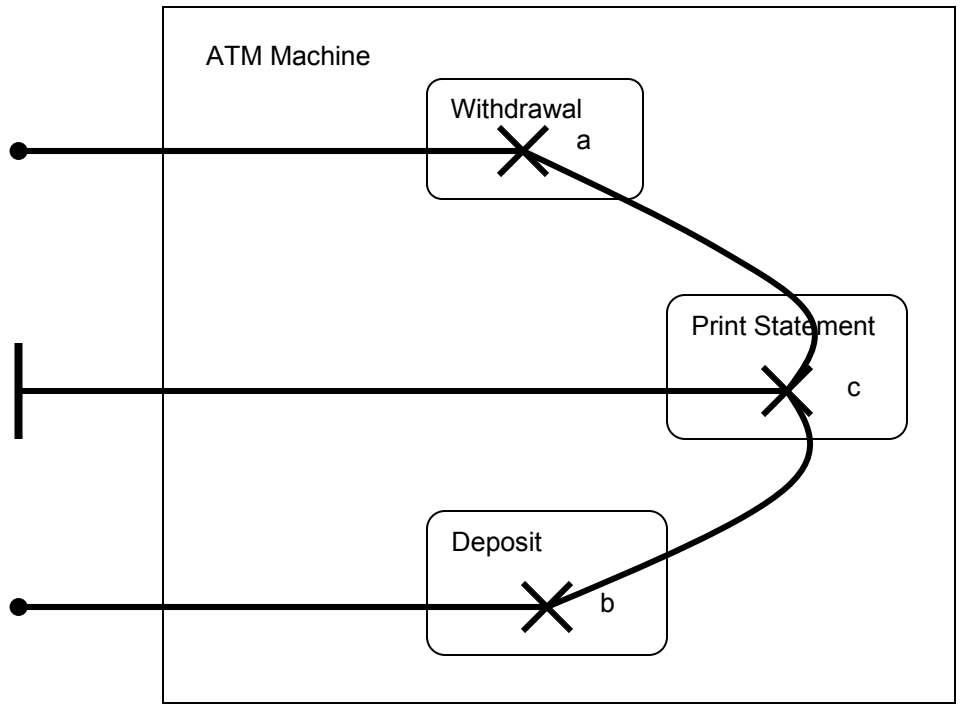
2.2 Construction of UCMs

UCMs may provide the designer with different views of the same system, each view showing a different level of granularity of the system. For example, the system as a whole may be represented by a single team only, with system level responsibilities bound to it, thus avoiding the detail. See Figure 2.21 below. Figure 2.21 shows an ATM Machine which allows the users to make withdrawals, make deposits and print a statement after they have made their deposit or withdrawal. This is a level 1 UCM. The level 2 UCM shown in Figure 2.22 then goes into more detail. It shows how responsibilities a, b and c are bound to withdrawal, deposit and print statement objects. The process of going into more and more detail can continue until the



Responsibilities
a. make withdrawal
b. make deposit
c. print statement

Figure 2.21: ATM Machine Level 1 UCM



Responsibilities
a. make withdrawal
b. make deposit
c. print statement

Figure 2.22: ATM Machine Level 2 UCM

developer has built up a satisfactory picture of the required system, with all its detail. This process is known as *recursive decomposition* [18].

A problem can arise when UCMs become very cluttered because of too many responsibilities attached to one object, for example. In this case a stub may be added to the diagram to factor out the cluttered part of the diagram into another UCM called a submap. See Figure 2.23 below, which shows an ATM Machine that allows users to post interest to their account. Depending on the balance of their account, the interest will be posted at a high rate, middle rate or low rate.

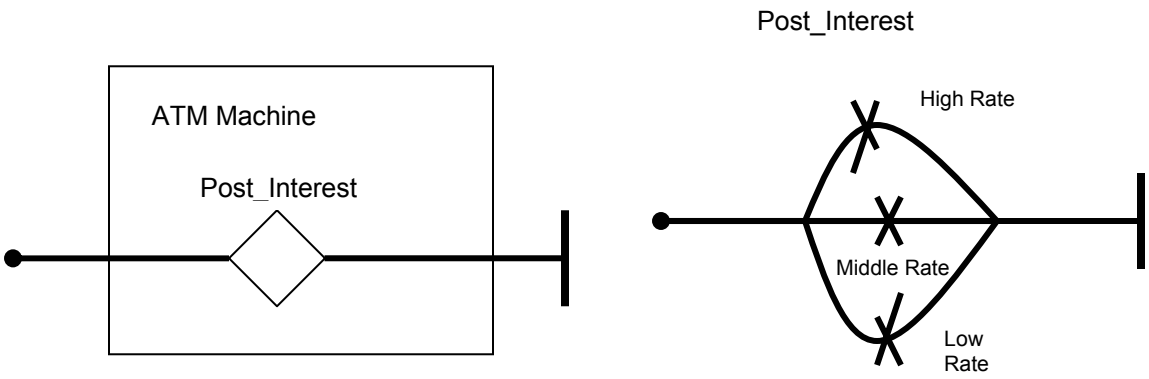


Figure 2.23: ATM Machine with Post Interest Function

Dynamic stubs show how behaviour patterns evolve at run-time since they allow for more than one submap to be associated with one (dynamic) stub. The context of the diagram determines which submap would be used for each situation.

Note that the submap in Figure 2.23 is not bound to a component. UCMs such as this are called unbound maps since the responsibilities in them are not bound to any system components such as objects. Similarly, UCMs can be constructed that have only components and responsibilities and no use case path(s) or UCMs can be constructed with only use case paths and components and no responsibilities. These UCMs are called "Incomplete UCMs". In [17], Buhr summarises this by saying "*Useful incomplete UCMs may be created by combining any pair of the three fundamental elements* [use case paths, components and responsibilities]."

2.3 Vocabulary of UML – Definitions

All of the definitions in this section are taken or adapted from those given in [12] [13] [14] [28] [32].

The vocabulary (or building blocks) of Unified Modeling Language is grouped into three types namely, *diagrams, things* and *relationships*. There are of nine types of diagram which group together the things and the relationships which show a particular aspect of the system being modelled. There are four categories of things in UML – *structural things, behavioural things, grouping things* and *annotational things*. Relationships bind these things together. The bond among diagrams, things and relationships on the UML notation is illustrated in Figure 2.24 below.

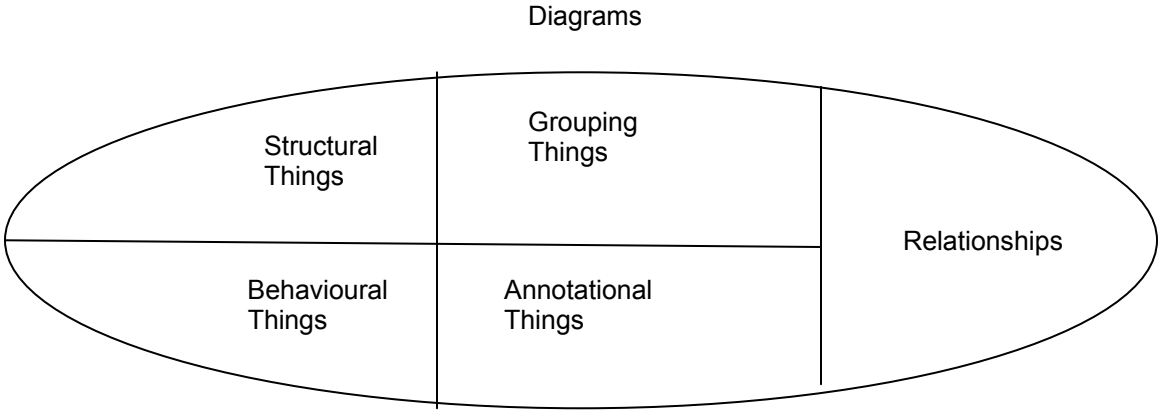


Figure 2.24: Bond among diagrams, things and relationships on UML notation

There are seven types of *structural things* in UML as follows:

Classes a class defines a set of objects which possess common attributes, operations, relationships and semantics. In diagrams, a class is represented by a rectangle which may include the name of the class, its attributes and its operations. See Figure 2.25 below.

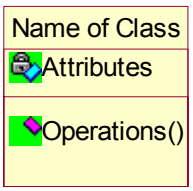


Figure 2.25: A Class

Interfaces an interface is a collection of operations which comprise the services that a class or component can provide. In diagrams, an interface is represented by a circle with the name of the interface. See Figure 2.26 below.

Figure 2.26: An Interface

Collaborations a collaboration defines an interaction comprising of roles and other elements which cooperate to provide some behaviour which is greater than the sum of all the elements. In diagrams, a collaboration is represented by an ellipse with dashed lines which usually only includes its name. See Figure 2.27 below.

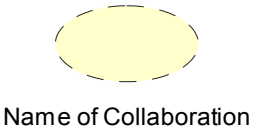


Figure 2.27: A Collaboration

Use case a use case is a prose description of a set of sequence of actions that a system performs that yields an observable result of value to a particular actor. In diagrams, a use case is represented by an ellipse with solid lines which usually only includes its name. See Figure 2.28 below.

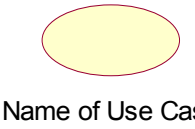


Figure 2.28: A Use Case

Active Classes active classes are classes which can initiate control activity since their objects own one or more processes or threads. In diagrams, an active class is represented in exactly the same way as a class but with thick surrounding lines. Note that the Rational Rose CASE tool does not indicate active classes with thick

surrounding lines but does indicate them by using bold font for the classes' name, attributes and operations. See Figure 2.29 below.

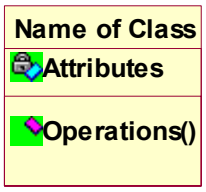


Figure 2.29: An Active Class

Components components are physical and replaceable parts of a system that conform to and provide the realization of a set of interfaces. In diagrams, a component is represented by a rectangle with tabs which usually only includes its name. See Figure 2.30 below.

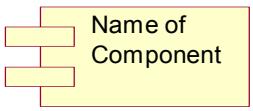


Figure 2.30: A Component

Nodes nodes are physical, run-time elements that represent computational resources which usually have some memory and processing capability. A node may house a set of components. In diagrams, a node is represented by a cube which usually only includes its name. See Figure 2.31 below.

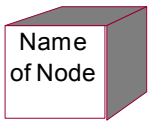


Figure 2.31: A Node

There are two types of *behavioural things* in UML as follows:

Interactions behaviours that comprise a set of messages exchanged between sets of objects in order to achieve a goal. In diagrams, a message is represented by a directed line labelled with the name of its operation. See Figure 2.32 below.

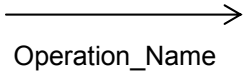


Figure 2.32: A Message

State Machines behaviours that specify the sequence of states that an object goes through during its life in response to events. A state machine has several elements such as *states,* (the interval between two events), *transitions* (change of state caused by an event), *events* (things that trigger transitions), *operations (actions* and *activities*[6] performed in response to a transition) and, possibly,

[6] *Activities* are ongoing nonatomic (i.e. interruptible) computations within state machines that ultimately result in some *action*. An *action* is an atomic (i.e. non-interruptible) computation within a state machine which ultimately results in a change in state of the system or the return of a value.

substates (states which are nested within the container or compound state). With the exception of the initial state and the final state (which are special cases), a state is represented in diagrams by a rounded rectangle which includes the name of the state and possibly may also include entry and exit actions, internal transitions, activities and deferred events. See Figure 2.33 below.

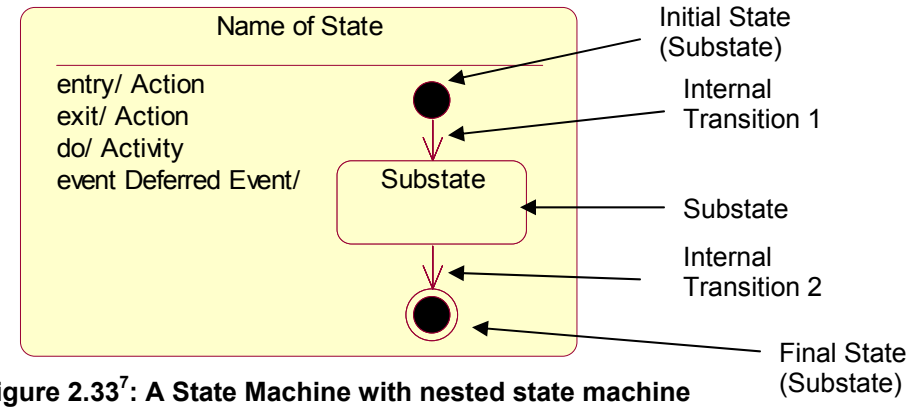


Figure 2.33[7]: A State Machine with nested state machine named Activity

There is one main type of *grouping thing* in UML as follows:

Packages ways for organising entities into groups. *Structural things, behavioural things* and other *grouping things* can be placed in a package. In diagrams, packages are represented by a tabbed folder which contains its name and sometimes its contents.

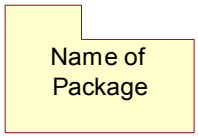


Figure 2.34: A Package

(Other types of *grouping things* such as *models* and *subsystems* are variations of *packages*.)

Annotational things are simply comments that can be applied to any element in a model in order to help describe it, remark upon it or explain it. In diagrams, a note which is represented by a rectangle with a dog-eared corner and some explanatory comment, or constraint, is appended to the model. See Figure 2.35 below.

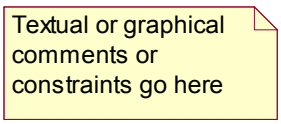


Figure 2.35: A Note

There are four types of *relationships* in UML as follows:

[7] Note that the text *Do/Activity* within the state machine is shorthand for the graphical representation of the nested state machine – normally there would be either the text or the graphical representation but not both, in the diagram.

Dependency a relationship between two things in which a change to one thing (the independent thing) may force a change in the other thing (the dependent thing). In diagrams, a dependency is represented by a dashed lined which may be directed and may have a label. See Figure 2.36 below.

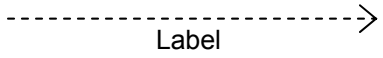


Figure 2.36: A Dependency

Association a structural relationship which describes a set of connections among objects. In diagrams, an association is represented by a solid line which may be directed, may have a label, and may show multiplicity and roles. See Figure 2.37 below.

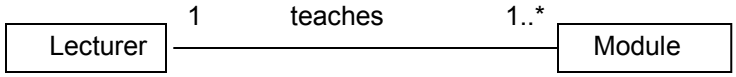


Figure 2.37: An Association

Figure 2.37 above shows an association (teaches) between Lecturers and Modules. The 1 (a multiplicity) at the Lecturer class end of the association indicates that every object of type Module is associated with one object of type Lecturer, i.e. every Module can be taught by only one Lecturer. The 1..* (also a multiplicity) at the Module class end of the association indicates that each Lecturer will teach one or more Modules. Multiplicities may be specified as:

- An exact number – simply by writing the number
- A range of numbers – lowest number in the range, followed by two dots, followed by largest number in the range
- An arbitrary, unspecified number – indicated by the * symbol.
- A comma separated list of multiplicities, e.g. 1, 10..20, 50..*

There are two special types of association, namely, *aggregation* and *composition. Aggregation* and *composition* are both “has” / “is part of” associations which indicate that an object of one class “has” / “is part of” an object of the other class. With *aggregation*, the component objects have independent existence from the compound object. With *composition,* the component objects only exist as long as the compound object exists. Multiplicity at the compound end of the composition is either 1 or 0..1. See Figure 2.38 and Figure 2.39 below.

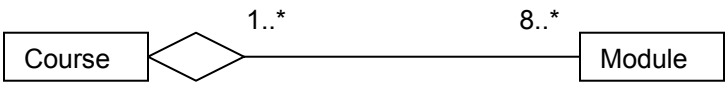


Figure 2.38: An Aggregation

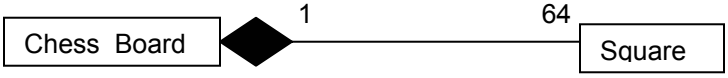


Figure 2.39: A Composition

Generalisation a *generalisation* is a “kind of” or “is a” relationship between a class (called a *superclass*) and one or more specialised versions of the *superclass* (called *subclasses).* A *subclass* inherits the characteristics of

its *superclass* (attributes, operations and relationships) as well as having its own specific attributes and operations. In diagrams, a generalisation relationship is represented by a solid line with a hollow arrowhead pointing to the *superclass.* See Figure 2.40 below.

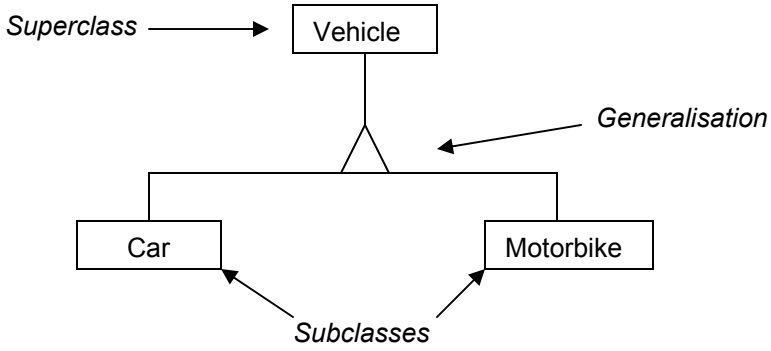


Figure 2.40: A Generalisation (Inheritance)

Realisations a *realisation* is a relationship between one classifier (which specifies a contract) and another classifier (which guarantees to carry out that contract). Classifiers describe structural and behavioural features and can include classes, interfaces, datatypes, signals, components, nodes, use cases and subsystems. *Realisations* occur between interfaces and the classes or components that realise them and between use cases and the collaborations that realise them. In diagrams, a *realisation* is represented by a directed dashed line with a hollow arrowhead pointing to the classifier that specifies the contract. See Figure 2.41 below.

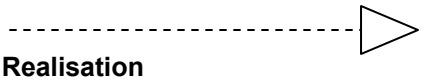


Figure 2.41: A Realisation

Please note that these are the four basic types of relationship that can be included in a UML model. However, as can be seen from the above explanation of association relationships, there are also variations on these.

There are nine diagrams in UML. These diagrams visualise a system from different perspectives. That is to say, they give different *views* of the system which is being modelled. Any particular element of the system being modelled may appear in all the diagrams, some of the diagrams or none of the diagrams. Generally, a particular element will appear in a few diagrams.

The nine diagrams contained in UML are as follows:

Class Diagram A class diagram is a set of classes, interfaces, collaborations and their relationships. They are used to model the static design view of a system, unless they contain *active classes,* in which case they model the static process view of a system. As with all other diagrams, a class diagram may contain notes and constraints. Also, they can contain packages or subsystems which are used to group elements of the diagram together.

Object Diagram An object diagram is a set of objects and their *links* at a particular moment in time. It is effectively an instance of a class diagram or the static part of an interaction diagram (sequence diagram or collaboration diagram). An object diagram is used to model the static design view or the static process view of a system. As with all other diagrams, an object diagram may contain notes and constraints. Also, they can contain packages or subsystems which are used to group elements of the diagram together.

Use Case Diagram A use case diagram (UCD) is a set of use cases, actors and their relationships. They are used to model the static use case view of a system. UCDs show the functional requirements of a system. As with all other diagrams, a UCD may contain notes and constraints. They can also contain packages which are used to group elements of the diagram together.

Sequence Diagram A sequence diagram is an interaction diagram which shows a set of objects and the messages that are exchanged between them. These messages are shown in time-ordered sequence. A sequence diagram is used to model the dynamic process view of a system. As with all other diagrams, a sequence diagrams may contain notes and constraints. A sequence diagram can be transformed into a collaboration diagram, and vice-versa.

Collaboration Diagram A collaboration diagram is an interaction diagram which shows a set of objects, the links between them and the messages that are exchanged between the set of objects. Collaboration diagrams show the organizational structure of the objects which exchange messages. A collaboration diagram is used to model the dynamic process view of a system. As with all other diagrams, a collaboration diagram may contain notes and constraints. A collaboration diagram can be transformed into a sequence diagram, and vice-versa.

Statechart Diagram A statechart diagram shows a state machine. They model the dynamic process view of a system. A statechart diagram is used to visualise the behaviour of interfaces, classes and collaborations by focusing on the event-ordered behaviour of objects. As with all other diagrams, a statechart diagram may contain notes and constraints.

Activity Diagram Activity diagrams are specialised statechart diagrams which visualise the flow of control among the objects of a system. They show a system moving from activity to activity as it functions. Activity diagrams model the dynamic process view of a system. As with all other diagrams, an activity diagram may contain notes and constraints.

Component Diagram A component diagram is a set of components and their relationships and, like a deployment diagram, it models the physical aspects of a system. For example, a component diagram may be used to model source code, executable releases, physical databases or adaptable systems. Component diagrams show the static implementation view of a system. As with all other diagrams, a component diagram may contain notes and constraints.

Deployment Diagram A deployment diagram shows a set of run-time processing nodes (and possibly the components housed on them) and their relationships. Deployment diagrams are used to visualise the configuration of these nodes and components. Like component diagrams, they are used to model the physical aspects of a system. Deployment diagrams show the static deployment view of a system. As with all other diagrams, a deployment diagram may contain notes and constraints.

UML can be used to model almost any type of application and can even be used to model non-software systems e.g. workflow in an office. However, it should be noted that UML is a discrete modelling language and is therefore not suitable for modelling the continuous systems that one finds in areas such as physics, engineering, etc. It is intended for modelling software-intensive

systems and has been successfully used for this in market sectors such as Banking and Financial Services, Telecommunications, Transport, Defence/Aerospace, Retail, to name but a few.

2.4 Construction of Use Case Diagrams

In [13], Booch, Jacobson and Rumbaugh say "*A use case diagram is a diagram that shows a set of use cases and actors and their relationships.*" Use case diagrams contain use cases, actors and dependency, generalization and association relationships as well as, possibly, containing notes and constraints.

In [14], Britton and Doake say "*A use case specifies a set of interactions between a user and the system to achieve a particular goal.*" Use cases represent the functional requirements of the system. The interactions are known as scenarios. A scenario is said to be an instance of a use case. The user (known as an actor) can be a human being that interacts with the system, an organization that interacts with the system or another automated system, such as another computer programme, that interacts with the system. Actors are the outside clients of the system.

Actors are represented diagrammatically by a stick man with an appropriate label. See Figure 2.42 below.

Figure 2.42: An Actor

Actors can be related to each other by associations and by generalizations and each actor is related to at least one use case by an association. Actors are related to use cases only by associations.

Use cases can be related to each other by generalizations and by dependencies. The dependencies are stereotyped into *include* (common behaviour) and *extend* (optional or exceptional behaviour) relationships. Include relationships occur where two or more use cases have common behaviour which is factored out into another use case. Extend relationships occur where a use case has optional or exceptional behaviour which can be factored out into another use case.

The method which one should employ in constructing a use case diagram is as follows:

1) Identify the actors (usually nouns in the *requirements specification*[8]) which need assistance from the system to achieve their goals.
2) Identify relationships (associations or generalizations) between the actors.
3) Label the actors with an appropriate stereotype.
4) Identify the functions that each actor requires of the system.
5) Name these functions as use cases.
6) Identify which use case(s) each actor interacts with.
7) Factor out common behaviour among the use cases into new use cases with include relationships
8) Factor out optional or exceptional behaviour among the use cases into new use cases with extend relationships

[8] *Requirements specification* refers to the structured or unstructured text which describes the system's design features, properties or behaviours which the system's actors rely upon.

9) Factor out specialization (inheritance) among the use cases into new use cases with generalization relationships
10) Add notes to the diagram if needed.

Example 2.1
Figure 2.43 below shows a possible use case diagram for the computer system of the MTI Retail Sales company.

MTI Retail Sales is a company that sells its products by mail order. It also takes telephone orders and orders via its website. Sales Clerks deal with the mail orders and the telephone orders. They also ship orders and partially completed orders to customers and send customers their bills when the system generates them. Customers may place orders directly, via MTI's website. In this case they have the option of paying immediately for their goods by credit card or by debit card. All orders are validated which involves checking the customers credit history and, in the case of credit and debit card payment, getting the transaction authorised by the relevant clearing house.

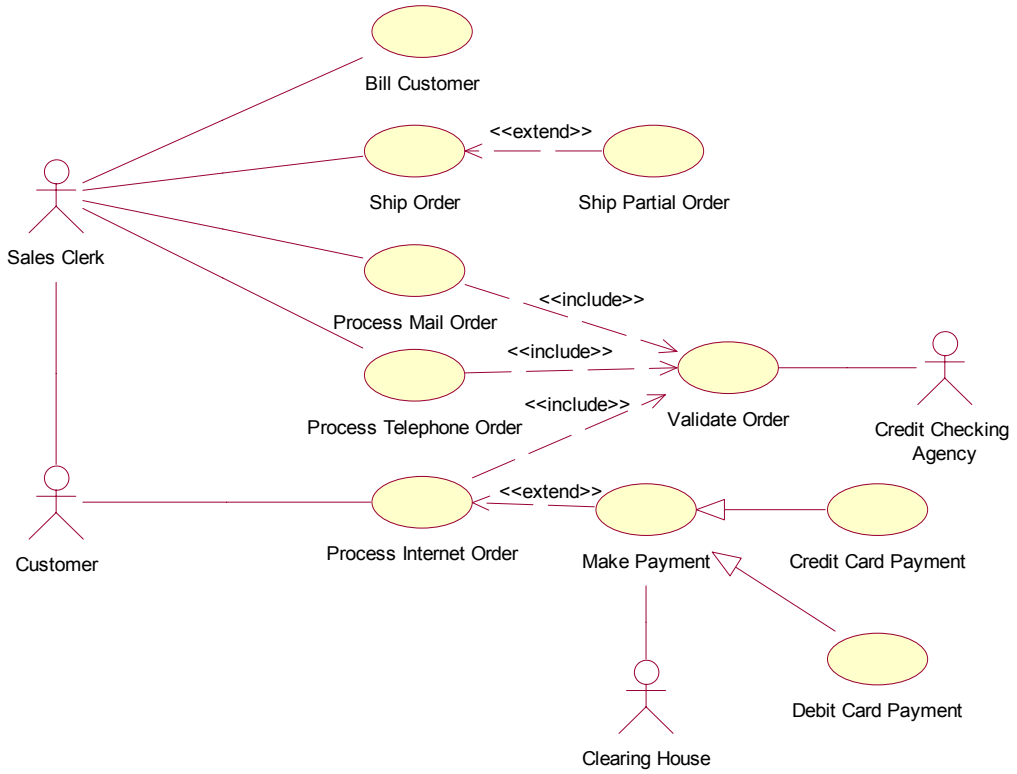


Figure 2.43: MTI Retail Sales Use Case Diagram

2.5 Construction of Class Diagrams

The principle modelling technique at each stage of the analysis and design of a piece of software is the class diagram.

In [14], Britton and Doake say "*The class diagram models the data elements in the system, the ways in which these may be grouped together, and the relationships between them.*"

When modelling the static design view of a system, class diagrams are commonly used to model the vocabulary of the system, simple collaborations and logical database schemas.

The method which one should employ in constructing a class diagram is as follows:

1) Identify objects and the classes from which they are derived.
2) Identify attributes (data elements) of the classes (see section 2.5.1).
3) Identify the operations belonging to each of the classes (see section 2.5.2).
4) Identify relationships (associations, aggregations, compositions and generalizations) between the classes (see section 2.5.3).
5) Repeat stages 1) to 4) in order to refine the model.

Every noun in the requirements specification should be noted as a potentially useful object which may, possibly, be generalized into a class. However, it is usually the case that not every object identified at this stage will become a class and, conversely, the class diagram may require classes which are not identifiable from the requirements specification. We can also identify classes from a data dictionary[9], which can be derived from the system's documents. We may discover that not all of the proposed classes perform a system function – in this case the class should be discarded.

Objects identified from the requirements specification should not be generalized into classes if they are:

1) Duplicates – two or more objects alias one another in that they are named differently but represent the same thing, e.g. *chair* and *seat* both refer to the same thing. In this case only one of the objects should be generalized into a class.
2) Roles – two or more objects alias one another in that they both refer to the same thing but have different names in different parts of the system, e.g. *electrician* is an alias for *employee*. In this case, we should retain the employee object and generalize it into a class because employee refers to the basic identity of the class. The electrician object should be discarded.
3) Irrelevant – reject objects which exist in the problem domain but are not part of the required system.
4) Vague – objects which are to be generalized into classes must have a precise meaning.
5) General – objects with too broad a meaning should be rejected.
6) Attributes – if the only operations we can identify for the proposed class are *Set* and *Get* (data) operations then it is likely that the object is an attribute (data element) of another class.
7) Associations – the nouns in the system requirements specification will sometimes represent relationships (associations, aggregations, compositions and generalizations) between other objects.

2.5.1 Attributes

The attributes (data elements) of the classes will usually be nouns in the system's requirements Specification, which have been rejected as candidates for classes. We may also identify attributes from a data dictionary, which can be derived from the system's documents.

When identifying attributes of the classes we should reject ones that are irrelevant to the specific problem, we should avoid storing unnecessary (redundant) information such as attributes that can be derived from other attributes. If we have the same attributes in more than one class this indicates that we may have one or more redundant class(es) which can be discarded or consolidated into a single class.

2.5.2 Operations (Methods)

In [28], McCready says "*….operations represent the services that the object* [instance of the class] *provides to users of the object (other connected objects).*"

[9] *Data dictionaries* are not covered in this paper. For a discussion of data dictionaries, see [14] and [28].

In [14], Britton and Doake say "*Operations may be identified from verb phrases in the problem description* [requirements specification]*……or from the presence of an attribute that needs to be updated* [Set method] *or read* [Get method]."

Generally, we do not add operations to the class diagram until we have constructed relevant sequence diagrams (and/or collaboration diagrams) and statechart diagrams, both of which assist us in identifying operations for the class(es). In particular, sequence diagrams (collaboration diagrams), which visualise the interaction between objects needed to achieve the goals of use cases, are a great help in identifying the operations of classes. Another very powerful way of establishing the operations of classes is to use the Class-Responsibility-Collaboration (CRC) cards technique[10]. A control class would also be added to the class diagram at this stage.

2.5.3 Relationships between Classes

In order for a software system to work, the objects within that system must collaborate with one another. In class diagrams we specify the static relationships between classes.

There are four types of relationship which can be modelled in a class diagram, namely, association, aggregation, composition and generalization (inheritance).

In [14], Britton and Doake say "*If we model a relationship between two classes, A and B, we are saying that all objects of class A potentially have a relationship with one or more objects of class B.*" In the case of association, aggregation and composition relationships, we may refine this further by showing multiplicities on the relationship (see section 2.3).

An association relationship between two classes indicates that there is some sort of relationship between objects of the two classes.

An aggregation relationship is a special type of association relationship where one class is made up of several others, or more than one object of another class, and the component object(s) has independent existence for the compound object.

A composition relationship is a special type of association relationship where one class is made up of several others, or more than one object of another class, and the component object(s) only exist(s) as long as the compound object exists.

An inheritance (generalization) relationship between two classes indicates that one class (the child class or subclass) is a more specialized version of the other class (the parent class or superclass). Subclasses inherit all the features (attributes and methods) of their superclasses, as well as having features of their own.

Relationships between classes can be identified from the system's requirements specification – usually verb phrases, e.g. "Mechanic *repairs* Car" indicates a relationship between the class Mechanic and the class Car. We may also identify relationships from a data dictionary, which can be derived from the system's documents, and from the use case diagram, e.g. a generalization relationship between two use cases may indicate an inheritance relationship between classes. Another very powerful way of establishing relationships between classes is to use the Class-Responsibility-Collaboration (CRC) cards technique.

Object diagrams (see section 2.6) are very useful for ascertaining the multiplicities of relationships.

[10] The CRC cards technique is not discussed in this paper. For a discussion of the CRC cards technique see [13] [14] [32].

Example 2.2
Figure 2.44[11] below shows a possible class diagram for the MTI Retail Sales system which was outlined in Example 2.1 on page 30. (Class methods are omitted in Figure 2.44.) Note that we have simply used common sense to determine the classes' attributes in Figure 2.44, rather than pulling them from the system's brief requirements specification.

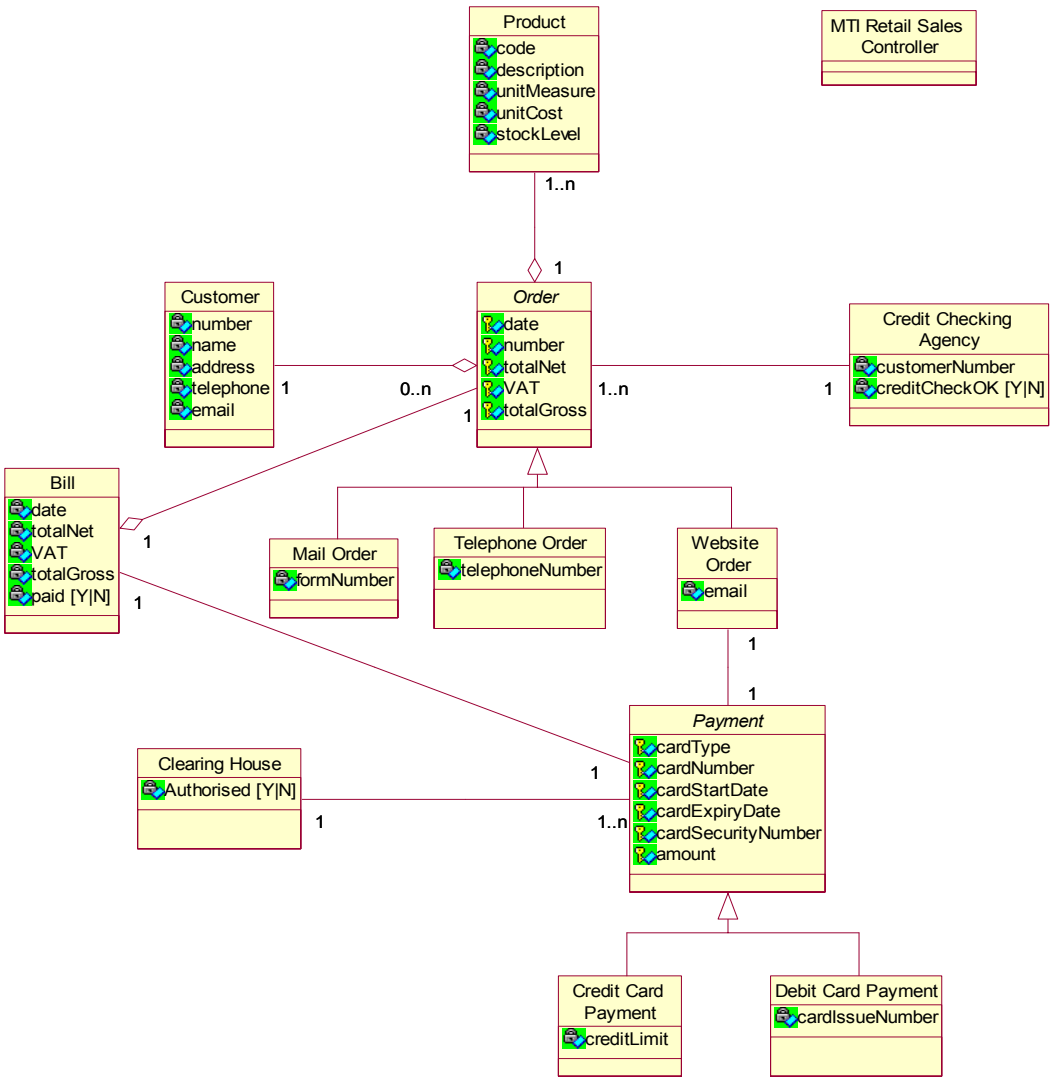


Figure 2.44: MTI Retail Sales Class Diagram

2.6 Construction of Object Diagrams

In [13], Booch, Jacobson and Rumbaugh say "*Object diagrams model the instances of things contained in class diagrams.*" They are used to show an instance of a class diagram or the static part of an interaction diagram (sequence diagram or collaboration diagram).

Object diagrams show sets of objects and their relationships at some specific point in time, i.e. an object diagram is a snapshot of the system being modelled, at some specific point in time, which

[11] Classes *Order* and *Payment* are abstract classes. This characteristic of a class is denoted in class diagrams by the class name being in italics.

shows a set of objects, their relationships and the state of the objects at that particular moment in time.

Object diagrams model the static design view or static process view of a system. In this context, they are usually used to model object structures. Here, object structures are understood to mean sets of concrete or prototypical objects.

Object diagrams contain *objects* and *links*. Objects are instances of classes. The relationships which connect objects in an object diagram are called links. Links are instances of the associations shown in the class diagram.

Object diagrams are not usually deliverables in the analysis and design of a software system but can be very useful in establishing the necessity and multiplicities of a relationship in the class diagram.

The method which one should employ in constructing an object diagram is as follows:

1) Identify the function or behaviour (of the system) that you want to model.
2) Identify the classes, interfaces, or other elements, and their relationships which collaborate to achieve the function or behaviour (of the system) which is being modelled.
3) Consider a scenario of the function or behaviour being modelled and freeze it at one particular moment in time. Show each object which is collaborating to achieve the function or behaviour at that moment in time.
4) Show the state and attribute values of each of these objects at the particular moment in time which is being considered.
5) Show the links between these objects.

Objects are shown in the object diagram as follows:

objectName = the name of the object (instance of the class). If the “objectName” is omitted the object is said to be a multi-object that consists of instances of the class “ClassName” which represent entities that have been identified but not yet assigned.
ClassName = the name of the class to which the object(s) belong(s).

Attributes are allocated values according to the objects state at the point in time being modelled. See Figure 2.45 below.

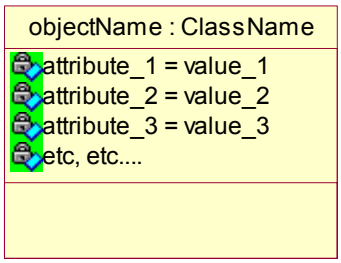


Figure 2.45: An Object from an Object Diagram

Example 2.3
Figure 2.46 below shows one possibly object diagram which corresponds to the class diagram in Example 2.2. Here we are modelling the state of the system when a mail order is being validated by a Credit Checking Agency. Note that the attribute “customerNumber” in the class “Credit Checking Agency” aliases the attribute “number” in the class “Customer”.

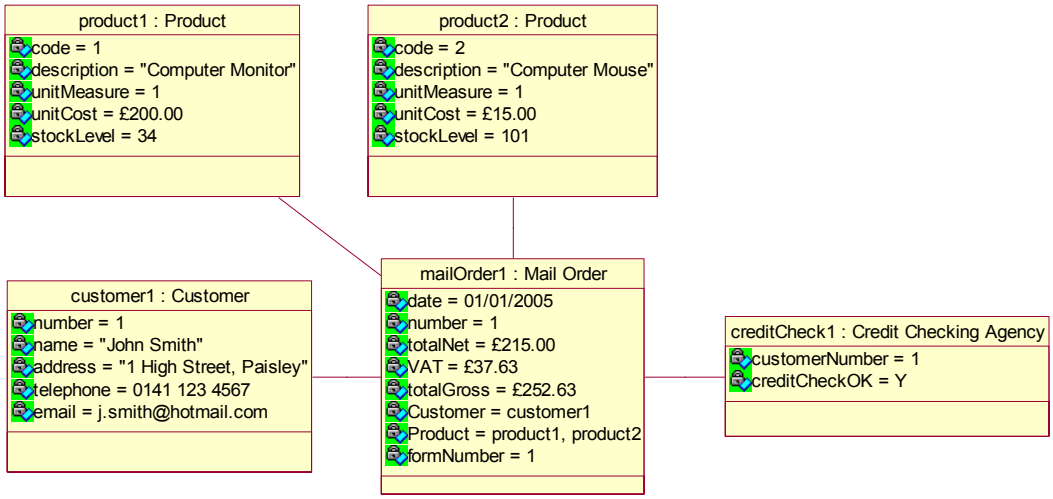


Figure 2.46: An Object Diagram

2.7 Construction of Interaction Diagrams

Interaction diagrams model the actor's interactions with the system, i.e. they model the scenarios that comprise the use cases in the use case diagram. More specifically, they model the flows of control in the system. They show a set of related objects, their relationships and the messages that pass between the objects.

There are two types of interaction diagram, namely, sequence diagrams and collaboration diagrams. In [13], Booch, Jacobson and Rumbaugh say "*A sequence diagram is an interaction diagram that emphasizes the time ordering of messages; a collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.*" Sequence diagrams and collaboration diagrams are isomorphic, i.e. sequence diagrams can be transformed into collaboration diagrams, and vice versa.

2.7.1 Sequence Diagrams

In [13], the authors say "*Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.*" Usually, the actor(s) that initiate(s) the interaction is(are) placed on the left of the X axis and the objects are placed along the X axis with the most significant to the left and the least significant to the right. The actor(s) and objects are placed at the top of the diagram. The messages that the objects and actor(s) send and receive are placed along the Y axis with the earliest message at the top of the diagram and the last message at the bottom. Messages can be iterative, i.e. sent several times, which is indicated by an asterisk before the message or the messages can be conditional, i.e. only sent if a condition applies, which is indicated by the condition being placed in square brackets before the message. Messages can create and delete objects – these messages have create and delete stereotypes and any necessary parameters are in round brackets after the create stereotype. Messages may be reflexive, i.e. an object can call an operation on itself. Beneath each object is a vertical dashed line which is referred to as the objects *lifeline*. The object's lifeline represents the existence of the object over time – usually this lasts for the duration of the interaction. Lastly, there are long, thin rectangles placed along the objects' lifelines, which indicate that the object is performing a task. Each rectangle is known as a *focus of control.* The top of the focus of control indicates the start of the task, usually accompanied by an initiating message, and the bottom of the focus of control indicates the end of the task, sometimes accompanied by a returning message.

The method which one should employ in constructing a sequence diagram is as follows:

1) Identify the scenario(s) that is(are) to be modelled.
2) Identify the actor(s) that participate(s) in the interaction. Lay them out along the top of the diagram, with the actor that most directly interacts with the system, to the right of the diagram, and the actor that least directly interacts with the system, to the left of the diagram.
3) Identify the object(s) that participate in the interaction. Lay them out along the top of the diagram with the most important objects to the left of the diagram and the least important objects to the right of the diagram.
4) Add the actor(s) and objects' lifelines.
5) Starting with the initiating message (usually from an actor to the controller class), lay out each message from top to bottom between the lifelines, with the earliest message at the top and the last message at the bottom. Show the properties of the messages, e.g. <<create>>, conditions, iteration, etc.
6) Add the focus of control to each object's lifeline.

Example 2.4
Figure 2.47 below shows a possible sequence diagram corresponding to the scenarios from the *Process Mail Order* use case shown in Figure 2.43 on page 30 (use case diagram). The use case description (pseudo-code) corresponding to these scenarios is as follows:

```
Clerk receives order form
Clerk enters customer details
For each product
        Clerk enters product details
Next
Order is created
Order is validated
If creditCheckOK = “No” Then
        Display rejection
        Clerk writes to customer
Else
        Generate bill
        Clerk prints bill
        Clerk ships goods + bill
End If
```

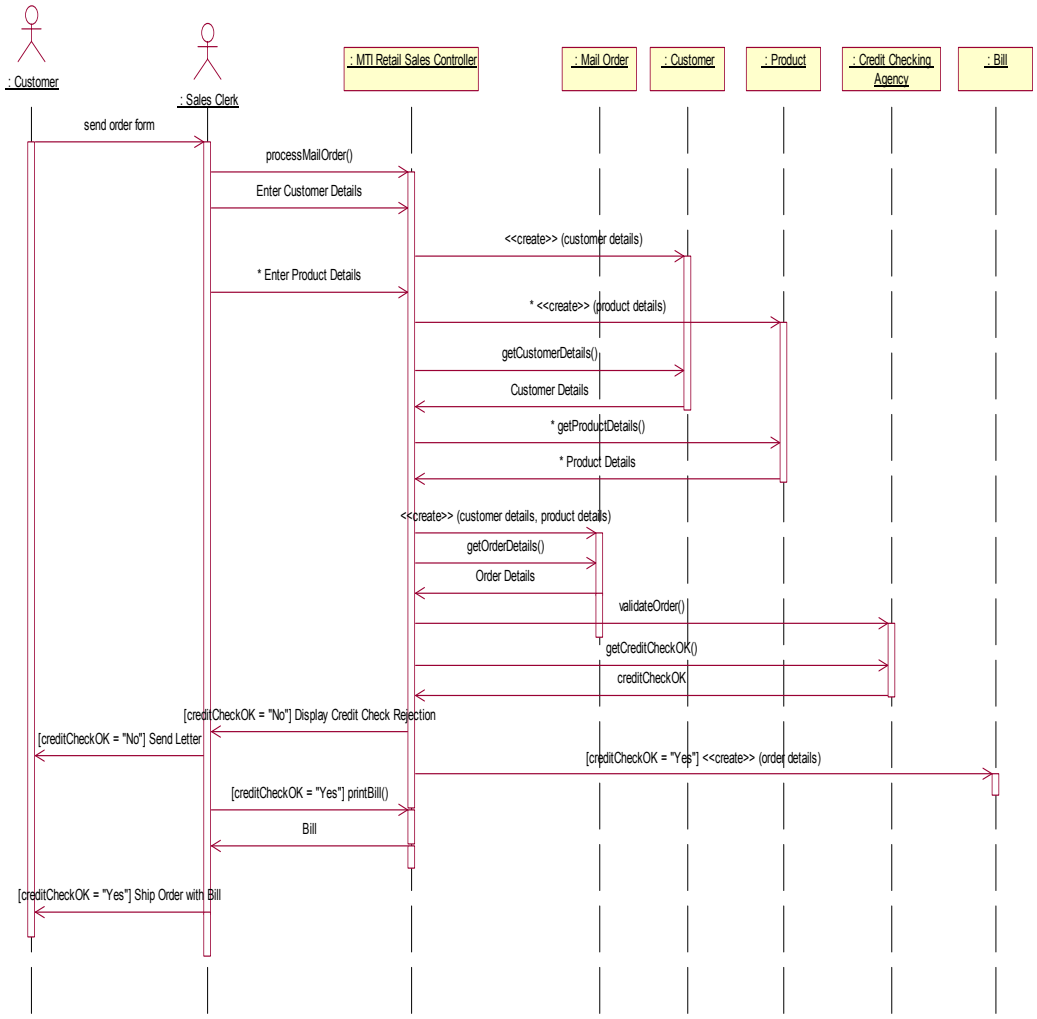


Figure 2.47: A Sequence Diagram

Once the sequence diagram has been created the class diagram (the main diagram in the analysis and design) can be updated with methods (operations) and, possibly, attributes identified from the sequence diagram.

The first method in the sequence diagram will correspond to the name of the relevant use case from the use case diagram and will be added to the controller class. Subsequently, messages received by an object become methods on the corresponding class, e.g. in our example above getOrderDetails() will become a method on the class Mail Order. Similarly, if data is passed to an object and that data is stored by the object, this would indicate that the data would be held in an attribute of the object. Thus, we may need to add an attribute to the relevant class in the class diagram.

In light of the sequence diagram in Figure 2.47, our class diagram can be updated as follows (see Figure 2.48 below):

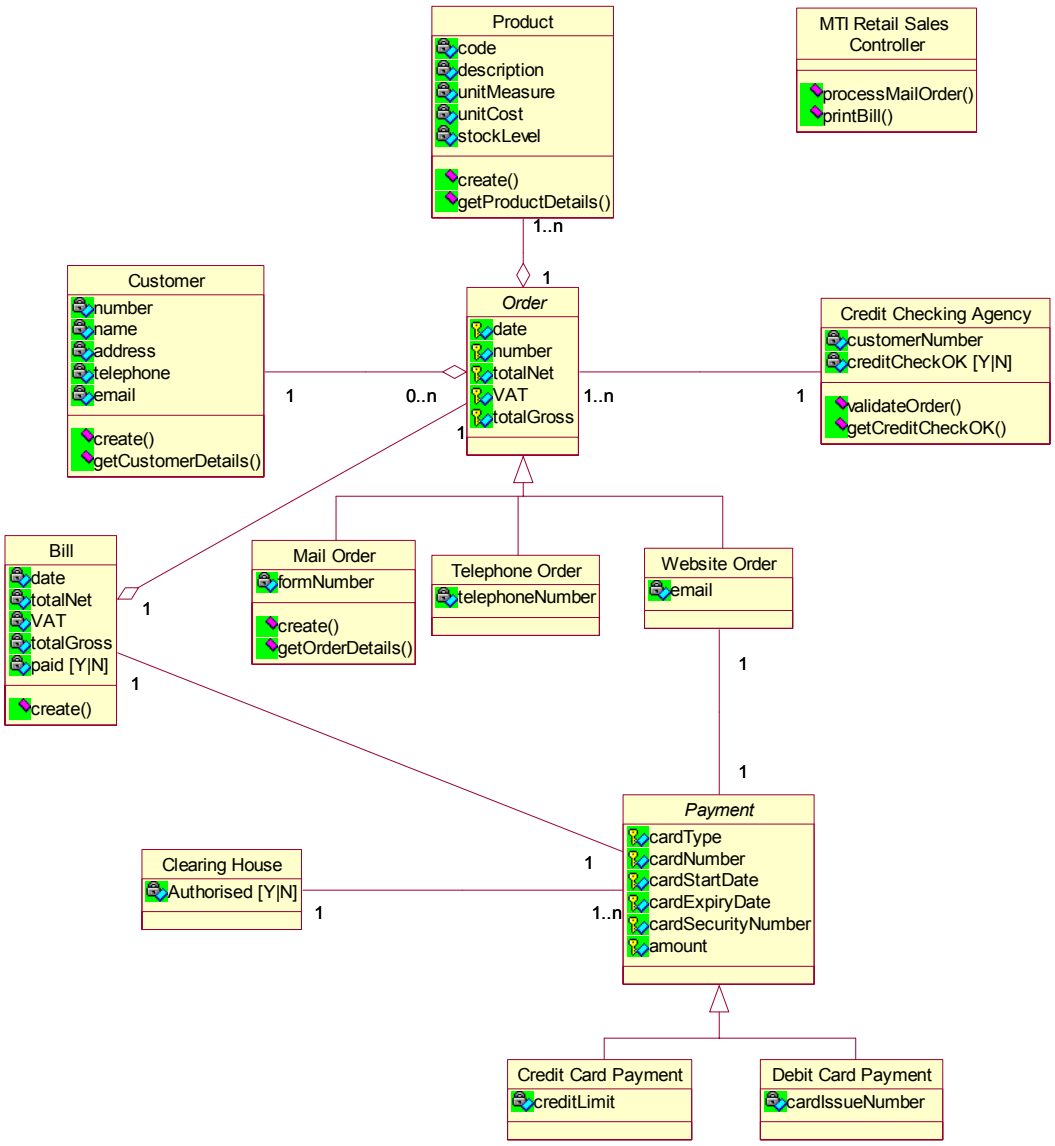


Figure 2.48: Updated MTI Retail Sales Class Diagram

2.7.2 Collaboration Diagrams

Collaboration diagrams show, mostly, the same information as sequence diagrams but with the emphasis on the organizational structure of the objects that send and receive messages rather than on the time ordering of the messages, as in sequence diagrams. Collaboration diagrams show the way in which objects are linked, e.g. with <<local>> or <<global>> stereotypes, whereas sequence diagrams do not give this information, and collaboration diagrams number messages to show the order in which they occur in time. However, the two diagrams are semantically equivalent.

The method which one should employ in constructing a collaboration diagram is as follows:

1) Identify the scenario(s) that is(are) to be modelled.
2) Identify the actor(s) that participate(s) in the interaction. Lay them out in the diagram as vertices of a graph.

3) Identify the object(s) that participate in the interaction. Lay them out in the diagram as vertices of a graph with most significant object(s) to the centre of the diagram.
4) If appropriate, set the initial values of the objects' attributes. If the objects' attribute values change during the interaction add a duplicate object to the diagram with the new attribute values and connect the old and new versions of the object by a message with a <<become>> or <<copy>> stereotype.
5) Identify the links between the objects, along which messages may pass. Lay out association links first and other links such as local or global second.
6) Attach the messages to the appropriate links, starting with the initiating message from an actor to the controller. The initiating message should be numbered 1, the second message should be numbered 2, and so on. Messages may be nested. In this case, they are numbered x.1, x.2, x.3, and so on, i.e. with Dewey decimal numbering.

Figure 2.49[12] below shows the collaboration diagram that corresponds to the sequence diagram shown in Figure 2.47

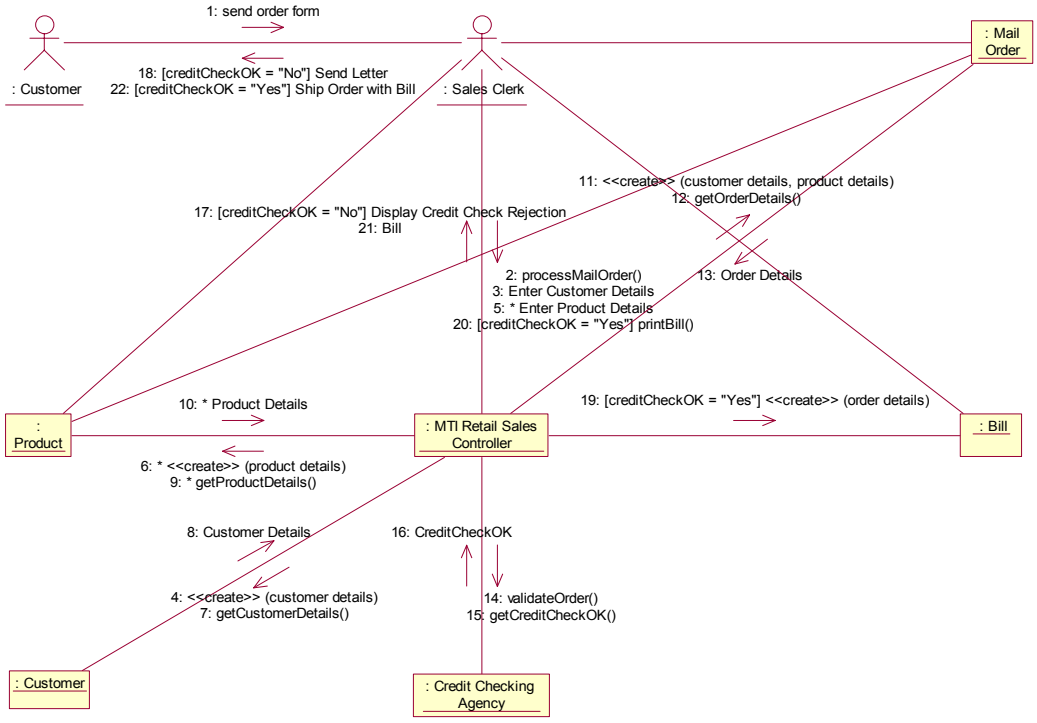


Figure 2.49: Collaboration Diagram corresponding to the Sequence Diagram in Figure 2.47

2.8 Construction of Statechart Diagrams

In [13], Booch, Jacobson and Rumbaugh say "*A Statechart diagram is basically a projection of the elements found in a state machine.*"

Statechart diagrams model the lifetime of reactive objects – the objects react to events (messages which they receive). In particular, they show the event ordered behaviour of reactive objects during the objects' lifetime.

[12] The collaboration diagram in Figure 2.49 was generated automatically from the sequence diagram in Figure 2.47 using the Rational Rose Enterprise Edition CASE tool by first constructing the sequence diagram in Rational Rose then, when the sequence diagram window in Rational Rose was active, *F5* on the keyboard was pressed. Sequence diagrams can be automatically generated from collaboration diagrams in the same way.

Statechart diagrams can be used to model the dynamic aspects of a system, class or use case. When being used to model a class, statechart diagrams can be used to identify additional attributes and operations of the class being modelled.

The method which one should employ in constructing a statechart diagram is as follows:

1) decide whether it is a system, class or use case that is to be modelled
2) decide on the initial and final states for the object (with pre- and post-conditions, if appropriate)
3) decide on the stable states of the object, starting with high-level states and then considering possible substates
4) decide on the meaningful partial ordering of stable states over the object's lifetime
5) decide on the events that could trigger a transition from state to state
6) attach actions to the states and/or transitions
7) simplify the diagram by using substates, branches, forks and joins
8) make sure that each state can be reached through some combination of events
9) make sure that each state, other than the final state, can be transitioned into another state by some combination of events
10) check the state machine against scenarios of typical interactions (sequences of events and their responses).

In [28], McCready summarises this by saying the "*steps in constructing a state machine diagram* [are as follows:]

- *prepare scenarios of typical interactions*
- *identify events and states*
- *construct the diagram*
- *analyse the diagram for attributes/operations of relevant classes*
    - o *the class for which the state diagram has been constructed as well as for classes that interact with this class (as indicated by e.g. guard conditions)*"

Guard conditions are included next to the name of the event that labels the transition and any action attached to the transition. See Figure 2.50 below.

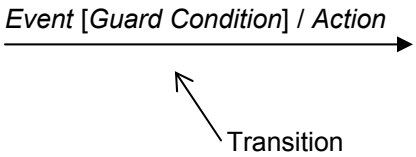


Figure 2.50: A Transition

Note that events, guard conditions and actions are optionally attached to transitions, i.e. a transition may be triggerless (have no event attached to it, no guard condition attached to it and no action attached to it).

Figure 2.51 below shows the statechart diagram for the class Bill in Figure 2.48 (Updated MTI Retail Sales Class Diagram) on page 38.

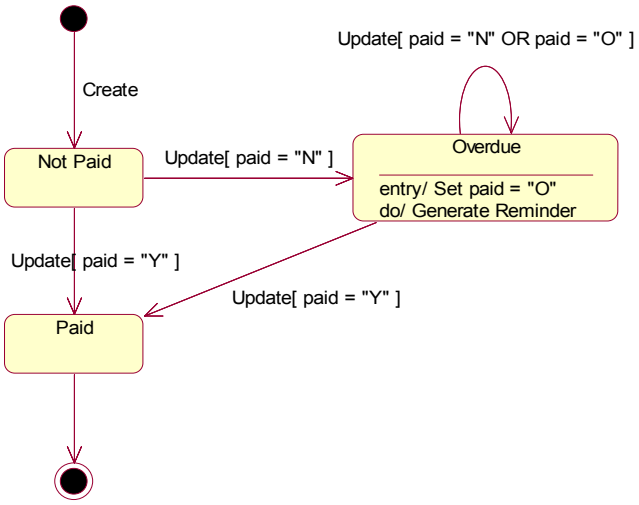


Figure 2.51: Statechart Diagram for class Bill from MTI Retail Sales Class Diagram in Figure 2.48

The class diagram in Figure 2.48 can now be updated, in light of the statechart diagram above. See Figure 2.52 below.

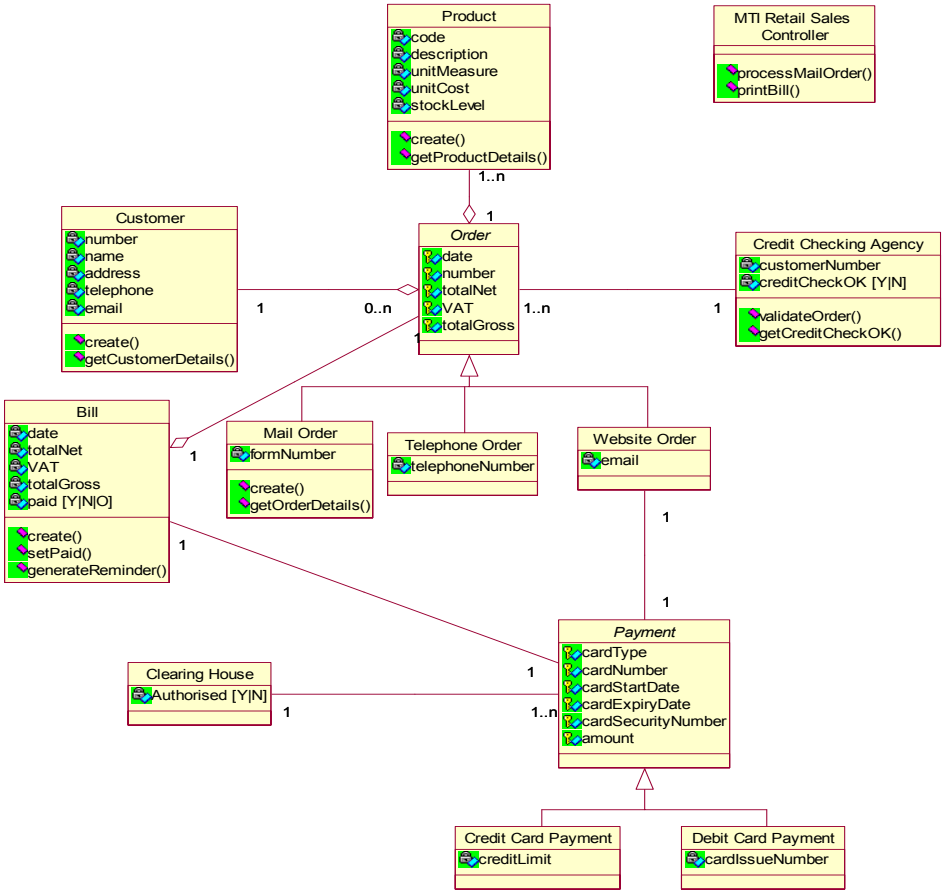


Figure 2.52: Updated MTI Retail Sales Class Diagram

2.9 Construction of Activity Diagrams

In [13], Booch, Jacobson and Rumbaugh say "*An activity diagram is a special case of a statechart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state.*"

Activity diagrams are commonly used to model a workflow or model an operation. In the case of modelling a workflow, activity diagrams focus on activities as seen by the actors that interact with the system. In the case of modelling operations, activity diagrams are used as flowcharts to model the details of a computation.

Activity diagrams usually contain action states and activity states, transitions and objects as well as initial and final states, decision diamonds, synchronization bars and swimlanes.

Action states and activity states are represented by lozenge shapes inside of which is written any expression to represent the action or activity being undertaken. Activity states can be decomposed with their activity being represented by other activity diagrams, and their work can be interrupted by other events. Action states cannot be decomposed and the work of the action state cannot be interrupted. See Figure 2.53 below.

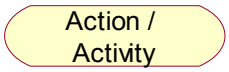


Figure 2.53: An Action/Activity State

Transitions are shown as directed lines which are usually not labelled as the transitions are caused by the completion of the previous activity/action (in the source state).

Objects are represented in a similar way as they are in object diagrams but with the state of the object named in square brackets below the object's name. See Figure 2.54 below.

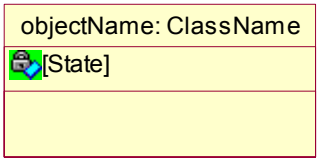


Figure 2.54: Object with its state shown

Objects are shown when, for example, an activity creates or modifies the object in question.

Initial and final states are represented in the same way as they are in statechart diagrams. See Figure 2.55 below.

Initial State Final State

Figure 2.55: Initial and final states.

Decision diamonds allow branching which specifies alternate paths to be taken based on guard conditions attached to each outgoing transition. See Figure 2.56 below.

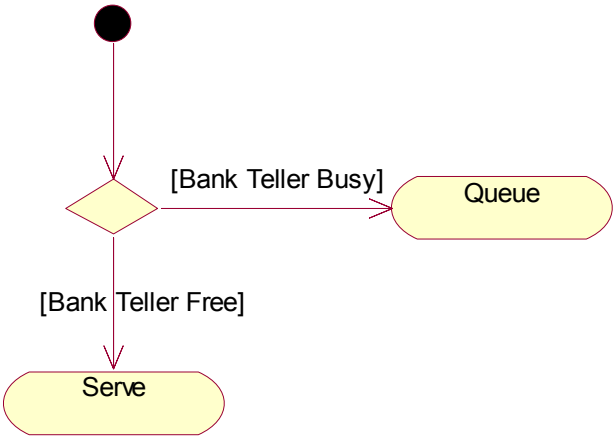


Figure 2.56: Decision Diamond

Synchronization bars allow concurrent flows of control. Forks let the flow of control split into two or more concurrent paths while joins synchronize these concurrent paths – when all the concurrent activities and actions have been completed flow of control passes on to a single path again. See Figure 2.57 below.

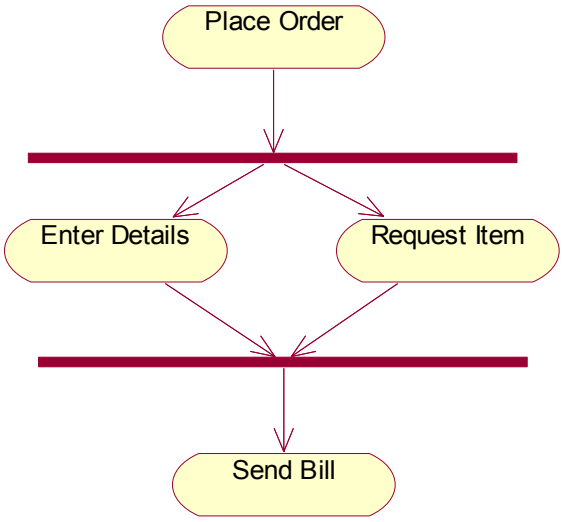


Figure 2.57: Synchronization Bars

Swimlanes partition the activity states and action states into groups on an activity diagram. This is particularly useful when modelling business processes. They can be used to allocate activities to system objects, actors, etc. Swimlanes are so called because each group of states is separated from its neighbour by a vertical solid line. See Figure 2.58 below (Figure 2.58 is taken from [13]).

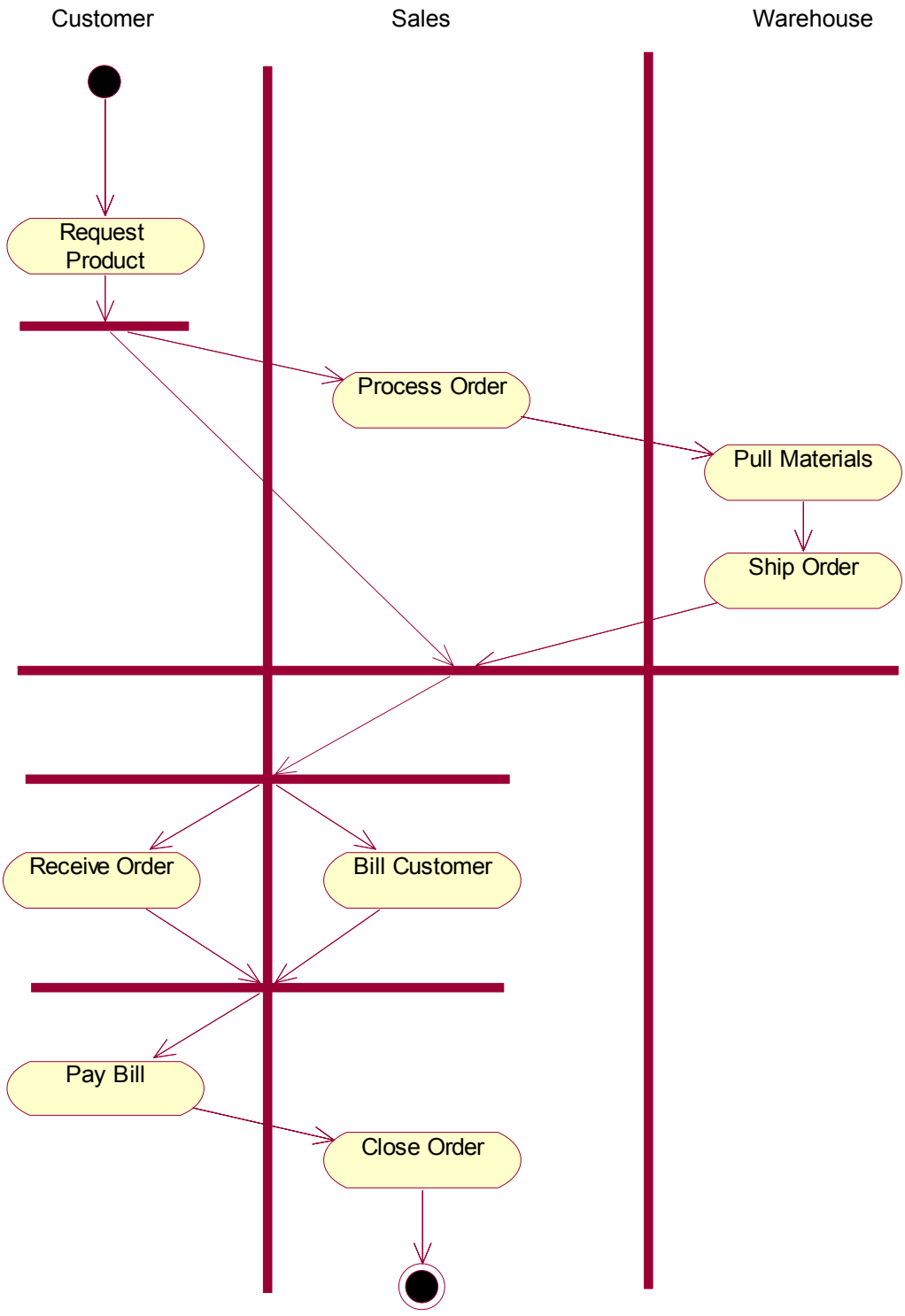


Figure 2.58: Swimlanes grouping activities (diagram taken from [13])

The method which one should employ in constructing an activity diagram to model workflow is as follows:

1) Establish what aspect of the system is to be modelled, i.e. ascertain a focus for the workflow.
2) Create a swimlane for each important business object (e.g. customer, sales department, etc.) of the system being modelled.
3) Identify the pre- and post-conditions of the initial and final states, respectively.
4) Identify the actions and activities that take place and show them as activity states and action states in the diagram.
5) Show complicated actions and actions that appear several times as activity states with separate activity diagrams that expand them.
6) Show the transitions that connect the action states and activity states, starting with sequential flow of control, then branching and, finally, synchronization.
7) If there are objects which are, for example, created or modified by an activity, show these objects along with their changing values and states.

The method which one should employ in constructing an activity diagram to model an operation is as follows:

1) Ascertain the operation's variables and parameters, the attributes of the class to which the operation belongs and the attributes of any other adjacent classes.
2) Identify the pre- and post-conditions of the initial and final states, respectively.
3) Identify the actions and activities that take place and show them as action states and activity states in the diagram.
4) Show the transitions that connect the action states and the activity states, starting with sequential flow of control and then branching, to show conditional paths and iteration.
5) If and only if the operation is owned by an active class, use synchronization to show concurrent flows of control.

Note that, activity diagrams will usually only be used to model complicated operations that are not easily understandable by simply looking at the code.

2.10 Construction of Component Diagrams

In [13], Booch, Jacobson and Rumbaugh say "*A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.*"

Interfaces are collections of operations that are used to specify the services provided by a class or component.

A component that realizes an interface is connected to that interface by an elided realization. See Figure 2.59 below.

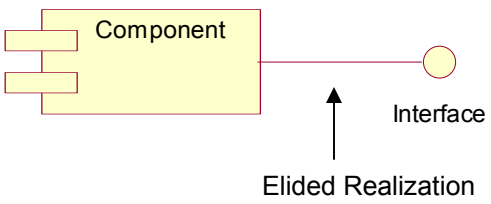


Figure 2.59: A Component realizing an Export Interface

An interface that a component realizes is called an *export interface* because it is an interface that provides services to other components. An interface that a component uses is called an *import*

interface since it is an interface that the component conforms to and builds on. Components that use the services of an interface, hence another component, are connected to them by a dependency relationship. See Figure 2.60(a) and Figure 2.60(b) below.

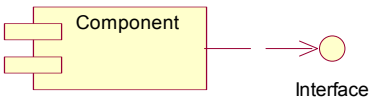


Figure 2.60(a): Component using the services of an Import Interface

Figure 2.60(b): Component 2 using the services of Component 1 (interface is elided)

There are three types of component, namely, *deployment components* such as .dll and .exe files, *work product components* such as source code files and data files, and *execution components* created as a consequence of a system executing, e.g. COM[13] or CORBA[14] objects.

Component diagrams model the physical aspects of systems. They show a set of components and their relationships. Component diagrams usually contain components, interfaces and dependency, generalization, association and realization relationships.

Component diagrams are usually used to model source code, executable releases, physical databases and adaptable systems.

The method which one should employ in constructing a component diagram to model a system's source code is as follows:

1) Identify the set of source code files to be modelled and render them as components stereotyped as files.
2) If the system being modelled is large, use packages to model groups of source code files.
3) Tag the components with information such as the version number of the file, the author of the file and the date the file was last edited.
4) Show the compilation dependencies between the source code files using dependency relationships.

The method which one should employ in constructing a component diagram to model an executable release is as follows:

1) Identify the set of components to be modelled.
2) Attach an appropriate stereotype to each component, e.g. file, table, executable, library, document, etc.
3) For each component, show its relationship to the other components. Commonly this will involve interfaces. If the seams of the system are to be shown explicitly, model the interfaces explicitly; otherwise, elide the interfaces.

In [13], the authors say the method which one should employ in constructing a component diagram to model a physical database is as follows:

1) *"Identify the classes in your model that represent your logical database schema."*
2) *"Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system."*
3) *"To visualize, specify, construct and document your mapping, create a component diagram that contains components stereotyped as tables."*

[13] COM :- Component Object Model
[14] CORBA :- Common Object Request Broker Architecture

4) *"Where possible, use tools to help you transform your logical design into a physical design."*

In [13], the authors say the method which one should employ in constructing a component diagram to model an adaptable system is as follows:

1) *"Consider the physical distribution of the components that may migrate from node to node. You can specify the location of a component instance by marking it with a location tagged value, which you can then render in a component diagram (although, technically speaking, a diagram that contains only instances is an object diagram)."*
2) *"If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. You can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value."*

Figure 2.61 below shows a component diagram which illustrates how an executable release of a Visual Basic .NET application could communicate with a relational database using ActiveX Data Objects.

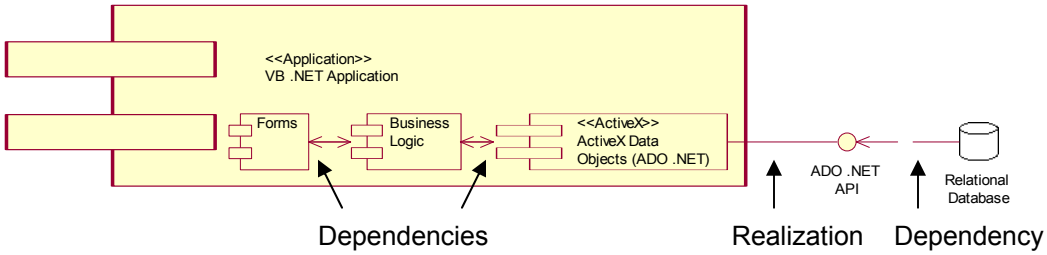


Figure 2.61: Component diagram showing VB .NET application communicating with a relational database

2.11 Construction of Deployment Diagrams

Usually, deployment diagrams are used to model the topology of the hardware on which object-oriented systems execute. They focus on a system's nodes.

In [13] Booch, Jacobson and Rumbaugh say "*A node is a physical element that exists at run-time and represents a computational resource, generally having at least some memory and, often, processing capability.*"

Most commonly, nodes split into two categories, namely, *processors* and *devices*. A processor is a node that can execute a component, i.e. it has processing capability. See Figure 2.62 below.

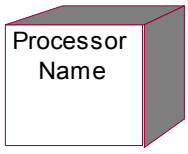


Figure 2.62: A processor node.

Generally, a device is a node which interfaces to the real world and has no processing capability, relative to the model's level of abstraction, e.g. a monitor, a thin client, etc.. See Figure 2.63 below.

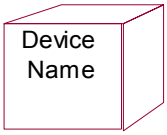


Figure 2.63: A device node

Deployment diagrams usually contain nodes, and dependency and association relationships. They are used to model embedded systems, client/server systems and fully distributed systems.

The method which one should employ in constructing a deployment diagram to model an embedded system is as follows:

1) Identify the system's devices and processors.
2) Give the system's devices and processors appropriate names which make them recognisable.
3) Model the relationships between these processors and devices in a deployment diagram, specifying the relationship between components from the component diagrams and the nodes in the deployment diagram.
4) Illustrate the inner workings of any intelligent devices using another deployment diagram.

The method which one should employ in constructing a deployment diagram to model a client/server system is as follows:

1) Identify the system's client and server processors.
2) Identify any relevant, i.e. architecturally significant, devices such as card readers, etc.
3) Give the system's devices and processors appropriate names which make them recognisable.
4) Model the topology between these processors and devices in a deployment diagram, specifying the relationship between components from the component diagrams and the nodes in the deployment diagram.

The method which one should employ in constructing a deployment diagram to model a fully distributed system is as follows:

1) Identify the system's client and server processors.
2) Identify any relevant, i.e. architecturally significant, devices such as card readers, etc.
3) Model the system to a sufficient level of detail to allow for reasoning about the system's performance or changes to the network, if necessary.
4) Logically group nodes together using packages, if necessary.
5) Model the topology between these processors and devices in deployment diagrams.

Figure 2.64 below shows a deployment diagram for a banks client/server system.

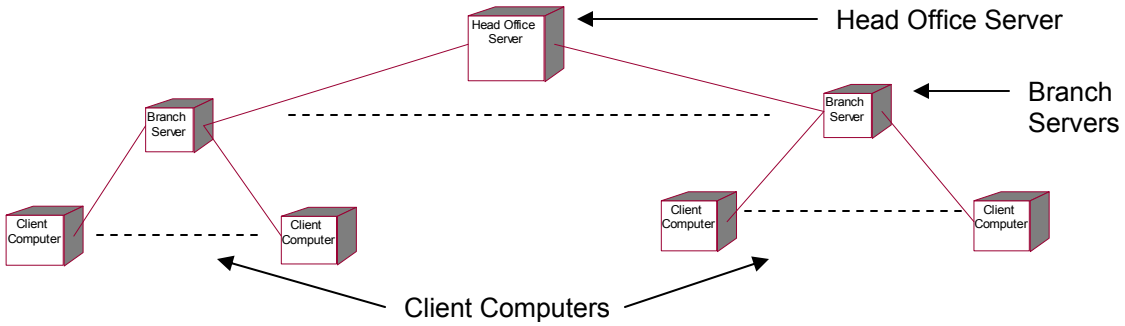


Figure 2.64: A bank's computer network

2.12 Extending UML with UCMs

In [2], Amyot categorizes the nine diagrams of UML into two sets, namely, *behavioural diagrams* and *structural diagrams,* as follows:

Behavioural Diagrams	**Structural Diagrams**
Use Case Diagrams	Class Diagrams
Sequence Diagrams	Object Diagrams
Collaboration Diagrams	Component Diagrams
Statechart Diagrams	Deployment Diagrams
Activity Diagrams	

Behavioural diagrams focus on functional and dynamic aspects of systems. Structural diagrams focus on components and static features of systems.

In [4], Amyot and Mussbacher say "*…there exists a conceptual gap between functional requirements (and use cases) and their realization in terms of behavioural diagrams…. Requirements and use cases often provide a black-box view where the system is described according to its external behaviour. UML behavioural diagrams have a glass-box view describing internal behaviour in a detailed way. A UCM view represents a useful piece of the puzzle that helps bridge the gap between requirements and design. UCMs can provide a gray-box view…"* This is illustrated in Figure 2.65 below (diagram taken from [2] [4] [5]).

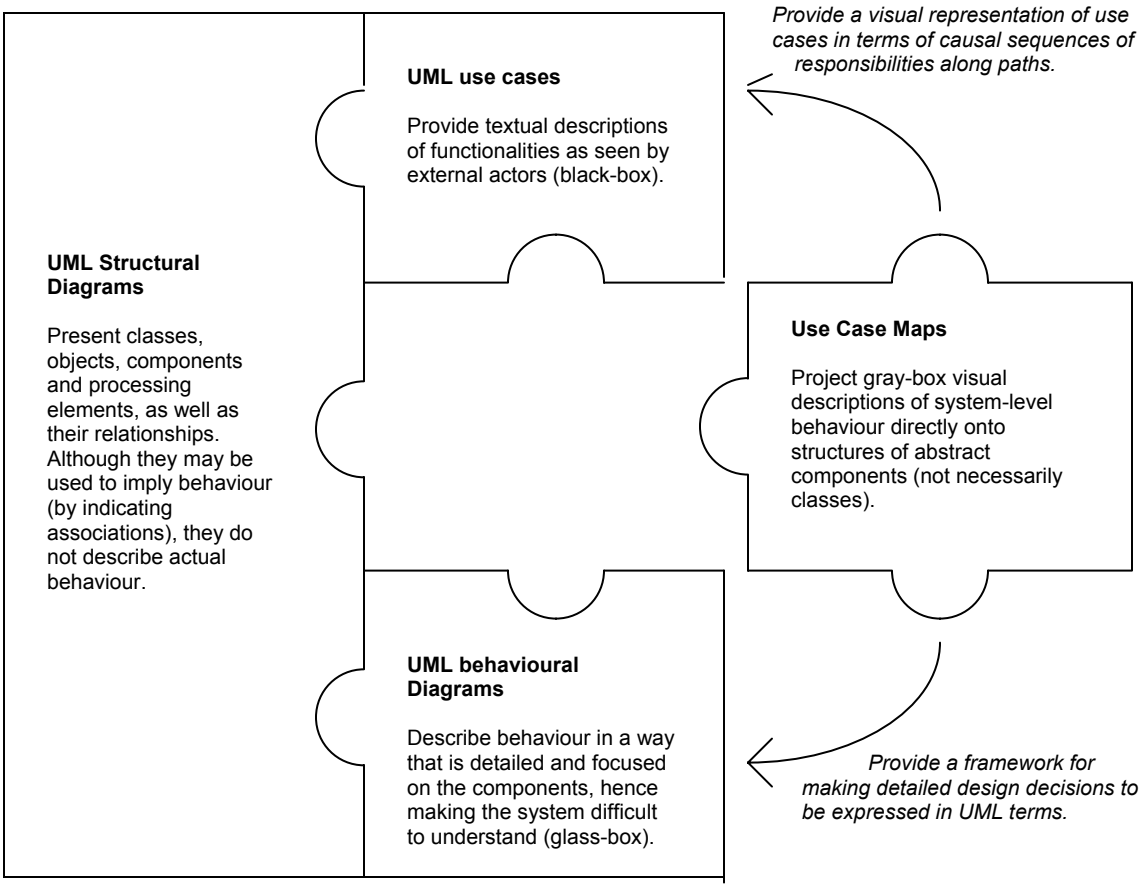


Figure 2.65: UCMs as a missing piece of the puzzle

In [5], Amyot and Mussbacher say "*The addition of several useful concepts found in the UCM notation to UML would make the latter a more appealing tool for designing reactive systems."* UCMs have a number of properties which would increase the appeal of UML. Firstly, UCMs allow scenarios to be mapped to different architectures, they allow variations of run-time behaviour and structures to be expressed, and they allow scenarios to be structured and integrated incrementally in such a way that undesirable interactions can be identified early in the design process [5].

Clearly, the UML is a comprehensive language and, similarly, the UCM notation has considerable scope. Hence, there is some semantic overlap between the two. For example, UML's use case diagrams show actors and use cases together with their relationships – the relationships between the use cases are *include, extend* and *generalization.* In UCM notation the include relationship can be represented by a static stub with the inclusion use case represented by a plug-in UCM (or sub-map). The extend relationship can be represented in UCM notation by either OR-forks or dynamic stubs, and the generalization relationship can be represented by either OR-forks/OR-joins or multiple dynamic stubs [2].

Similarly, in [5] the authors map UCM concepts to UML Activity Graphs metaclasses[15] in order to illustrate the conceptual overlap of the two. However, there are some differences. For example, in activity diagrams all of the paths leaving a fork must eventually merge in a subsequent join and multiple layers of forks and joins must be well nested. UCMs are not restricted in this way. Also, swimlanes allow UML activity diagrams to bind scenarios to system components but only in a one dimensional way. UCMs support bidimensional structures, i.e. components can contain sub-components [5]. In [3], Amyot and Eberlein say "*Activity diagrams share many characteristics with UCMs: focus on sequences of actions, guarded alternatives and concurrency; complex activities can be refined; and simple mapping of behaviour to components can be achieved through vertical swimlanes. However, activity diagrams do not capture dynamicity well, they do not support time constructs and the binding of actions to 'components' is semantically weak in the current UML standard."* Thus, it would appear that UCMs offer everything that UML activity diagrams have to offer and more.

In [2], the author identifies four possible ways of integrating UCMs with UML, as follows:

1) as UML is such a large language, there are UML *profiles* which provide a standard way of using the language in a particular area, e.g. object-oriented analysis and design or business modelling. One possibility is to tailor an appropriate profile so that it encompasses UCM concepts.
2) Simply add the UCM notation to the existing UML diagrams.
3) Extend existing UML diagramming techniques and semantics to support UCM concepts. The obvious candidate for this would be UML activity diagrams.
4) Substitute or reorganize one or more UML diagrams with UCMs. Again, activity diagrams are a candidate for this.

In this paper, we opt for option 2) and evaluate the benefits of adding a UCM view of our system.

[15] See Appendix A – Glossary for definition of *metaclasses.*

Chapter 3

The Case Study

3.1 Background

BestBank is a hypothetical Scottish bank that was formed in 1893. It is a high street (or clearing) bank registered in Paisley, Scotland which currently has 272 branches throughout the United Kingdom and Northern Ireland. *BestBank* has 3 million customers, manages assets of over £54 billion and had pre-tax profits for the year ending 5th April 2005 of £417 million.

BestBank currently offers its customers the following products:

- Mortgages
- Savings Accounts
- Current Account
- Personal Loans
- Home and Car Insurance
- Credit Card

BestBank intends to offer its customers a new current account this coming autumn – the *FlexiPlus* account. Note that *Bestbank* does not accept business from overseas customers, i.e. the *FlexiPlus* account will only be available to UK customers. This is because tax on the interest of *FlexiPlus* accounts (currently 20%) will be deducted at source and so *FlexiPlus* customers must be liable to UK taxation; hence, resident in the UK.

The *FlexiPlus* account will provide direct debit, direct credit, standing order and CHAPS transactions and a VISA debit card to its account holders, as well as normal deposit and withdrawal facilities via ATM machines and *Bestbank's* teams of bank tellers in its branches. Customers will also be able to pay household bills via their *FlexiPlus* accounts. Direct debit, direct credit and standing orders are all types of BACS transactions (see section 3.1.2 below).

BestBank holds all of its data on its HP NonStop SQL Database (an enterprise level database) which resides on *BestBank's* HP NonStop Server. See Appendix 2 – Overview of HP NonStop Servers.

BestBank requires a software application which will allow its branch staff to manage the new *FlexiPlus* accounts. The application is to be called "*FlexiPlus* Accounts Management System" or FAMS for short. The database to which FAMS will be connected will hold all of the *FlexiPlus* account and customer details, will reside on *BestBank's* HP NonStop SQL Database (on *Bestbank's* HP NonStop Server) and will be called "FAMSDB".

The FAMS application will allow *BestBank's* branch staff to make deposits and withdrawals, post interest (when closing an account), enter and update customer details, enter and update account details and, view and print account statements.

All BACS, CHAPS, VISA, ATM transactions and Bill Payments are dealt with by separate software agents that interact with the FAMSDB database autonomously. See sections 3.1.1 – 3.1.4 below. Cheque processing is dealt with by *BestBank's* Clearing Centre which has separate software agents that interact with the FAMSDB database autonomously. See section 3.1.5.

Separate software agents are also responsible for copying balance and transaction details to other accounting systems such as the general ledger, reporting systems (for things like management reports, risk reports, statutory reporting to the Bank of England, FSA and Inland Revenue) and anti-money laundering detection systems.

Non-functional Requirements such as volumes, throughput, response times, security, auditing, usability, user documentation, online help, training, testing facilities, availability, reliability, disaster recovery, etc. are to be dealt with by *BestBank's* Group Technology Services Department.

The detailed requirements specification for the FAMS application is contained in a letter from Robert Brown, *BestBank's* Head of Product Development, to the author. The letter can be seen in section 3.2 below.

3.1.1 ATM Transaction Procedures

BestBank's HP NonStop Server is connected to the LINK® cash machine (ATM) network; the busiest shared ATM network in the world. The LINK® cash machine (ATM) network is operated by LINK® Interchange Network Ltd.

The LINK® cash machine (ATM) network comprises ATMs that are connected to host servers via either leased line connections or dial-up connections – dial-up connections are preferred by retail merchants, e.g. ATMs in convenience stores or public houses, as they are cheaper to run than leased line ATMs. The ATM host servers are LINK® Interchange Network Ltd. member bank's servers. For a list of LINK® Interchange Network Ltd members see Appendix 5 - LINK® Interchange Network Ltd.' Member Banks, Building Societies and Independent ATM Deployers.
.

When a customer initiates a transaction, e.g. a withdrawal of money, through an ATM, the ATM forwards the transaction information to the ATM's host server. The information is then forwarded to *BestBank's* server via LINK's server. If the cardholder is requesting cash, the ATM's host server initiates an electronic funds transfer to take place from the customer's bank account to the ATM host's bank account. The ATM host's server will then send an approval code to the ATM authorizing the machine to dispense the cash. The ATM host's server then transfers the cardholder's funds into the merchant's[16] bank account, usually the next business day. The merchant is reimbursed for all funds dispensed by the ATM. All ATM transactions take place in real-time. See Figure 3.1 below (adapted from diagram in [8]).

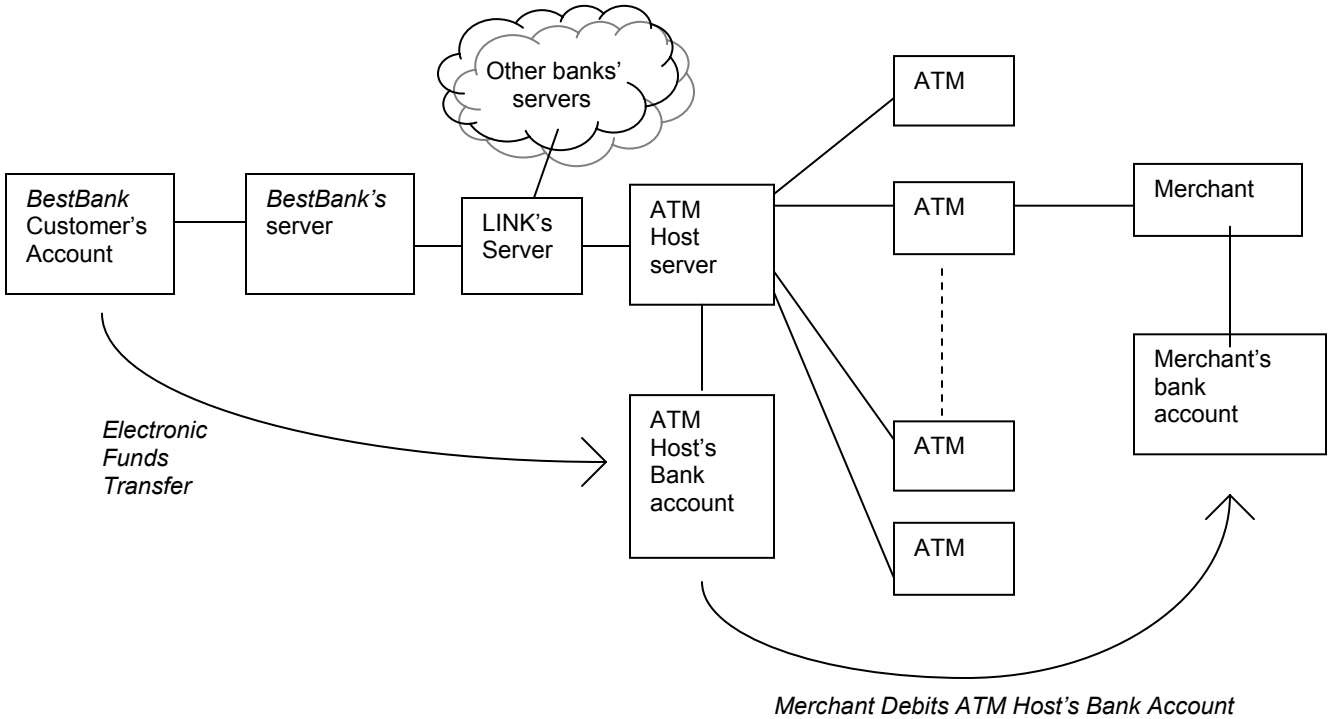


Figure 3.1: ATM cash withdrawal over LINK® cash machine (ATM) network

[16] The term *merchant* refers to anyone who hosts an ATM machine on their premises, e.g. a bank, a convenience store, a public house, etc.

3.1.2 BACS Transaction Procedures

BACS transactions comprise direct debit, corporate payments, dividend payments, credit payments such as standing orders, and weekly wages and monthly salaries, in the UK. Two companies in the UK are jointly responsible for BACS transactions.

BACS Payment Schemes Limited is a membership-based[17] industry body whose role is to develop, enhance and promote the use and integrity of automated payment and payment related services. It promotes best practice amongst those companies who offer payment services and is responsible for the associated payment clearing and settlement services. BACS Payment Schemes Limited carries out administration for Voca Limited.

Voca Limited (formerly known as BACS Limited prior to 12 October 2004) is the commercial company that physically processes BACS transactions. Voca Limited operates a three day payment cycle as follows:

Day 1: Input	7:00am - 11:30pm	BACS files are submitted, via the BACS Telecomms Service BACSTEL®, to Voca Limited either directly from member banks and building societies or from organizations sponsored by member banks or building societies. BACS files are specially formatted files which contain credit and debit items, headers, footers, etc.
Day 2: Processing	11:30pm (Day 1) – 9:00am	If specified in the BACS file, processing day can be deferred for up to 30 days, i.e. processing day can be up to 30 days after Input day. On receipt of the BACS files, Voca Limited sends the bank, building society or sponsored organization that sent the BACS file a BACS Online Telecomms Report – Acceptance or Rejection. This confirms that the BACS files conform to BACS standards before being passed through detailed validation. Detailed validation (by Voca) involves checking the BACS files against a series of criteria, e.g. account limits, previously submitted files, valid branch sort codes, etc. If any of these are incorrect the error or infringement is referred to the relevant member bank or building society who will decide whether the file can be processed or extracted from the system. After detailed validation an input report is sent to the bank, building society or sponsored organization confirming that the BACS file has been processed – this includes details of any amended, rejected or returned items for that file. Voca Limited then sorts the data from the BACS files into payments for each member bank or building society and sends them the relevant files. The data is then downloaded into the bank's or building society's computer systems.
Day 2: Processing	9:00am - 12 midnight	The member banks and building societies validate the files, e.g. check that customers' accounts have not been closed.
Day 3: Entry	12 midnight (Day 2) - 9:30am	Items are applied to customers' accounts and settlement occurs between the member banks and building societies.

[17] See *Appendix 4 – BACS Payment Schemes Limited' Member Banks and Building Societies*.

Voca Limited provides member banks and building societies with hardware and software to allow them to connect to Voca's network (see Appendix 3 – Overview of Voca Limited Connectivity Products).

3.1.3 CHAPS Transaction Procedures

Clearing House Automatic Payments System, or CHAPS for short, is a real-time gross settlement system (RTGS) used for high-value transactions, where money is transferred from one bank to another on the same day.

There are two components to CHAPS: one operating in sterling; the other operating in euro. CHAPS euro is linked to the pan-European RTGS system, TARGET. TARGET links all the European central banks with each other.

The main (UK) banks and building societies are "direct" members of CHAPS. There are also over 400 "indirect" members – usually smaller banks and building societies who have access to the system through a direct member.

CHAPS opens for business at 6:00am every day and payments have to have started by 4:00pm each day. Later payments (up to 5:00pm) are used for settlement between banks (usually their Treasury Departments).

The CHAPS payment cycle is as follows:

1) The customer (usually a business or firm of solicitors) asks their bank to make a CHAPS payment to another (receiving) bank account (usually the bank account of the business's customer or another firm of solicitors)
2) This instruction is passed to the customer's bank's central processing centre who validate the instruction, e.g. check that there are sufficient funds in the customer's bank account, check that the sort code of the receiving account is valid, etc.
3) The payment is made (electronically) to the receiving bank account via The Bank of England's inter-bank payment and settlement system, i.e. an electronic funds transfer takes place between the customer's bank account and the receiving bank account.

Note that all CHAPS transactions take place in real-time.

3.1.4 VISA Transaction Procedures

Visa International is a membership association owned by 21,000 financial institutions worldwide. Visa Europe is one of six independent regions that make up Visa International. The EU region has more than 5,000 members in Western Europe, Israel and Turkey.

When a customer goes into a shop to make a purchase using their Visa debit card, their card is swiped using an Electronic Point of Sales (EPOS) terminal and the retailer goes online to seek authorisation for the transaction. A request for authorisation is sent via the Acquiring bank's server (the retailer's bank's server) to *BestBank's* server. *BestBank* verifies that the card details are correct and checks that the customer has enough money in their account to cover the transaction. If all is in order, *BestBank's* server sends an approval message and authorisation code back to the Acquiring bank's server which in turn passes the approval message and authorisation code to the EPOS terminal. The transaction then goes ahead and the funds are held to one side on the customer's account for a few days until the retailer has time to process the debit. When the retailer submits the debit to their bank (the Acquiring bank) the customer's account is then debited and the retailers account is credited. See Figure 3.2 below.

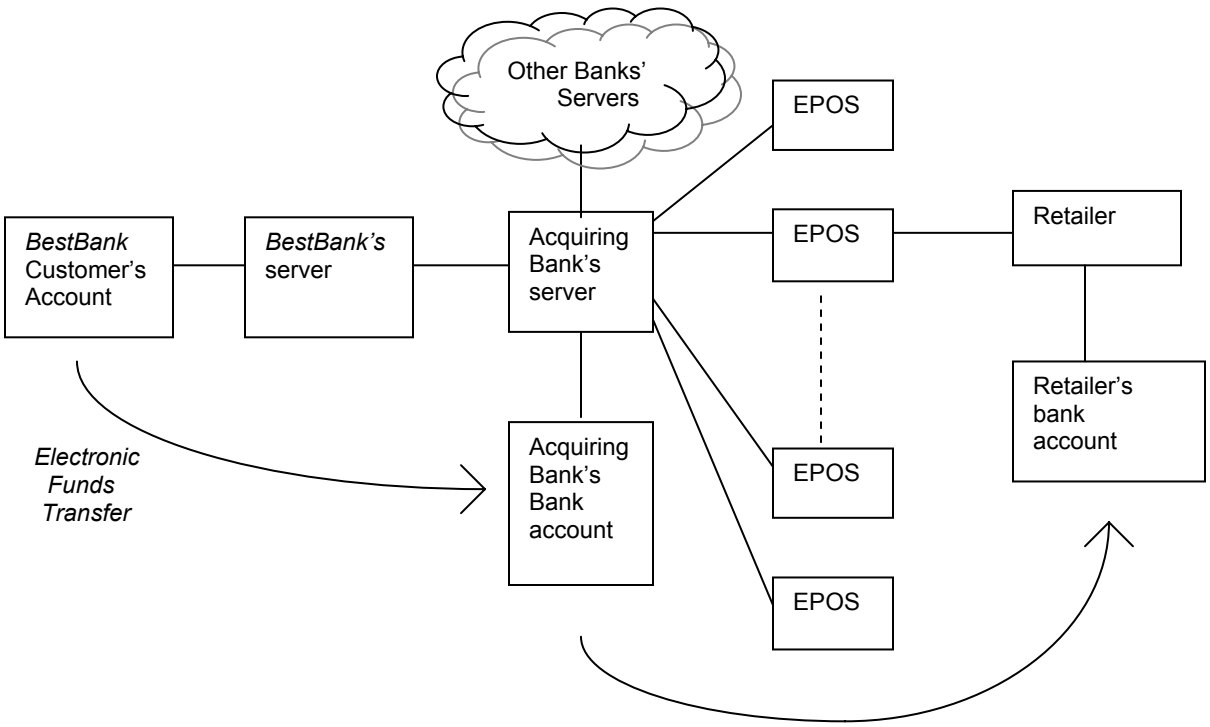


Figure 3.2: Visa Debit Card Transaction

3.1.5 Cheque Processing

The Cheque and Credit Clearing Company is responsible for the bulk clearing of cheques and paper credits in Great Britain. In Northern Ireland cheque and credit payments are processed locally.

Members of the Cheque and Credit Clearing Company are individually responsible for processing cheques drawn by or credited to their customer's accounts. For a list of the Cheque and Credit Clearing Company's members see Appendix 6 – Cheque and Credit Clearing Company Members.

A cheque is defined as a written order by an account holder to his banker to pay a specified sum of money to the bearer or named recipient. A cheque is not legal tender but is a legal document, the use of which is governed by the Bills of Exchange Act 1882, and the Cheques Acts of 1957 and 1992.

The Cheque and Credit Clearing Company operates a three working day processing cycle for cheques, as follows:

Day 1	A cheque paid into a member bank anywhere in Britain during the course of the day's business will normally be processed by the collecting bank that evening. Each member bank's branch will pass the cheques they have received that day to its bank's clearing centre. Cheques will arrive at the bank's clearing centre by the early hours of Day 2.
Day 2	At the collecting bank's clearing centre the cheques are mechanically read and sorted, and codeline and amount details of each cheque are passed electronically through a secure data exchange network, called the Inter Bank Data Exchange, to the appropriate paying bank's clearing centre by 11:00am on Day 2.

Day 2	The cheques are then delivered to an Exchange Centre at either Milton Keynes or Edinburgh, where member banks hand over all the cheques drawn on the other banks and collect all the cheques drawn on themselves.
Day 3	On the morning of Day 3, bank staff review the cheques presented for payment and make decisions about whether to pay or return them. If the paying bank decides not to pay a cheque the cheque is returned to the collecting bank by first-class post on Day 3, or under certain circumstances Day 4. The collecting bank will receive the unpaid cheque on Day 4 or Day 5. Settlement between the member banks, for the net values of the cheques exchanged between them, takes place over their Bank of England settlement accounts on Day 3.

Interest may be paid on a cheque that has been paid into an account at any time from Day 1 onwards. Every bank makes its own commercial decision on the issue of when to start paying interest on cheques.

Similarly, if a customer's account is overdrawn, the overdraft can be reduced at any time from Day 1 onwards. Every bank makes its own commercial decision on the issue of when to reduce overdrafts.

Lastly, customers may be allowed to withdraw funds from Day 1 onwards. Again, this issue of when to allow customers to make a withdrawal on a cheque is a commercial decision for the individual bank to make.

3.2 Requirements Specification for "*FlexiPlus* Accounts Management System"

The following letter from Robert Brown, *BestBank's* Head of Product Development, to the author outlines the requirements specification for the "*FlexiPlus* Accounts Management System" application.

1st May 2005

BestBank

Head Office
100 – 105 Case Study Street
Paisley
PA1 8XQ
Scotland
Tel.: +44 (0)141 020 4000
Fax.: +44 (0)141 020 4001
Email: enquiries@bb.co.uk
Web: www.bb.co.uk

Mr Bill Blackley
MSc IT Student
University of Paisley
Paisley
PA1 2BE
Scotland

Dear Mr Blackley,

Requirements Specification for "*FlexiPlus* Accounts Management System (FAMS)": Branch Application for *FlexiPlus* Accounts.

BestBank requires a Windows application that will allow its Bank Tellers in its branches to manage the new *FlexiPlus* accounts. The application is to be called "*FlexiPlus* Accounts Management System" or FAMS for short. The FAMS application must allow *BestBank's* branch staff to make deposits and withdrawals, post interest (when closing an account), enter and update customer details, enter and update account details and, view and print account statements.

BestBank holds all of its data on its HP NonStop SQL Database which is on its HP NonStop Server at Head Office in Paisley.

Only authorised members of *BestBank's* staff should be able to access the FAMS application.

Cash Deposits
When a customer deposits cash in their *FlexiPlus* account that sum must be immediately available for withdrawal, must immediately start earning interest provided the account is not overdrawn and must immediately reduce any overdraft on the account. The transaction is to be described as "Cash" on account statements.

Cheque Deposits
Cheques are to have a three working day clearing period. That is to say, sums deposited by cheque will not be available for withdrawal, start earning interest or reduce any overdraft for three working days. This is to allow for the normal three working day processing cycle for cheques, as operated by the Cheque and Credit Clearing Company. Thus, if a customer deposits a cheque for £100.00 on Monday that £100.00 will not be available for withdrawal, earn interest or reduce any overdraft until the following Wednesday. The transaction is to be described as "Cheque" on account statements.

The FAMS application will merely allow cheques to be deposited in *FlexiPlus* accounts. The mechanics of cheque processing, including making cleared funds available for

withdrawal, available to earn interest and available to reduce any overdraft, will be carried out by *BestBank's* Clearing Centre.

Inter-Account Transfers
Inter-Account Transfers can take place between *FlexiPlus* accounts and any other *BestBank* account. They can take the form of deposits or withdrawals. In both cases, the paying account is debited and the collecting account is credited, by the sum being transferred. The transaction is to be described as "Transfer" on account statements.

In the case where a *FlexiPlus* account is collecting a sum from another account, the sum transferred into the *FlexiPlus* account will be immediately available for withdrawal, will immediately start earning interest provided the account is not overdrawn and will immediately reduce any overdraft on the account.

In the case where a *FlexiPlus* account is paying a sum into another account, the sum must be debited immediately from the account, any interest earned on the account after the Inter-Account Transfer must be reduced by an amount corresponding to the sum debited and any overdraft on the account must immediately be increased.

Cash Withdrawals
When a customer withdraws cash from their *FlexiPlus* account the sum must be debited immediately from the account, any interest earned on the account after the Cash Withdrawal must be reduced by an amount corresponding to the sum withdrawn from the account and any overdraft on the account must immediately be increased. The transaction is to be described as "Cash" on account statements.

Banker's Drafts
Customers can request a Banker's Draft to be drawn on their account. In this case the sum issued must be debited immediately from the account, any interest earned on the account after the Banker's draft has been issued must be reduced by an amount corresponding to the sum withdrawn from the account and any overdraft on the account must immediately be increased. The transaction is to be described as "Banker's Draft" on account statements.

Interest (to be implemented by the Database Administrator)
Interest on daily closing balances will be paid to all *FlexiPlus* accounts that are not overdrawn, i.e. interest will be paid on all *FlexiPlus* accounts which have a positive balance at close of business each day. The rate of interest will be paid at a flat rate to all *FlexiPlus* accounts, e.g. 1% below the base rate of interest, as set by the Bank of England's Monetary Policy Committee. Interest will be paid to all *FlexiPlus* accounts on the last day of each month, provided the account has not been closed.

Interest (to be implemented by the FAMS application)
When a customer requests that their *FlexiPlus* account is closed, the Bank Teller will post interest to the account prior to paying out the closing balance and closing the account. This must take account of overdraft charges due on the account.

Overdraft Charges (to be implemented by the Database Administrator)
Arranged overdrafts will be charged at a flat rate, e.g. 10% per annum. However, customers will only be charged for each day that they are overdrawn. Thus, overdrafts will actually be charged at a flat daily rate, e.g. for a rate of 10% per annum, they will be charged at approximately 0.02739726% per day. There will also be a flat rate charge of £25 for customers who exceed their arranged overdraft. Overdraft charges will be

deducted from *FlexiPlus* accounts at the end of each month. Overdraft charges are to be described as “Overdraft Charge” on account statements. The rate at which overdrafts are charged at will be reviewed each month in accordance with changes made to the base rate of interest, as set by the Bank of England’s Monetary Policy Committee.

Note that if the account is a “Student Account”, i.e. if the account holder is a student, they are to pay no charges for their arranged overdraft.

Customer Details
Personal details held about *FlexiPlus* account holders should include:

- Title
- Forenames
- Surname
- Address
- Post Code
- Contact Telephone Number
- Email Address
- Date of Birth
- Date of Death, if appropriate.

Account Details
Account details held about *FlexiPlus* accounts should include:

- Account Number
- Sort Code
- Opening Date
- Closing Date, if appropriate
- Balance
- Overdraft Amount
- Transactions
- Account Status
- Notes regarding the Account
- Card Number

Each Transaction record should include the following information:

- Date of Transaction
- Description of Transaction, e.g. “Cash”, “Cheque”, “Transfer”, etc.
- Debit or credit amount, i.e. the value of the transaction
- Balance

Transaction records should be archived on their second anniversary, i.e. after they are two years old, by the Database Administrator.

Account Status should be set to “Open” when the account is opened unless the customer is a student with a valid student card or matriculation card, in which case the Account Status should be set to “Student” and the customer will pay no charges for their arranged overdraft. Accounts can remain “Student” accounts for one year after the customer graduates from their course.

If *BestBank* is informed that the customer is deceased the account status should be set to “Deceased”. In order for the executor(s)/executrix(es) to close the account and have the funds paid out, *BestBank* will require sight of a copy of the customer’s death certificate. Once the death certificate has been presented, the customer’s Date of Death should be entered on the system and the account can be close, i.e. the closing balance paid out to the beneficiaries and the Account Status set to “Closed”.

Should a customer request that their account be closed the closing balance will be paid out according to the customer’s instruction and the account status set to “Closed”.

If no transactions take place on an account for more than one year the account’s status must be set to “Dormant”. Accounts that remain “Dormant” for five years or more will be closed and have their balances transferred (or written off if they are overdrawn) to *BestBank’s* Internal Suspense Account to make the ledgers balance.

Note that when closing a customer’s account interest must be posted to the account and any overdraft charges deducted from the account. Also, once an account has been closed, i.e. its status set to “Closed”, no further transactions should be permitted on the account.

Statements
BestBank’s staff must be able to view and print off statements of all transactions carried out on any *FlexiPlus* account for up to the previous two years (transactions are to be archived after two years by the Database Administrator). They must also be able to view and print off statements showing the transactions carried out during any period within the last two years by entering the “Start Date” and “End Date” of the period they wish to examine. The technical infrastructure for printing statements is already in place and is therefore beyond the scope of the FAMS development programme.

This concludes the Requirements Specification for the FAMS application.

Yours sincerely,

Robert Brown
Head of Product Development.

Chapter 4

Overview of the System's Architecture

4.1 Computer Network Topology

In section 3.2 it is given that *BestBank* holds all of its data on its HP NonStop SQL Database which sits on *BestBank's* HP NonStop Server at Head Office in Paisley. It follows that the topology of *BestBank's* hardware, i.e. its servers and client computers, is of the client/server type. The following deployment diagram models *BestBank's* computer network infrastructure. See Figure 4.1 below.

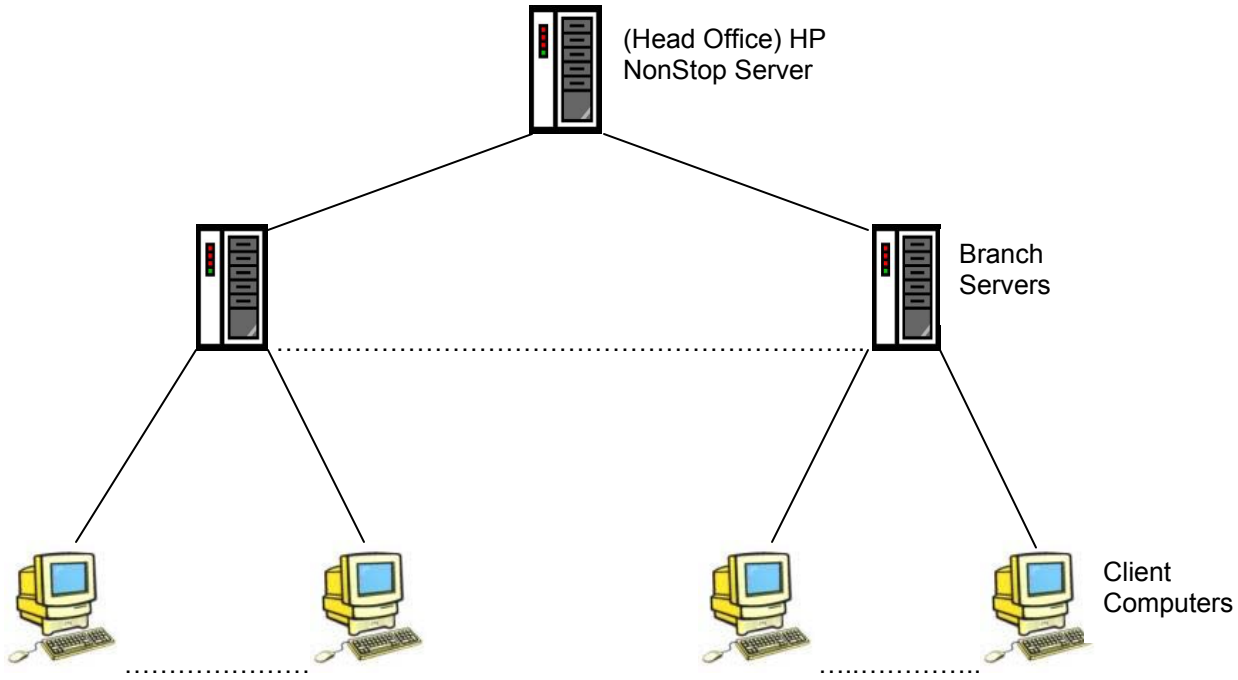


Figure 4.1: *BestBank's* Client/Server Computer Network Topology

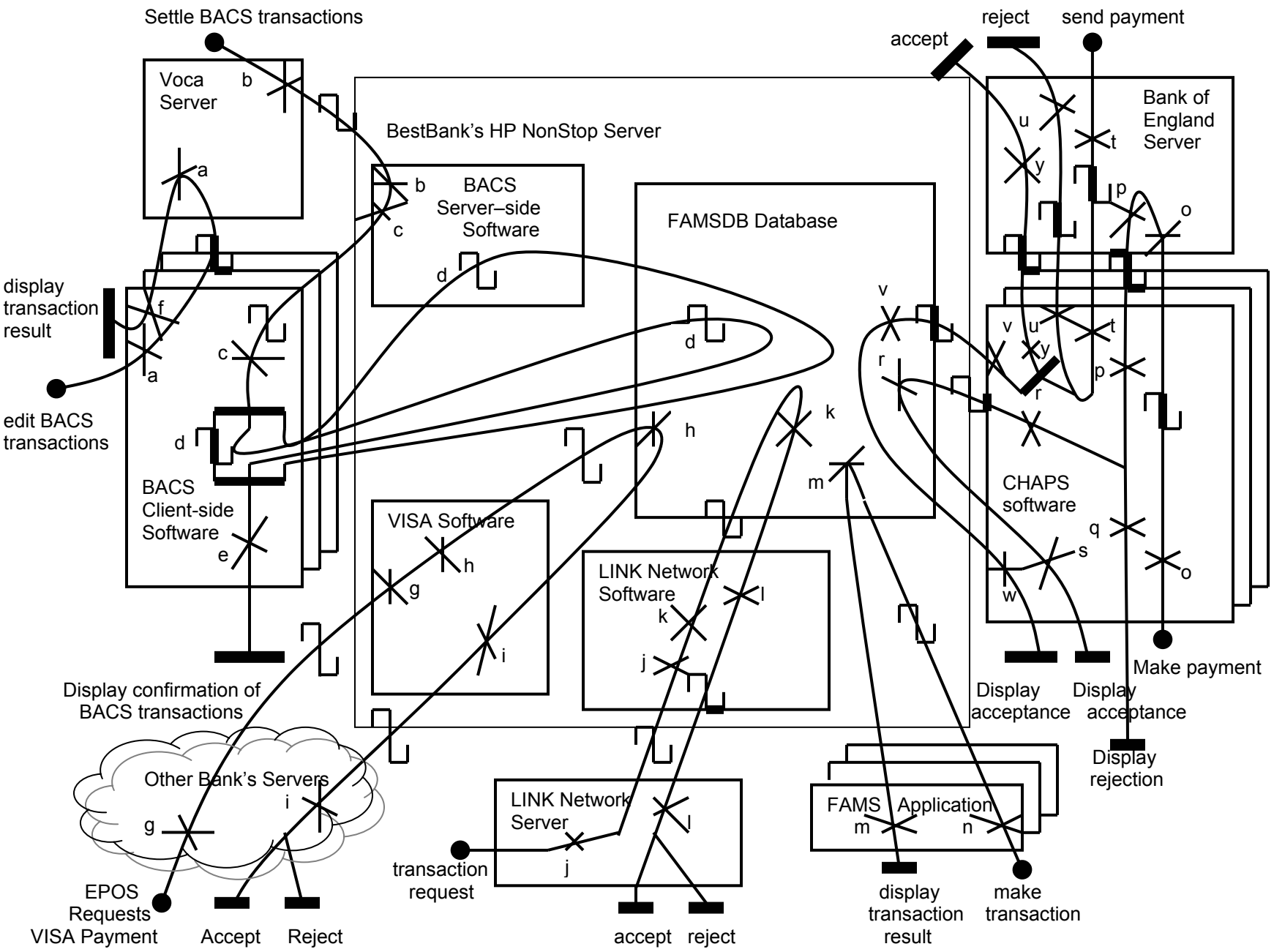
The FAMS application will reside on the client computers and communicate with the FAMSDB database which will sit on the Head Office server. Client Computers in BestBank's branches are connected to the Head Office server via a high bandwidth leased line.

4.2 Multi-Agent System

In section 3.1 it is given that all BACS, CHAPS, VISA and ATM transactions are dealt with by separate software agents. During the writing of Chapter 3 – The Case Study (the requirements gathering phase of this project) the following UCM was constructed in order to visualise the separation of software entities needed in order to provide all the services required for the *FlexiPlus* accounts. The mechanism for Bill Payments is omitted here. See Figure 4.2 below.

Note that this UCM is purely conjecture and is merely intended to illustrate a possible architecture for the separate autonomous software agents that deal with BACS, CHAPS, VISA and ATM transactions, as well as the architecture of the FAMS application and the FAMSDB database.

Figure 4.2: UCM – Overview of Required System for *FlexiPlus* Accounts



Responsibilities of Figure 4.2: UCM – Overview of Required System for *FlexiPlus* Accounts

Responsibilities
a. upload and validate new BACS transactions/delete BACS transactions
b. download BACS payment file to BestBank's server
c. download BACS payment file to BACS client-side software
d. update FlexiPlus database
e. confirm BACS transactions
f. display acceptance or rejection of new BACS transaction or confirm deletion of old BACS transaction
g. request VISA payment
h. query database and update database if there are enough funds on account
i. send accept or reject (VISA transaction) response to other bank's server
j. request transaction
k. query database and update database if required
l. send accept/reject response
m. query database and update database if required
n. display confirmation or rejection of transaction
o. upload and validate withdrawal from *FlexiPlus* A/c
p. send acceptance or rejection
q. display rejection
r. debit *FlexiPlus* A/c
s. display confirmation of withdrawal
t. download and validate deposit to *FlexiPlus* A/c
u. send rejection
v. credit *FlexiPlus* A/c
w. display confirmation of deposit
y. send acceptance

Documentation of Figure 4.2: UCM – Overview of Required System for *FlexiPlus* Accounts

Responsibilities a to f
Members of *BestBank's* staff update the BACS files and upload them to Voca Limited's Server. An input report is the sent back to *BestBank* from Voca's server confirming that the BACS files have been received. Payment files are then sent from Voca's server to *BestBank's* server and the payment files are then validated by *BestBank's* staff. Items are then applied to customer's accounts, i.e. the FAMSDB database is updated, and confirmation that the transactions have been successfully processed is given to *BestBank's* staff.

Responsibilities g to i
When a *FlexiPlus* customer uses their Visa Debit card it is firstly swiped in an EPOS terminal. A request for payment is then sent to the Acquiring Bank's server (Other Bank's Servers) which passes it to *BestBank's* server. The customer's account is then checked to see if there are sufficient funds to pay the Acquiring bank, i.e. the FAMSDB database is queried and updated if there are enough funds available. Then an Acceptance code or Rejection code is sent to the EPOS terminal via the Acquiring bank's server, i.e. the transaction is either accepted or rejected.

Responsibilities j to l
When a FlexiPlus customer puts their card in an ATM and requests, e.g. cash, the request is sent from the ATM to the Acquiring bank's server (the ATM Host's server) which then sends it to LINK's server which in turn sends it to *BestBank's* server. The customer's account is then checked, i.e. the FAMSDB database is queried and the transaction is either accepted or rejected, i.e. an Acceptance code or a Rejection code is then sent to the ATM via LINK's server and the Acquiring bank's server.

Responsibilities n to m
When a FlexiPlus customer goes into a *BestBank* branch and, for example, asks to withdraw money the Bank teller will check to see if the customer has enough funds, i.e. they will query the FAMSSDB database. The FAMS application will then indicate if there are enough funds to the teller who will then issue the money or reject the customer's request.

Responsibilities o to s
When a FlexiPlus customer request that a CHAPS payment be made payment details (receiving banks sort code and receiving account number, etc) are sent to the Bank of England who will either accept or reject the transaction.

If they accept it, an Acceptance code is sent from the Bank of England to *BestBank*, the customers account is debited, i.e. the FAMSDB database is updated, and confirmation of the successful transaction is displayed to *BestBank's* staff.

If they reject it, a Rejection code is sent from the Bank of England to *BestBank* and confirmation of the rejection is displayed to *BestBank's* staff.

Responsibilities t to y
CHAPS payments will be received by *FlexiPlus* customers in the following way: notification of the CHAPS payment will be received by *BestBank's* staff who will either accept or reject it.

If *BestBank's* staff accept the payment, the *FlexiPlus* account is credited, i.e. the FAMSDB database is updated, and an Acceptance code is sent back to the Bank of England. Confirmation of the accepted payment will be displayed to *BestBank's* staff.

If *BestBank's* staff reject the payment, a Rejection code is sent back to the Bank of England.

4.3 3-Tier Structure of the FAMS Application

In section 3.2 it is given that *BestBank* holds all of its data on its HP NonStop SQL Database which sits on *BestBank's* HP NonStop Server at Head Office in Paisley. It follows that the FAMS application will require a 3-tier structure.

The first tier is the presentation logic layer. As the FAMS application is to be a Visual Basic .NET application this layer will be implemented with Forms and Controls.

The second lay is the business logic layer which will deal with creating a set of business objects and imposing business rules on them.

The third layer is the data access layer which is responsible for establishing a connection to the FAMSDB database and getting information to and from the FAMSDB database. This function is carried out by the ADO.NET Application Programming Interface.

The 3-tier structure of the FAMS application is illustrated in the component diagram shown in Figure 4.3 below.

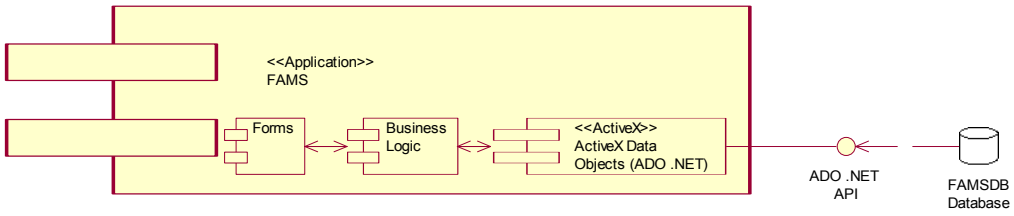


Figure 4.3: 3-Tier Structure of the FAMS Application

Chapter 5

Analysis & Design of the FAMSDB Database

Note that as the FAMSDB Database is being designed for implementation with Microsoft Jet 4.0 and Microsoft Access 2002 rather than *BestBank's* enterprise level HP NonStop SQL database, the design may not be optimal.

5.1 Conceptual Design

The conceptual schema and Global ER Diagram below have had all the features that cannot be represented directly in the relational model removed, i.e. the following have been removed from the model:

- many-to-many (*:*) binary relationships
- many-to-many (*:*) recursive relationships
- complex relationships
- multi-valued attributes.

This has been done in order to simplify the Logical Design in section 5.2.

5.1.1 Conceptual Schema - Entities

Staff {staffNo, forenames, surname, username, password, localIPAddress}

Primary Key: staffNo
Alternate Key: username
Attributes are all atomic attributes.

Customer {customerNo, name(title, forenames, surname), address(addressLine1, addressLine2, addressLine3, city, county), postcode, telephoneNo, emailAddress, DoB, DoD}

Primary Key: customerNo
Atomic attributes: customerNo, postcode, telephoneNo, emailAddress, DoB, DoD
Composite attributes: name(title, forenames, surname), address(addressLine1, addressLine2, addressLine3, city, county)

Account {sortCode, accountNo, availableBalance, balance, overdraftLimit, status, openingDate, closingDate, totalInterest, charges, notes, productType, name, addressLine1, addressLine2, addressLine3, city, county, postCode}

Partial Primary Keys: sortCode, accountNo
Attributes are all atomic attributes.

Transaction {sortCode, accountNo, transactionNo, date, description, credit, debit, balance}

Partial Primary Keys: sortCode, accountNo, transactionNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)
Attributes are all atomic attributes.

Card {cardNo, name(title, forenames, surname), pinNo, startDate, expiryDate, ccvNo, issueNo, sortCode, accountNo, cardValid, withdrawnToday, dailyWithdrawalLimit}

Primary Key: cardNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)
Atomic attributes: cardNo, pinNo, startDate, expiryDate, ccvNo, issueNo, sortCode, accountNo, cardValid, withdrawnToday, dailyWithdrawalLimit
Composite attributes: name(title, forenames, surname)

CustomerMaintenance {transactionNo, date, action}

No Primary Key
Weak Entity
Owner Entities: Staff, Customer
Attributes are all atomic attributes

AccountMaintenance {transactionNo, date, action}

No Primary Key
Weak Entity
Owner Entities: Staff, Account
Attributes are all atomic attributes

AccountHolder {customerNo1, customerNo2, customerNo3}

No Primary Key
Weak Entity
Owner Entities: Customer, Account
Foreign Keys: customerNo1, customerNo2, customerNo3
Foreign Keys Reference: Customer(customerNo)
Attributes are all atomic attributes

5.1.2 Conceptual Schema - Relationships.

Makes	{(Staff, Transaction), (0..*), (1..1)}	Cardinality Ratio: (1:*)
Requires	{(Customer, Transaction), (1..*), (1..1)}	Cardinality Ratio: (1:*)
Conducts	{(Account, Transaction), (1..*), (1..1)}	Cardinality Ratio: (1:*)
Is	{(Customer, AccountHolder), (1..*), (1..1)}	Cardinality Ratio: (1..*)
HeldBy	{(Account, AccountHolder), (1..1), (1..1)}	Cardinality Ratio: (1..1)
Has	{(Customer, Card), (1..*), (1..1)}	Cardinality Ratio: (1:*)
AccessedBy	{(Account, Card), (1..3), (1..1)}	Cardinality Ratio: (1:3)
Provides	{(Staff, CustomerMaintenance), (0..*), (1..1)}	Cardinality Ratio: (1..*)
Uses	{(Customer, CustomerMaintenance), (1..*), (1..1)}	Cardinality Ratio: (1..*)
CarriesOut	{(Staff, AccountMaintenance), (0..*), (1..1)}	Cardinality Ratio: (1..*)
Needs	{Account, AccountMaintenance), (1..*), (1..1)}	Cardinality Ratio: (1..*)

5.1.3 Global Entity-Relationship Diagram

The conceptual schema above yields the Global ER Diagram shown in Figure 5.1 below.

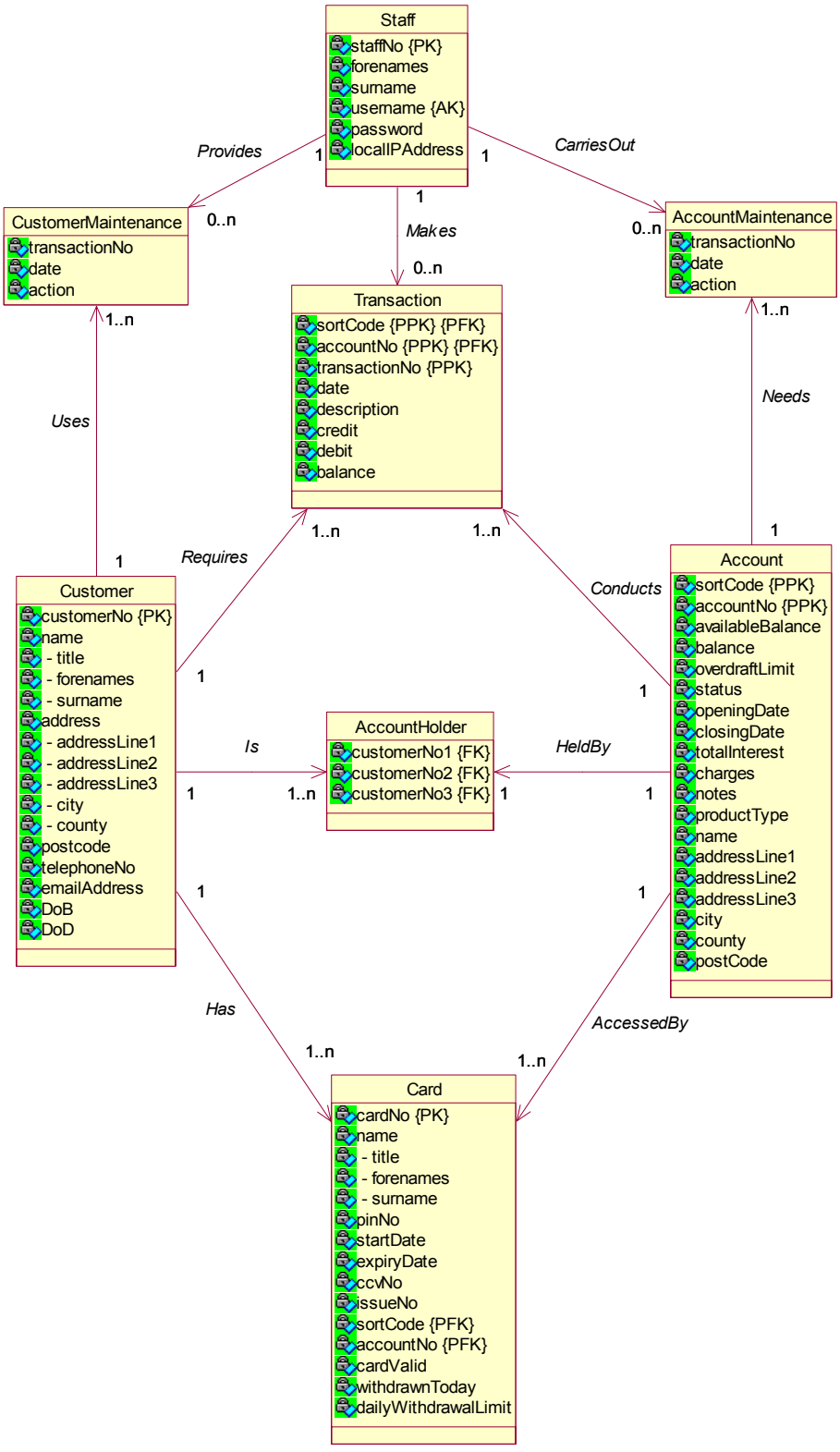


Figure 5.1: Global Entity-Relationship Diagram

Documentation of Figure 5.1: Global Entity-Relationship Diagram

Under normal circumstances *FlexiPlus* accounts will be held by either single adults or jointly by two adults (usually partners). However, exceptionally, a third customer may be named on a *FlexiPlus* account when, for example, a joint account holder has given a third customer Power of Attorney over their affairs or two parents have named a child as a third customer.

The attribute “status” of entity type “Account” can hold the values “Open”, “Student”, “Deceased” or “Closed”.

The attribute “description” of entity type “Transaction” can hold the values “Cash”, “Cheque”, “Transfer”, “Banker’s Draft”, “Bill Payment”, “Interest” or “Overdraft Charge”.

The attribute “cardValid” of entity type “Card” can hold the values “Yes” or “No”.

The attributes “updated” of the entity types “CustomerMaintenance” and “AccountMaintenance” contain the name of the attribute in the “Customer” and “Account” relations, respectively, that was updated by the member of staff and the “date” attributes of the entity types “CustomerMaintenance” and “AccountMaintenance” contain the date that the attribute in the “Customer” and “Account” relations, respectively, was changed. The data in these relations will be archived after two years, i.e. when it becomes two years old, by the Database Administrator.

Other attribute domains are as would be expected.

5.2 Logical Design

5.2.1 Mapping Entities and their Attributes – Strong Entities

Staff {staffNo, forenames, surname, username, password, localIPAddress}

Primary Key: staffNo
Alternate Key: username

Customer {customerNo, title, forenames, surname, addressLine1, addressLine2, addressLine3, city, county, postcode, telephoneNo, emailAddress, DoB, DoD}

Primary Key: customerNo

Account {sortCode, accountNo, availableBalance, balance, overdraftLimit, status, openingDate, closingDate, totalInterest, charges, notes, productType, name addressLine1, addressLine2, addressLine3, city, county, postCode }

Partial Primary Keys: sortCode, accountNo

Transaction {sortCode, accountNo, transactionNo, date, description, credit, debit, balance}

Partial Primary Keys: sortCode, accountNo, transactionNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)

Card {cardNo, title, forenames, surname, pinNo, startDate, expiryDate, ccvNo, issueNo, sortCode, accountNo, cardValid, withdrawnToday, dailyWithdrawalLimit}

Primary Key: cardNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)

5.2.2 Mapping Relationships and their Attributes – One-to-Many (1..*) Binary Relationships

Makes {(Staff, Transaction), (0..*), (1..1)}

Transaction {sortCode, accountNo, transactionNo, date, description, credit, debit, balance, staffNo}

Primary Keys: sortCode, accountNo, transactionNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)
Foreign Key: staffNo References: Staff(staffNo)

Requires {(Customer, Transaction), (1..*), (1..1)}

Transaction {sortCode, accountNo, transactionNo, date, description, credit, debit, balance, staffNo, customerNo}

Primary Keys: sortCode, accountNo, transactionNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)
Foreign Key: staffNo References: Staff(staffNo)
Foreign Key: customerNo References: Customer(customerNo)

Conducts {(Account, Transaction), (1..*), (1..1)}

Transaction {sortCode, accountNo, transactionNo, date, description, credit, debit, balance, staffNo, customerNo}

Primary Keys: sortCode, accountNo, transactionNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)
Foreign Key: staffNo References: Staff(staffNo)
Foreign Key: customerNo References: Customer(customerNo)

Primary Key of Account relation is already included as attributes of Transaction relation.

AccessedBy {(Account, Card), (1..3), (1..1)}

Card {cardNo, title, forename, surname, pinNo, startDate, expiryDate, ccvNo, issueNo, sortCode, accountNo, cardValid, withdrawnToday, dailyWithdrawalLimit}

Primary Key: cardNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)

Primary Key of Account relation is already included as attributes of Card relation.

Has {(Customer, Card), (1..*), (1..1)}

Card {cardNo, title, forename, surname,pinNo, startDate, expiryDate, ccvNo, issueNo, sortCode, accountNo, cardValid, withdrawnToday, dailyWithdrawalLimit, customerNo}

Primary Key: cardNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)
Foreign Key: customerNo References: Customer(customerNo)

Provides {(Staff, CustomerMaintenance), (0..*), (1..1)}

CustomerMaintenance {staffNo, transactionNo, date, action}

Uses {(Customer, CustomerMaintenance), (1..*), (1..1)}

CustomerMaintenance {staffNo, customerNo, transactionNo, date, action}

Partial Primary Keys: staffNo, customerNo, transactionNo
Foreign Key: staffNo References: Staff(staffNo)
Foreign Key: customerNo References: Customer(customerNo)

CarriesOut {(Staff, AccountMaintenance), (0..*), (1..1)}

AccountMaintenance {staffNo, transactionNo, date, action}

Needs {(Account, AccountMaintenance), (1..*), (1..1)}

AccountMaintenance {sortCode, accountNo, staffNo, transactionNo, date, action}

Partial Primary Keys: sortCode, accountNo, staffNo, transactionNo
Partial Foreign Keys: sortCode, accountNo, staffNo
Partial Foreign Keys Reference: Account(sortCode, accountNo) and Staff(staffNo)

Is {(Customer, AccountHolder), (1..*), (1..1)}

AccountHolder already has reference to Customer(customerNo)

5.2.3 Mapping Relationships and their Attributes – One-to-One (1..1) Binary Relationships

HeldBy {(Account, AccountHolder), (1..1), (1..1)}

As participation on both sides of the relation is mandatory, we combine the Account and AccountHolder entities to give the relation Account as follows:

Account {sortCode, accountNo, availableBalance, balance, overdraftLimit, status, openingDate, closingDate, totalInterest, charges, notes, productType, customerNo1, customer No2, customerNo3, name, addressLine1, addressLine2, addressLine3, city, county, postCode}

5.2.4 Relational Schema

Staff {staffNo, forenames, surname, username, password, localIPAddress}

Primary Key: staffNo
Alternate Key: username

Transaction {sortCode, accountNo, transactionNo, date, description, credit, debit, balance, staffNo, customerNo}

Primary Keys: sortCode, accountNo, transactionNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)
Foreign Key: staffNo References: Staff(staffNo)
Foreign Key: customerNo References: Customer(customerNo)

Customer {customerNo, title, forenames, surname, addressLine1, addressLine2, addressLine3, city, county, postcode, telephoneNo, emailAddress, DoB, DoD}

Primary Key: customerNo

Account {sortCode, accountNo, availableBalance, balance, overdraftLimit, status, openingDate, closingDate, totalInterest, charges, notes, productType, customerNo1, customerNo2, customerNo3, name, addressLine1, addressLine2, addressLine3, city, county, postCode}

Partial Primary Keys: sortCode, accountNo

Card {cardNo, title, forename, surname, pinNo, startDate, expiryDate, ccvNo, issueNo, sortCode, accountNo, cardValid, withdrawnToday, dailyWithdrawalLimit, customerNo}

Primary Key: cardNo
Partial Foreign Keys: sortCode, accountNo
Partial Foreign Keys Reference: Account(sortCode, accountNo)
Foreign Key: customerNo References: Customer(customerNo)

CustomerMaintenance {staffNo, customerNo, transactionNo, date, action}

Partial Primary Keys: staffNo, customerNo, transactionNo
Foreign Key: staffNo References: Staff(staffNo)
Foreign Key: customerNo References: Customer(customerNo)

AccountMaintenance {sortCode, accountNo, staffNo, transactionNo, date, action}

Partial Primary Keys: sortCode, accountNo, staffNo, transactionNo
Partial Foreign Keys: sortCode, accountNo, staffNo
Partial Foreign Keys Reference: Account(sortCode, accountNo) and Staff(staffNo)

5.2.5 Normalization

In this subsection we validate the relational schema by checking that each relation is in Boyce-Codd Normal Form (BCNF).

5.2.5.1 First Normal Form (1NF)

As there are no multi-valued attributes in any of the relations in the relational schema given in subsection 5.2.3 above, all the relations are in 1NF.

5.2.5.2 Second Normal Form (2NF)

The following relations in the relational schema are in 2NF as they are in 1NF and all have a single attribute for their primary key:

- Staff {staffNo, forenames, surname, username, password, localIPAddress}
- Customer {customerNo, title, forenames, surname, addressLine1, addressLine2, addressLine3, city, county, postcode, telephoneNo, emailAddress, DoB, DoD}
- Card {cardNo, title, forename, surname, pinNo, startDate, expiryDate, ccvNo, issueNo, sortCode, accountNo, cardValid, withdrawnToday, dailyWithdrawalLimit, customerNo}

The following relations in the relational schema are in 2NF as they are in 1NF and each of their non-primary key attributes is fully functionally dependent on the primary key:

- Transaction {sortCode, accountNo, transactionNo, date, description, credit, debit, balance, staffNo, customerNo}

- Account {sortCode, accountNo, availableBalance, balance, overdraftLimit, status, openingDate, closingDate, totalInterest, charges, notes, productType, customerNo1, customerNo2, customerNo3, name, addressLine1, addressLine2, addressLine3, city, county, postCode}
- CustomerMaintenance {staffNo, customerNo, transactionNo, date, action}
- AccountMaintenance {sortCode, accountNo, staffNo, transactionNo, date, action}

Thus, the relational schema is in 2NF.

5.2.5.3 Third Normal Form (3NF)

The following relations in the relational schema are in 3NF as they are in 1NF and 2NF and have no transitive dependencies on their primary keys:

- Transaction {sortCode, accountNo, transactionNo, date, description, credit, debit, balance, staffNo, customerNo}
- Account {sortCode, accountNo, availableBalance, balance, overdraftLimit, status, openingDate, closingDate, totalInterest, charges, notes, productType, customerNo1, customerNo2, customerNo3, name, addressLine1, addressLine2, addressLine3, city, county, postCode}
- Staff {staffNo, forenames, surname, username, password, localIPAddress}
- Customer {customerNo, title, forenames, surname, addressLine1, addressLine2, addressLine3, city, county, postcode, telephoneNo, emailAddress, DoB, DoD}
- Card {cardNo, title, forename, surname, pinNo, startDate, expiryDate, ccvNo, issueNo, sortCode, accountNo, cardValid, withdrawnToday, dailyWithdrawalLimit, customerNo}
- CustomerMaintenance {staffNo, customerNo, transactionNo, date, action}
- AccountMaintenance {sortCode, accountNo, staffNo, transactionNo, date, action}

Thus, the relational schema is in 3NF.

5.2.5.4 Boyce-Codd Normal Form (BCNF)

The following relations in the relational schema are in BCNF as they are in 1NF, 2NF and 3NF and every determinant is a candidate key:

- Transaction {sortCode, accountNo, transactionNo, date, description, credit, debit, balance, staffNo, customerNo}
- Account {sortCode, accountNo, availableBalance, balance, overdraftLimit, status, openingDate, closingDate, totalInterest, charges, notes, productType, customerNo1, customerNo2, customerNo3, name, addressLine1, addressLine2, addressLine3, city, county, postCode}
- Staff {staffNo, forenames, surname, username, password, localIPAddress}
- Customer {customerNo, title, forenames, surname, addressLine1, addressLine2, addressLine3, city, county, postcode, telephoneNo, emailAddress, DoB, DoD}
- Card {cardNo, title, forename, surname, pinNo, startDate, expiryDate, ccvNo, issueNo, sortCode, accountNo, cardValid, withdrawnToday, dailyWithdrawalLimit, customerNo}
- CustomerMaintenance {staffNo, customerNo, transactionNo, date, action}
- AccountMaintenance {sortCode, accountNo, staffNo, transactionNo, date, action}

Thus, the relational schema is in BCNF.

5.3 Physical Design

"Table 5.1: Tables, Fields and their constraints" below contains details of the data types of the attributes (fields) in each relation (table) of the FAMSDB database along with any enterprise constraint imposed on the data.

Table	Field Name	Data Type	Field Size	Format	Decimal Places	Input Mask	Caption	Default Value	Validation Rule	Validation Text	Required	Allow Zero Length	Indexed
Staff	staffNo	Text	50				Staff No.				Yes	No	Yes (No Duplicates)
Staff	forenames	Text	50				Forenames				No	No	No
Staff	surname	Text	50				Surname				No	No	No
Staff	username	Text	50				Username				No	No	No
Staff	Password	Text	50			Pass-word	Password				No	No	No
Staff	localIPAddress	Text	50				Local IP Address				No	Yes	No
Transaction	sortCode	Text	50				Sort Code				Yes	No	No
Transaction	accountNo	Text	50				Account No.				Yes	No	No
Transaction	transactionNo	Text	50				Transaction No.				Yes	No	No
Transaction	date	Text	50				Date				No	Yes	No
Transaction	description	Text	50				Description		Must be one of Cash, Cheque, Transfer, Banker's Draft, Bill Payment, Interest or Overdraft Charge	Invalid Description	Yes	No	No
Transaction	Credit	Text	50				Credit				No	Yes	No
Transaction	debit	Text	50				Debit				No	Yes	No
Transaction	balance	Text	50				Balance				No	Yes	No
Transaction	staffNo	Text	50				Staff No.				No	Yes	No
Transaction	customerNo	Text	50				Customer No.				No	Yes	No
Customer	customerNo	Text	50				Customer No.				Yes	No	Yes (No Duplicates)
Customer	title	Text	50				Title				No	Yes	No
Customer	forenames	Text	50				Forenames				No	Yes	No
Customer	surname	Text	50				Surname				Yes	No	No
Customer	addressLine1	Text	50				Address Line 1				No	Yes	No
Customer	addressLine2	Text	50				Address Line 2				No	Yes	No
Customer	addressLine3	Text	50				Address Line 3				No	Yes	No
Customer	city	Text	50				City				No	Yes	No
Customer	county	Text	50				County				No	Yes	No
Customer	postcode	Text	50				Postcode				No	Yes	No
Customer	telephoneNo	Text	50				Telephone No.				No	Yes	No
Customer	emailAddress	Text	50				Email Address				No	Yes	No
Customer	DoB	Text	50				Date of Birth				No	Yes	No
Customer	DoD	Text	50				Date of Death				No	Yes	No
Account	sortCode	Text	50				Sort Code				Yes	No	No
Account	accountNo	Text	50				Account No.				Yes	No	No
Account	availableBalance	Text	50				Available Balance				No	Yes	No

Table	Field Name	Data Type	Field Size	Format	Decimal Places	Input Mask	Caption	Default Value	Validation Rule	Validation Text	Required	Allow Zero Length	Indexed
Account	balance	Text	50				Balance				No	Yes	No
Account	overdraftLimit	Text	50				Overdraft Limit				No	Yes	No
Account	Status	Text	50				Status		“Open”, “Closed”, “Student” Or “Deceased”	Invalid Status	No	Yes	No
Account	openingDate	Text	50				Opening Date				No	Yes	No
Account	ClosingDate	Text	50				Closing Date				No	Yes	No
Account	totalInterest	Text	50				Total Interest				No	Yes	No
Account	charges	Text	50				Charges				No	Yes	No
Account	notes	Memo					Notes				No	Yes	No
Account	productType	Text	50				Product Type				No	Yes	No
Account	customerNo1	Text	50				Customer No. 1				No	Yes	No
Account	customerNo2	Text	50				Customer No. 2				No	Yes	No
Account	customerNo3	Text	50				Customer No. 3				No	Yes	No
Account	name	Text	50				Name				No	Yes	No
Account	addressLine1	Text	50				Address Line 1				No	Yes	No
Account	addressLine2	Text	50				Address Line 2				No	Yes	No
Account	addressLine3	Text	50				Address Line 3				No	Yes	No
Account	city	Text	50				City				No	Yes	No
Account	county	Text	50				County				No	Yes	No
Account	postcode	Text	50				Post Coode				No	Yes	No
Card	cardNo	Text	50				Card No.				Yes	No	Yes (No Duplicates)
Card	title	Text	50				Title				No	Yes	No
Card	forenames	Text	50				Forenames				No	Yes	No
Card	Surname	Text	50				Surname				No	Yes	No
Card	pinNo	Text	50				Pin No.				No	Yes	No
Card	startDate	Text	50				Start Date				No	Yes	No
Card	expiryDate	Text	50				Expiry date				No	Yes	No
Card	ccvNo	Text	50				CCV No.				No	Yes	No
Card	issueNo	Text	50				Issue No.				No	Yes	No
Card	sortCode	Text	50				Sort Code				No	Yes	No
Card	accountNo	Text	50				Account No.				No	Yes	No
Card	cardValid	Text	50				Card Valid				No	Yes	No
Card	withdrawnToday	Text					Withdrawn Today				No	Yes	No
Card	dailyWithdrawalLimit	Text					Daily Withdrawal Limit				No	Yes	No
Card	customerNo	Text	50				Customer No.				No	Yes	No
Customer-Maintenance	staffNo	Text	50				Staff No.				Yes	No	No
Customer-Maintenance	customerNo	Text	50				Customer No.				Yes	No	No
Customer-Maintenance	transactionNo	Text	50				Transaction No.				Yes	No	No
Customer-Maintenance	Date	Text	50				Date				No	Yes	No

Table	Field Name	Data Type	Field Size	Format	Decimal Places	Input Mask	Caption	Default Value	Validation Rule	Validation Text	Required	Allow Zero Length	Indexed
Customer-Maintenance	action	Text	50				Action				No	Yes	No
Account-Maintenance	sortCode	Text	50				Sort Code				Yes	No	No
Account-Maintenance	accountNo	Text	50				Account No.				Yes	No	No
Account-Maintenance	staffNo	Text	50				Staff No.				Yes	No	No
Account-Maintenance	transactionNo	Text	50				Transaction No.				Yes	No	No
Account-Maintenance	Date	Text	50				Date				No	Yes	No
Account-Maintenance	action	Text	50				Action				No	Yes	No

Table 5.1: Tables, Fields and their constraints

Note that all fields in the database are of type “Text” with the exception of the “Notes” field in the account table which is of type “Memo”. This is to prevent type mismatches when updating the database from the FAMS application.

5.4 Scheduled Tasks carried out by the DBA.

5.4.1 Calculating Interest
Each night all *FlexiPlus* accounts which are not overdrawn would have that day's interest calculated and added to the "totalInterest" field of their record in the "Account" table. The following code would be used:

Account.totalInterest = Account.totalInterest + (InterestRate/365)*Account.availableBalance

5.4.2 Monthly Interest Posting
At 12 midnight on the last day of each month the value of the totalInterest field would be added to the "availableBalance" and the "balance" fields of the relevant records.

5.4.3 Calculating Overdraft charges
Each night all *FlexiPlus* accounts which are overdrawn and do not have Student status would have that days overdraft charge calculated and added to the "charges" field of their record in the "Account" table. The following code would be used:

Account.charges = Account.charges + (ChargesRate/365)*(-Balance)

5.4.4 Monthly Overdraft Charges Posting
At the end of each month, each *FlexiPlus* account would have the value of the "charges" field of their record deducted from the "availableBalance" and "balance" fields of their record in the Account table.

Chapter 6

Analysis & Design of the FAMS Application with UML

6.1 Use Case Diagram

Figure 6.1 below illustrates the functional requirements of the *FlexiPlus* system. Note that, it is supposed that *BestBank* were consulted during the construction of the UCD in Figure 6.1.

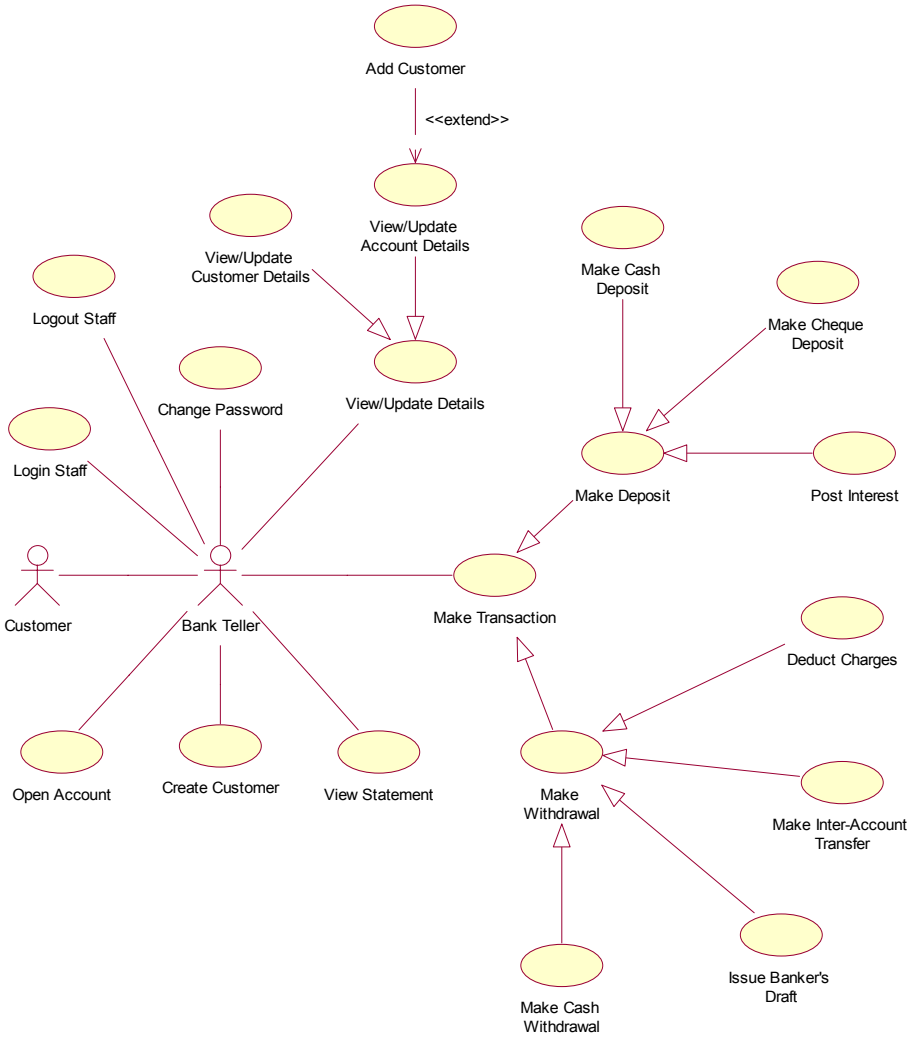


Figure 6.1: Use Case Diagram for FAMS Application

6.2 Use Case Descriptions (Scenarios)

6.2.1 "Login Staff" Description

User Opens FAMS Application

Form frmLogin opens

User Enters Username and Password

User Clicks “Login” button in Form frmLogin

If Username <> “” AND Password <> “” Then

Open Database connection

Clear DataSet (virtual Database)

SELECT all
FROM Staff
WHERE username = ‘Username’ AND password = ‘Password’

If record exists Then

Open Form frmMainMenu
Set Staff.localIPAddress = IP Address of computer that the user has logged in on
Hide Form frmLogin
Close Database Connection

Else

Close Database Connection
Display “User Not Found” Warning Message

End If

Else

Display “Must Enter Username and Password” Error Message

End If

6.2.2 “Logout Staff” Description

User Clicks “Logout” in Form frmMainMenu

Call mnuLogout_Click()

Open Database connection

Set Staff.localIPAddress = NULL (So IT Department can tell user is not logged into FAMS)
Close Database Connection
Call Garbage Collector
Exit Application, i.e. close form frmMainMenu and Hidden Form frmLogin + any other open forms

End Call to mnuLogout_Click()

6.2.3 “Change Password” Description

User Clicks “Change Password” in frmMainMenu

Open Form frmChangePassword

```
User Enters “Old Password” text in frmChangePassword
User Enters “New password” text in frmChangePassword
User Enters “Confirm New Password” text in frmChangePassword

User Clicks “Change Password” in frmChangePassword

Display “Are you sure you want to change your Password?” Question Message

If answer is YES then

        If (“Old Password” <> “” AndAlso “New Password” <> “”   _
        AndAlso <> “Confirm New Password” <> “”) Then

                Open Database connection

                Clear DataSet (virtual Database)

                SELECT all
                From Staff
                WHERE localIPAddress = IP Address of machine that the user is logged in on

                If “Old Password” = Staff.password in selected record Then

                        If “New Password” = “Confirm New Password” Then

                                Set Staff.password = “New Password”
                                Close Database Connection
                                Close Form frmChangePassword
                                Display “Password Changed” Information Message

                        Else

                                Close Database Connection
                                Display “New Passwords Must Match” Error Message

                        End If

                Else

                        Close Database Connection
                        Display “Old Password Incorrect” Warning Message

                End If

        Else

                Display “Must Enter Old and New Passwords” Error Message

        End If

End If
```

6.2.4 “Open Account” Description

User Enters branch “Sort Code” in Form frmMainMenu (integer in range 990000 to 990271 as there are 272 branches and *BestBank’s* sort code prefix is 99)

User Clicks “Create Account” Button in Form frmMainMenu

Display “Are you sure you want to Create New Account?” Question Message

If Answer is Yes Then

Call NewAccountNo() method

Declare accountNumber variable

accountNumber = Randomly generated number in the range 10000000 to 99999999

Open Database connection

Clear DataSet (virtual Database)

SELECT all
FROM Account
WHERE sortCode = “Sort Code” AND accountNo = ‘accountNumber’

Close Database Connection

If record exists Then

Recursively call NewAccountNo()

Else

Open Database Connection

INSERT INTO Account(sortCode, accountNo, customerNo1, customerNo2, customerNo3, availableBalance, balance, overdraftLimit, openingDate, closingDate, totalInterest, charges)
VALUES(“Sort Code”, accountNumber, “”, “”, “”, “0”, “0”, “0”, Today’s Date, “”, “0”, “0”)

Close Database Connection

Display accountNumber in TextBox in Form frmMainMenu

Open Form frmAccountDetails

End If

End Call to NewAccountNo() method

‘ Update AccountMaintenance Table

Open Database Connection

Clear DataSet (virtual Database)

SELECT staffNo
FROM Staff
WHERE localIPAddress = IP Address of machine that the member of staff if logged in on

INSERT INTO AccountMaintenance
VALUES(sortCode, accountNo, staffNo, “1”, Today’s Date, “Created”)

Close Database Connection

‘ End Update of AccountMaintenance Table

End If

6.2.5 “Create Customer” Description

User Clicks “Create Customer” Button in frmMainMenu

Display “Are you sure you want to Create New Customer?” Question Message

If Answer is Yes Then

Call NewCustomerNo() method

Declare customerNumber variable

customerNumber = Randomly generated Integer in range 1 to 2,147,483,647

Open Database connection

Clear DataSet (virtual Database)

SELECT all
FROM Customer
Where CustomerNo = ‘customerNumber’

Close Database Connection

If record exists Then

Recursively call NewCustomerNo()

Else

Open Database Connection

INSERT INTO Customer(customerNo)
VALUES(customerNumber)

Close Database Connection

Display customerNumber in TextBox in Form frmMainMenu

Open Form frmCustomerDetails

End If

End Call to NewCustomerNo() method

‘Update CustomerMaintenance Table

Open Database Connection

Clear DataSet (virtual Database)

SELECT staffNo
FROM Staff
WHERE localIPAddress = IP Address of machine that the member of staff is logged in on

INSERT INTO CustomerMaintenance
VALUES(staffNo, CustomerNumber, “1”, Today’s Date, “Created”)

Close Database Connection

‘End Update CustomerMaintenance Table

End If

6.2.6 “View/Update Customer Details” Description

User Enters “Customer Number”

User Clicks “View Customer” in frmMainMenu

Open Database connection

Clear DataSet (virtual database)

SELECT all
FROM Customer
WHERE customerNo = “Customer Number”

Close Database Connection

If record exists Then

Open Form frmCustomerDetails

Call frmCustomerDetails_Load() method

Open Database Connection

Clear DataSet (virtual database)

SELECT all
FROM Customer
WHERE customerNo = “Customer Number”

Close Database Connection

Set TextBox txtCustomerNo Text = Customer.customerNo

```
                Set TextBox txtTitle Text = Customer.title
                Set TextBox txtForenames Text = Customer.forenames
                Set TextBox txtSurname Text = Customer.surname
                Set TextBox txtAddressLine1 Text = Customer.addressLine1
                Set TextBox txtAddressLine2 Text = Customer.addressLine2
                Set TextBox txtAddressLine3 Text = Customer.addressLine3
                Set TextBox txtCity Text = Customer.city
                Set TextBox txtCounty Text = Customer.county
                Set TextBox txtPostcode Text = Customer.postcode
                Set TextBox txtTelephoneNo Text = Customer.telephoneNo
                Set TextBox txtEmailAddress Text = Customer.emailAddress
                Set TextBox txtDoB Text = Customer.DoB
                Set TextBox txtDoD Text = Customer.DoD

                'Now display Accounts Held by Customer

                Open Database Connection

                Clear DataSet (virtual Database)

                SELECT sortCode, accountNo, productType
                FROM Account
                WHERE (customerNo1 = "Customer Number" in frmMainMenu  OrElse & _
                        customerNo2 = "Customer Number" in frmMainMenu  OrElse & _
                        customerNo3 = "Customer Number" in frmMainMenu )

                Close Database Connection

                Display results of query in DataGrid Control dgdAccountsHeld

        End Call to frmCustomerDetails_Load() method

Else
        Display "Invalid Customer Number" Error Message

End If

If User Clicks "Update details" in Form frmCustomerDetails

        Display "Are you sure you want to Update the Customer's record?" Question Message

        If Answer is Yes Then

                Call mnuUpdateDetails_Click() method

                        Open Database Connection

                        Set Customer.title = TextBox txtTitle Text
                        Set Customer.forenames = TextBox txtForenames Text
                        Set Customer.surname = TextBox txtSurname Text
                        Set Customer.addressLine1 = TextBox txtAddressLine1 Text
                        Set Customer.addressLine2 = TextBox txtAddressLine2 Text
                        Set Customer.addressLine3 = TextBox txtAddressLine3 Text
                        Set Customer.city = TextBox txtCity Text
                        Set Customer.county = TextBox txtCounty Text
                        Set Customer.postcode = TextBox txtPostcode Text
```

Set Customer.telephoneNo = TextBox txtTelephoneNo Text
Set Customer.emailAddress = TextBox txtEmailAddress Text
Set Customer.DoB = TextBox txtDoB Text
Set Customer.DoD = TextBox txtDoD Text

Close Database Connection

'Update CustomerMaintenance Table

Declare staffNumber and transactionNumber variables

Open Database Connection

Clear DataSet (virtual Database)

Set StaffNumber = SELECT staffNo FROM Staff
WHERE localIPAddress = IP Address of machine that the member of staff is logged in on

Clear DataSet (virtual Database)

Set transactionNumber = SELECT MAX(transactionNo)
FROM CustomerMaintenance
WHERE StaffNo = StaffNumber AND CustomerNo = "Customer Number"

INSERT INTO CustomerMaintenance
VALUES(staffNumber, "Customer Number", "transactionNumber + 1", Today's Date, "Updated")

Close Database Connection

'End Update CustomerMaintenance Table

End Call to mnuUpdateDetails() method

End If

End If

6.2.7 "View/Update Account Details" Description

User Enters "Sort Code" in Form frmMainMenu

User Enters "Account Number" in Form frmMainMenu

User Clicks "View Account" in Form frmMainMenu

Open Database Connection

Clear DataSet (virtual database)

SELECT all
FROM Account
WHERE sortCode = "Sort Code" AND accountNo = "Account Number"

Close Database Connection

```
If record exists Then

        Open Form frmAccountDetails

        Call frmAccountDetails_Load() method

                If Account.productType <> “” Then

                        Disable ComboBox cboProductType

                End If

                If Account.closingDate <> “” AND Account.status = “Closed”  Then

                        Call DisableAccountDetails() method

                                Disable TextBox txtCustomerNo1
                                Disable TextBox txtCustomerNo2
                                Disable TextBox txtCustomerNo3

                                Disable TextBox txtTitle1
                                Disable TextBox txtTitle2
                                Disable TextBox txtTitle3

                                Disable TextBox txtForenames1
                                Disable TextBox txtForenames2
                                Disable TextBox txtForenames3

                                Disable TextBox txtSurname1
                                Disable TextBox txtSurname2
                                Disable TextBox txtSurname3

                                Disable ComboBox cboAccountStatus

                                Disable TextBox txtOverdraftLimit

                                Disable Menu mnuDeposits
                                Disable Menu mnuWithdrawals
                                Disable Menu mnuUpdateDetails

                        End Call to DisableAccountDetails() method

                End If

                Call DisplayAccountDetails() method

                        Open Database Connection

                        Clear DataSet (virtual database)

                        SELECT all
                        FROM Account
                        WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

                        Close Database Connection
```

```
            Set TextBox txtSortCode Text = Account.sortCode
            Set TextBox txtAccountNo Text = Account.accountNo
            Set ComboBox cboProductType Text = Account.productType
            Set TextBox txtCustomerNo1 Text = Account.customerNo1
            Set TextBox txtCustomerNo2 Text = Account.customerNo2
            Set TextBox txtCustomerNo3 Text = Account.customerNo3
            Set ComboBox cboAccountStatus Text = Account.status
            Set TextBox txtOpeningDate Text = Account.openingDate
            Set TextBox txtClosingDate Text = Account.closingDate
            Set TextBox txtAvailableBalance Text = Account.availableBalance
            Set TextBox txtBalance Text = Account.balance
            Set TextBox txtOverdraftLimit Text = Account.overdrfatLimit
            Set TextBox txtInterestDue Text = Account.totalInterest
            Set TextBox txtChargesDue Text = Account.charges
            Set TextBox txtNotes Text = Account.notes

End Call to DisplayAccountDetails() method

If TextBox txtCustomerNo1 Text <> “” Then

            Call DisplayCustomerNo1() method

                        Disable TextBox txtCustomerNo1
                        Disable TextBox txtTitle1
                        Disable TextBox txtForenames1
                        Disable TextBox txtSurname1

                        Open Database Connection

                        Clear DataSet (virtual database)

                        Select title, forenames, surname
                        FROM Customer
                        WHERE customerNo = TextBox txtCustomerNo1 Text

                        Close Database Connection

                        Set TextBox txtTitle1 Text = Customer.title
                        Set TextBox txtForenames1 Text = Customer.forenames
                        Set TextBox txtSurname1 Text = Customer.surname

            End Call to DisplayCustomerNo1() method

End If

If TextBox txtCustomerNo2 Text <> “” Then

            Call DisplayCustomerNo2() method

                        Disable TextBox txtCustomerNo2
                        Disable TextBox txtTitle2
                        Disable TextBox txtForenames2
                        Disable TextBox txtSurname2

                        Open Database Connection
```

```
                    Clear DataSet (virtual database)

                    Select title, forenames, surname
                    FROM Customer
                    WHERE customerNo = TextBox txtCustomerNo2 Text

                    Close Database Connection

                    Set TextBox txtTitle2 Text = Customer.title
                    Set TextBox txtForenames2 Text = Customer.forenames
                    Set TextBox txtSurname2 Text = Customer.surname

            End Call to DisplayCustomerNo2() method

        End If

        If TextBox txtCustomerNo3 Text <> “” Then

            Call DisplayCustomerNo3() method

                    Disable TextBox txtCustomerNo3
                    Disable TextBox txtTitle3
                    Disable TextBox txtForenames3
                    Disable TextBox txtSurname3

                    Open Database Connection

                    Clear DataSet (virtual database)

                    Select title, forenames, surname
                    FROM Customer
                    WHERE customerNo = TextBox txtCustomerNo3 Text

                    Close Database Connection

                    Set TextBox txtTitle3 Text = Customer.title
                    Set TextBox txtForenames3 Text = Customer.forenames
                    Set TextBox txtSurname3 Text = Customer.surname

            End Call to DisplayCustomerNo3() method

        End If

End Call to frmAccountDetails_Load() method

If User Clicks “Correspondence Address”

        Open Form frmCorrespondenceAddress

        Call frmCorrespondenceAddress_Load() method

            Open Database Connection

            SELECT status, closingDate, name, addressLine1, addressLine2,
            addressLine3, city, county, postcode
```

FROM Account
WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

Close Database Connection

Set TextBox txtName Text = Account.name
Set TextBox txtAddressLine1 Text = Account.addressLine1
Set TextBox txtAddressLine2 Text = Account.addressLine2
Set TextBox txtAddressLine3 Text = Account.addressLine3
Set TextBox txtCity Text = Account.city
Set TextBox txtCounty Text = Account.county
Set TextBox txtPostCode Text = Account.postcode

If Account.closingDate <> “” AND Account.status = “Closed” Then

Call DisableCorrespondenceAddress() method

Disable TextBox txtName
Disable TextBox txtAddressLine1
Disable TextBox txtAddressLine2
Disable TextBox txtAddressLine3
Disable TextBox txtCity
Disable TextBox txtCounty
Disable TextBox txtPostCode

Disable Menu mnuUpdate

End Call to DisableCorrespondenceAddress() method

End If

End Call to frmCorrespondenceAddress_Load() method

If User Clicks “Update” Then

Open Database Connection

Set Account.name = TextBox txtName Text
Set Account.addressLine1 = TextBox txtAddressLine1 Text
Set Account.addressLine2 = TextBox txtAddressLine2 Text
Set Account.addressLine3 = TextBox txtAddressLine3 Text
Set Account.city = TextBox txtCity Text
Set Account.county = TextBox txtCounty Text
Set Account.postcode = TextBox txtPostCode Text

‘Update AccountMaintenance table

Declare staffNumber and transactionNumber variables

Open Database Connection

Clear DataSet (virtual Database)

Set StaffNumber = SELECT staffNo FROM Staff
WHERE localIPAddress = IP Address of machine that the member of staff is logged in on

Clear DataSet (virtual Database)

Set transactionNumber = SELECT MAX(transactionNo)
FROM AccountMaintenance
WHERE StaffNo = StaffNumber AND sortCode = “Sort Code” AND accountNo = “Account Number”

INSERT INTO AccountMaintenance
VALUES(“Sort Code”, “Account Number”, “StaffNumber”
“transactionNumber + 1”, Today’s Date, “Updated Address”)

Close Database Connection

‘End Update AccountMaintenance Table

Close Database Connection

End If User Clicks “Update”

End If User Clicks “Correspondence Address”

If User Clicks “Update details” Then

Display “Are you sure you want to Update the Account’s record?” Question Message

If answer is Yes Then

Open Database Connection

Set Account.productType = ComboBox cboProductType Text
Set Account.customerNo1 = TextBox txtCustomerNo1 Text
Set Account.customerNo2 = TextBox txtCustomerNo2 Text
Set Account.customerNo3 = TextBox txtCustomerNo3 Text
Set Account.status = TextBox txtStatus Text
Set Account.closingDate = TextBox txtClosingDate Text
Set Account.AvailableBalance = (Overdraft Limit + Balance)
Set Account.overdraftLimit = TextBox txtOverdraftLimit Text
Set Account.notes = TextBox txtNotes Text

‘Update AccountMaintenance table

Declare staffNumber and transactionNumber variables

Open Database Connection

Clear DataSet (virtual Database)

Set StaffNumber = SELECT staffNo FROM Staff
WHERE localIPAddress = IP Address of machine that the member of staff is logged in on

Clear DataSet (virtual Database)

Set transactionNumber = SELECT MAX(transactionNo)

FROM AccountMaintenance
WHERE StaffNo = StaffNumber AND sortCode = “Sort Code” AND accountNo = “Account Number”

INSERT INTO AccountMaintenance
VALUES(“Sort Code”, “Account Number”, “StaffNumber” “transactionNumber + 1”, Today’s Date, “Updated Details”)

Close Database Connection

‘End Update AccountMaintenance Table

Close Database Connection

End If

End If User Clicks “Update details”

End If

6.2.8 “Make Transaction” Description

Declare InsertTransaction(Type As String, Amount As String, Balance As String)

Declare staffNumber and transactionNumber variables

Open Database Connection

Clear DataSet (virtual database)

SELECT MAX(transactionNo)
FROM Transactions
WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

If record exists Then

Set transactionNumber = transactionNo + 1

Else

Set transactionNumber = 1

End If

Clear DataSet (virtual Database)

Set StaffNumber =
SELECT staffNo
FROM Staff
WHERE localIPAddress = IP Address of machine that the user is logged in on

Close Database Connection

Select Case Type

Case “CashDeposit”

Open Database Connection

INSERT INTO Transactions
VALUES(“Sort Code”, “Account Number”, transactionNumber, Today’s Date, “Cash”, “Amount”, “”, Balance, StaffNumber, “”)

Close Database Connection

Case “ChequeDeposit”

Open Database Connection

INSERT INTO Transactions
VALUES(“Sort Code”, “Account Number”, transactionNumber, Today’s Date, “Cheque”, “Amount”, “”, Balance, StaffNumber, “”)

Close Database Connection

Case “Interest”

Open Database Connection

INSERT INTO Transactions
VALUES(“Sort Code”, “Account Number”, transactionNumber, Today’s Date, “Interest”, “Amount”, “”, Balance, StaffNumber, “”)

Close Database Connection

Case “Charges”

Open Database Connection

INSERT INTO Transactions
VALUES(“Sort Code”, “Account Number”, transactionNumber, Today’s Date, “Charges”, “”, “Amount”, Balance, StaffNumber, “”)

Close Database Connection

Case “PayTransfer”

Open Database Connection

INSERT INTO Transactions
VALUES(“Sort Code”, “Account Number”, transactionNumber, Today’s Date, “Transfer”, “”, “Amount”, Balance, StaffNumber, “”)

Close Database Connection

Case “ReceiveTransfer”

Open Database Connection

INSERT INTO Transactions
VALUES(“Sort Code”, “Account Number”, transactionNumber, Today’s Date, “Transfer”, “Amount”, “”, Balance, StaffNumber, “”)

Close Database Connection

Case “BankersDraft”

Open Database Connection

INSERT INTO Transactions
VALUES(“Sort Code”, “Account Number”, transactionNumber, Today’s Date, “Banker’s Draft”, “”, “Amount”, Balance, StaffNumber, “”)

Close Database Connection

Case “CashWithdrawal”

Open Database Connection

INSERT INTO Transactions
VALUES(“Sort Code”, “Account Number”, transactionNumber, Today’s Date, “Cash”, “”, “Amount”, Balance, StaffNumber, “”)

Close Database Connection

Case Else

Display Error Message

End Select

6.2.9 “Make Cash Deposit” Description

User Clicks “Deposits” in Form frmAccountDetails

Then, user clicks “Cash” in menu “Deposits” in Form frmAccountDetails

Open Form frmCashDeposit

User Enters “Amount” to be deposited

User Clicks OK

Call OK_Click() method in frmCashDeposit

Declare balance, newBalance, availableBalance and newAvailableBalance variables

Open Database Connection

Clear DataSet (virtual database)

SELECT balance, availableBalance
FROM Account
WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

Close Database Connection

Set balance variable = Account.balance

Set availableBalance variable = Account.availableBalance

Set newBalance = balance + “Amount”
Set newAvailableBalance = availableBalance + “Amount”

Open Database Connection

Set Account.balance = newBalance variable
Set Account.availableBalance = newAvailableBalance variable

‘Now insert transaction record

Call InsertTransaction() method from Transaction Class

Close Database Connection

Display “Amount Deposited” Information Message

End Call to OK_Click() method

6.2.10 “Make Cheque Deposit” Description

User Clicks “Deposits” in Form frmAccountDetails

Then, user clicks “Cheque” in menu “Deposits” in Form frmAccountDetails

Open Form frmChequeDeposit

User Enters “Amount” to be deposited

User Clicks OK

Call OK_Click() method in frmChequeDeposit

Declare balance and newBalance variables

Open Database Connection

Clear DataSet (virtual database)

SELECT balance
FROM Account
WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

Close Database Connection

Set balance variable = Account.balance

Set newBalance = balance + “Amount”

Open Database Connection

Set Account.balance = newBalance variable

Close Database Connection

‘Now insert transaction record

Call InsertTransaction() method from Transaction Class

Display “Amount Deposited” Information Message

End Call to OK_Click() method

6.2.11 “Post Interest” Description

User Clicks “Deposits” in Form frmAccountDetails

User Clicks “Post interest” in menu “Deposits” in Form frmAccountDetails

Display “Are you sure you want to Post Interest?” Question Message

If Answer is Yes Then

Declare totalInterest, balance, newBalance, availableBalance and newAvailableBalance variables

Open Database Connection

Clear DataSet (virtual database)

SELECT balance, availableBalance, totalInterest
FROM Account
WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

Close Database Connection

Set totalInterest variable = Account.totalInterest
Set balance variable = Account.balance
Set availableBalance variable = Account.availableBalance

Set newBalance variable = balance + totalInterest
Set newAvailableBalance variable = availableBalance + totalInterest

Open Database Connection

Set Account.totalInterest = “0”
Set Account.balance = newBalance variable
Set Account.availableBalance = newAvailableBalance variable

Close Database Connection

‘Now insert transaction record

Call InsertTransaction() method from Transaction Class

Display “Interest Posted” Information Message

End If

6.2.12 "Deduct Charges" Description

User Clicks "Withdrawals" in Form frmAccountDetails

User Clicks "Deduct charges" in menu "Withdrawals" in Form frmAccountDetails

Display "Are you sure you want to Deduct Charges?" Question Message

If Answer is Yes Then

Declare charges, balance, newBalance, availableBalance and newAvailableBalance variables

Open Database Connection

Clear DataSet (virtual database)

SELECT balance, availableBalance, charges
FROM Account
WHERE sortCode = "Sort Code" AND accountNo = "Account Number"

Close Database Connection

Set charges variable = Account.charges
Set balance variable = Account.balance
Set availableBalance variable = Account.availableBalance

Set newBalance variable = balance – charges
Set newAvailableBalance variable = availableBalance – charges

If newAvailableBalance < 0 Then

Set newAvailableBalance = 0

End If

Open Database Connection

Set Account.Charges = "0"
Set Account.balance = newBalance variable
Set Account.availableBalance = newAvailableBalance variable

Close Database Connection

'Now insert transaction record

Call InsertTransaction() method from Transaction Class

Connect to Bank of England's Server via SWIFTNet[18]

Credit *BestBank's* Bank of England account by amount of charges

Display "Charges Deducted" Information Message

[18] See Appendix A – Glossary for definition of SWIFTNet

End If

6.2.13 “Make Inter-Account Transfer” Description

User Clicks “Withdrawals” in Form frmAccountDetails

User Clicks “Inter-account transfer” in menu “Withdrawals” in Form frmAccountDetails

Open Form frmInterAccountTransfer

User Enters Collecting Account’s sort code

User Enters Collecting Account’s account number

User Enters “Amount” to be transferred to the collecting account

User Clicks “Transfer”

Call Transfer_Click() method

Declare balance, newBalance, availableBalance and newAvailableBalance variables

Open Database Connection

Clear DataSet (virtual database)

SELECT balance, availableBalance
FROM Account
WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

Close Database Connection

Set balance variable = Account.balance
Set availableBalance variable = Account.availableBalance

If availableBalance >= “Amount” (to be transferred) Then

Set newBalance variable = balance – “Amount”
Set newAvailableBalance variable = availableBalance – “Amount”

Open Database Connection

Set Account.balance = newBalance variable
Set Account.avaialbleBalance = newAvailableBalance variable

Close Database Connection

‘Now insert transaction record

Call InsertTransaction() method from Transaction Class

Open Database Connection

Clear DataSet (virtual Database)

SELECT balance, availableBalance

FROM Account
WHERE sortCode = “Collecting Account’s sort code” AND accountNo = “Collecting Account’s account number”

Close Database Connection

Set balance variable = Account.balance
Set availableBalance variable = Account.availableBalance

Set newBalance variable = balance + “Amount”
Set newAvailableBalance variable = availableBalance + “Amount”

Open Database Connection

Set Collecting Account.balance = newBalance variable
Set Collecting Account.availableBalance = newAvailableBalance variable

Close Database Connection

‘Now insert transaction record

Call InsertTransaction() method from Transaction Class

Display “Transfer Complete” Information Message

Else

Display “Insufficient Funds Available” Warning Message

End If

End Call to Transfer_Click method

6.2.14 “Issue Banker’s Draft” Description

User Clicks “Withdrawals” in Form frmAccountDetails

User Clicks “Banker’s draft” in menu “Withdrawals” in Form frmAccountDetails

Open Form frmBankrsDraft

User Enters “Amount” to be withdrawn

User Clicks OK

Call OK_Click method

Declare balance, newBalance, availableBalance and newAvailableBalance variables

Open Database Connection

Clear DataSet (virtual database)

SELECT balance, availableBalance
FROM Account
WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

Close Database Connection

Set balance variable = Account.balance
Set availableBalance variable = Account.availableBalance

If availableBalance >= “Amount” Then

Set newBalance = balance – “Amount”
Set newAvailableBalance = availableBalance – “Amount”

Open Database Connection

Set Account.balance = newBalance variable
Set Account.availableBalance = newAvailableBalance variable

Close Database Connection

‘Now insert transaction record

Call InsertTransaction() method from Transaction Class

Display “Amount Withdrawn” Information Message

Else

Display “Insufficient Funds Available” Warning Message

End If

End Call to OK_Click() method

6.2.15 “Make Cash Withdrawal” Description

User Clicks “Withdrawals” in Form frmAccountDetails

User Clicks “Cash” in menu “Withdrawals” in Form frmAccountDetails

Open Form frmCashWithdrawal

User Enters “Amount” to be withdrawn

User Clicks OK

Call OK_Click method

Declare balance, newBalance, availableBalance and newAvailableBalance variables

Open Database Connection

Clear DataSet (virtual database)

SELECT balance, availableBalance
FROM Account
WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

Close Database Connection

Set balance variable = Account.balance
Set availableBalance variable = Account.availableBalance

If availableBalance >= “Amount” Then
Set newBalance = balance – “Amount”

Set newAvailableBalance = availableBalance – “Amount”

Open Database Connection

Set Account.balance = newBalance variable
Set Account.availableBalance = newAvailableBalance variable

Close Database Connection

‘Now insert transaction record

Call InsertTransaction() method from Transaction Class

Display “Amount Withdrawn” Information Message

Else

Display “Insufficient Funds Available” Warning Message

End If

End Call to OK_Click() method

6.2.16 “View Statement” Description

User Clicks “Statements” in Form frmAccountDetails

Open Form frmStatements

Call frmStatements_Load() method

Set TextBox txtStartDate Text = “”
Set TextBox txtFinishDate Text = “”

End Call to frmStatements_Load() method

User Enters “Start Date”

User Enters “Finish Date”

User Clicks “View Statement” Button

Call cmdViewStatement_Click() method

If “Start Date” = “” AND “Finish Date” = “”

Open Database Connection

```
        Clear DataSet (virtual database)

        SELECT transactionNo, date, description, credit, debit, balance
        FROM Transactions
        WHERE sortCode = “Sort Code” AND accountNo = “Account Number”

        Close Database Connection

        Display Results of query in datagrid dgdResults

End If

If “Start Date” <> “” AND “Finish Date” <> “” Then

        Open Database Connection

        Clear DataSet (virtual database)

        SELECT transactionNo, date, description, credit, debit, balance
        FROM Transactions
        WHERE sortCode = “Sort Code” AND accountNo = “Account Number” AND
                date >= “StartDate” AND date <= “Finish Date”

        Close Database Connection

        Display Results of query in datagrid dgdResults

End If

If “Start Date” <> “” AND “Finish Date” = “” Then

        Open Database Connection

        Clear DataSet (virtual database)

        SELECT transactionNo, date, description, credit, debit, balance
        FROM Transactions
        WHERE sortCode = “Sort Code” AND accountNo = “Account Number” AND
          date >= “StartDate”

        Close Database Connection

        Display Results of query in datagrid dgdResults

End If

If “Start Date” = “” AND “Finish Date” <> “” Then

        Open Database Connection

        Clear DataSet (virtual database)

        SELECT transactionNo, date, description, credit, debit, balance
        FROM Transactions
        WHERE sortCode = “Sort Code” AND accountNo = “Account Number” AND
                date <= “Finish Date”
```

Close Database Connection

Display Results of query in datagrid dgdResults

End If

End Call to cmdViewStatement_Click()

6.3 Class Diagram

Figure 6.2 is the Class Diagram for the FAMS application. Note that the class “CloseButton” facilitates disabling of the close button on form frmMainMenu, forcing the user to log out of the FAMS application rather than simply closing the Main Menu window. Classes”CloseButton” and “Transaction” have shared (static) methods and so are not instantiated. Hence, they have unspecified multiplicities in the class diagram.

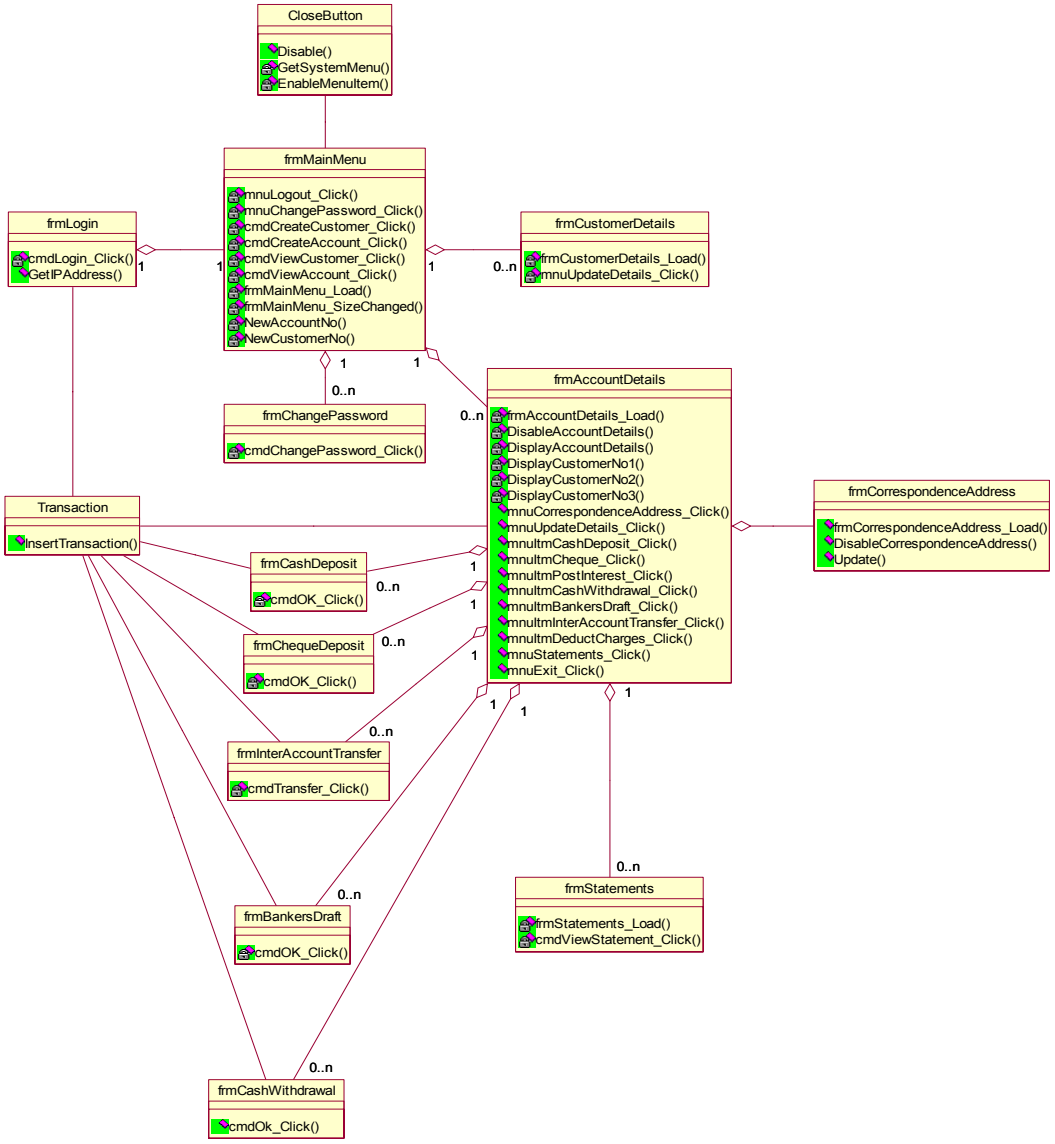


Figure 6.2: Class Diagram for FAMS Application

6.4 Mapping Classes to Relational Tables

The following table contains mappings of the classes in the FAMS application to the relational tables in the FAMSDB Database which contain the data that the methods in the classes manipulate.

Classes	Relational Tables That Class Methods Act On
frmLogin	Staff
frmMainMenu	Account, AccountMaintenance, Customer, CustomerMaintenance, Staff
frmChangePassword	Staff
frmCorrespondenceAddress	Account
frmCustomerDetails	Account, AccountMaintenance, Customer
frmAccountDetails	Account, AccountMaintenance, Customer, Transaction
Transaction	Staff, Transaction
frmCashDeposit	Account, Transaction
frmChequeDeposit	Account, Transaction
frmInterAccountTransfer	Account, Transaction
frmBankersDraft	Account, Transaction
frmCashWithdrawal	Account, Transaction
frmStatements	Transaction

6.5 Sequence Diagrams

6.5.1 "Login Staff" Sequence Diagram

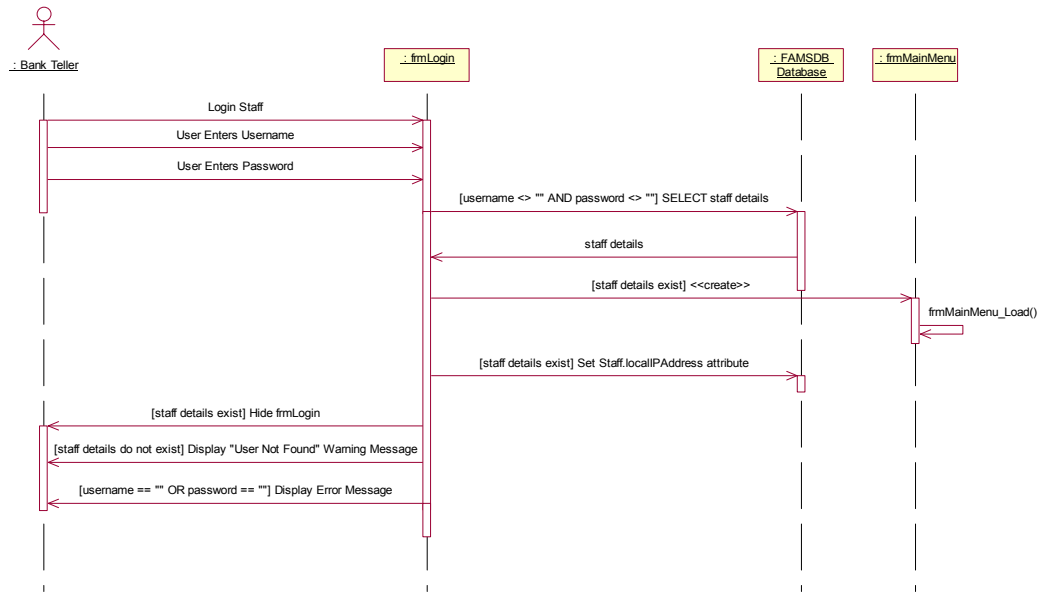


Figure 6.3: "Login Staff" Sequence Diagram

6.5.2 “Logout Staff” Sequence Diagram

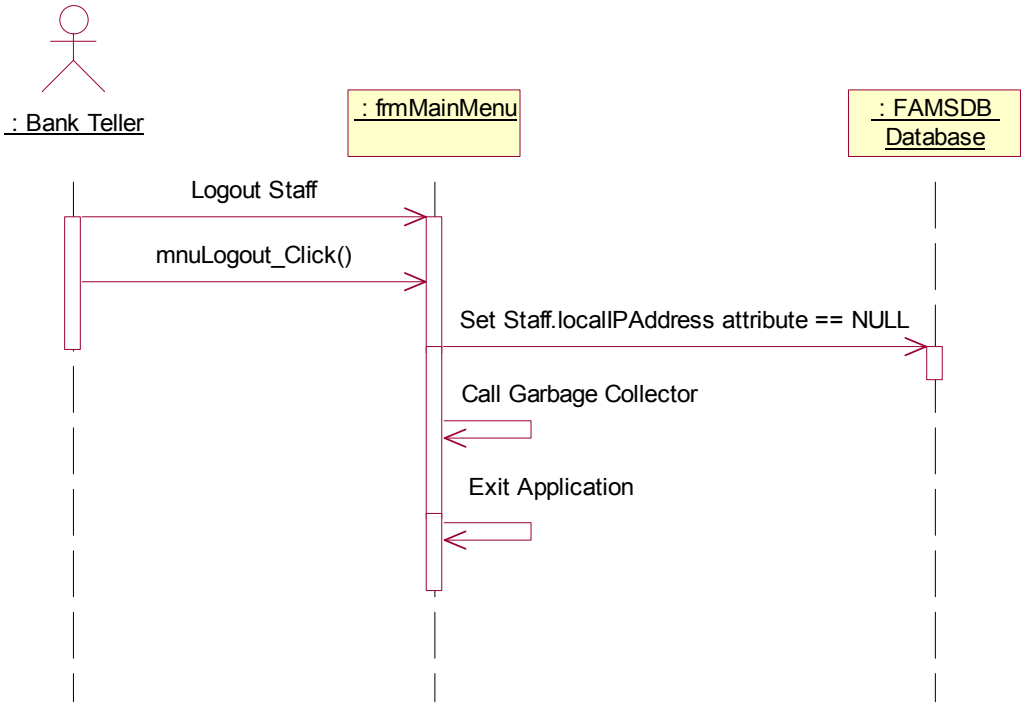


Figure 6.4 “Logout Staff” Sequence Diagram

6.5.3 “Change Password” Sequence Diagram

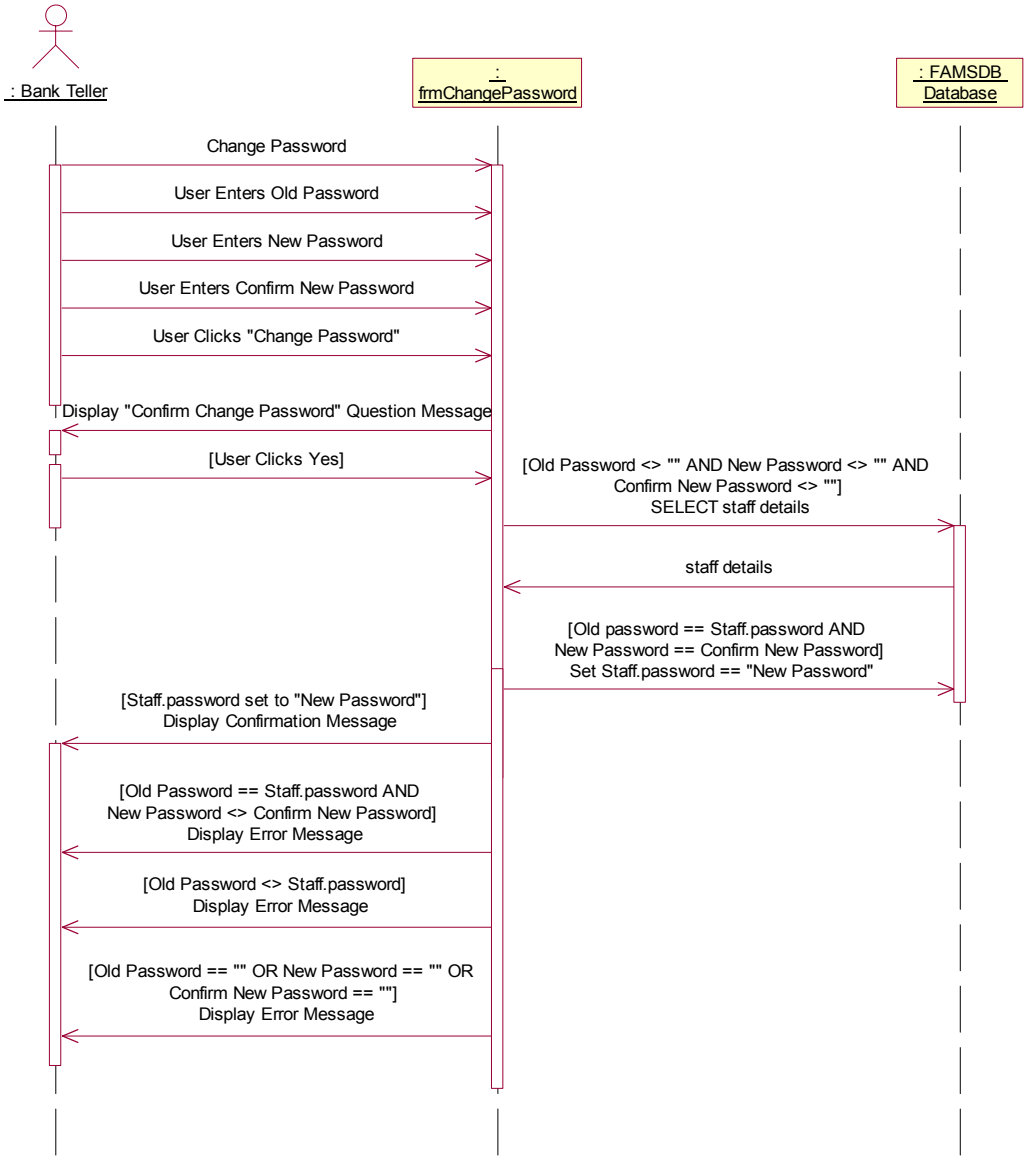


Figure 6.5 “Change Password” Sequence Diagram

6.5.4 “Open Account” Sequence Diagram

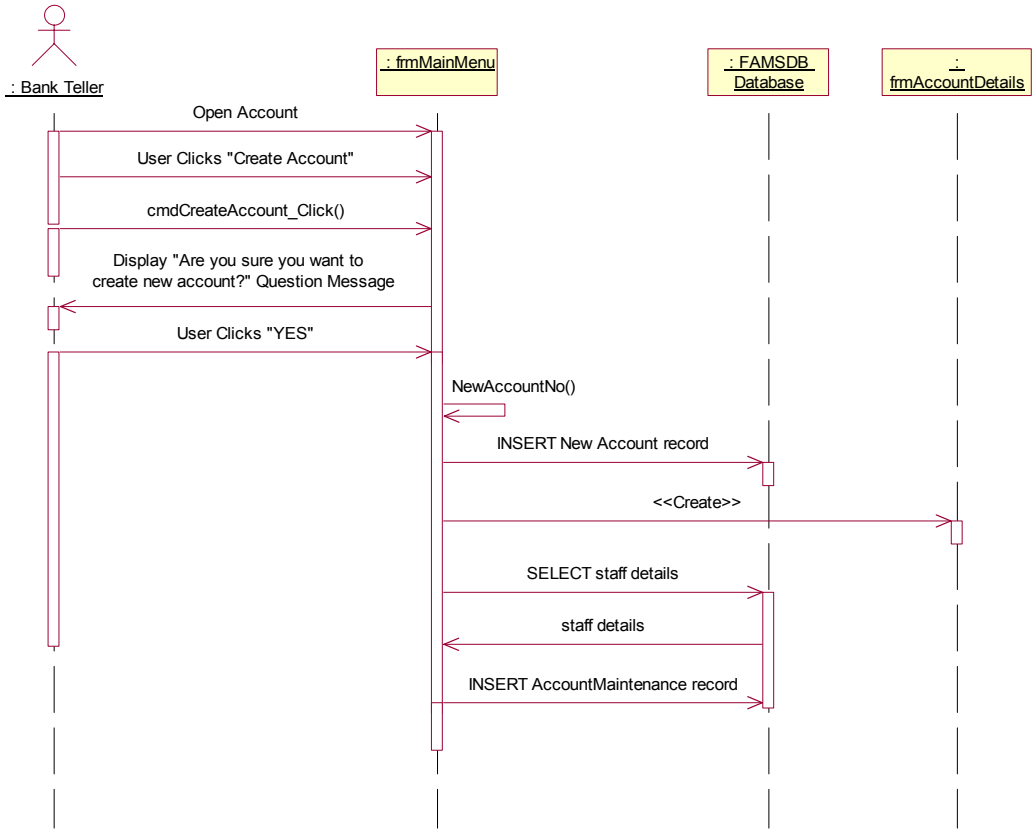


Figure 6.6 “Open Account” Sequence Diagram

6.5.5 “Create Customer” Sequence Diagram

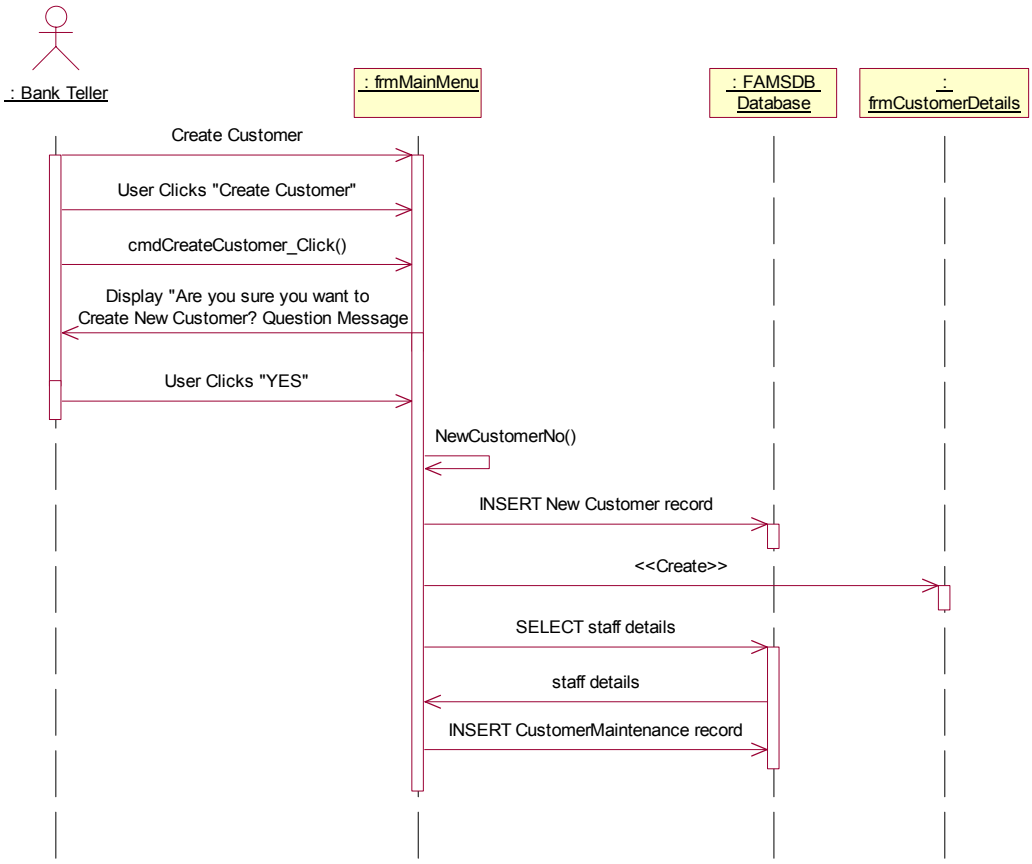


Figure 6.7 “Create Customer” Sequence Diagram

6.5.6 “View/Update Customer Details” Sequence Diagram

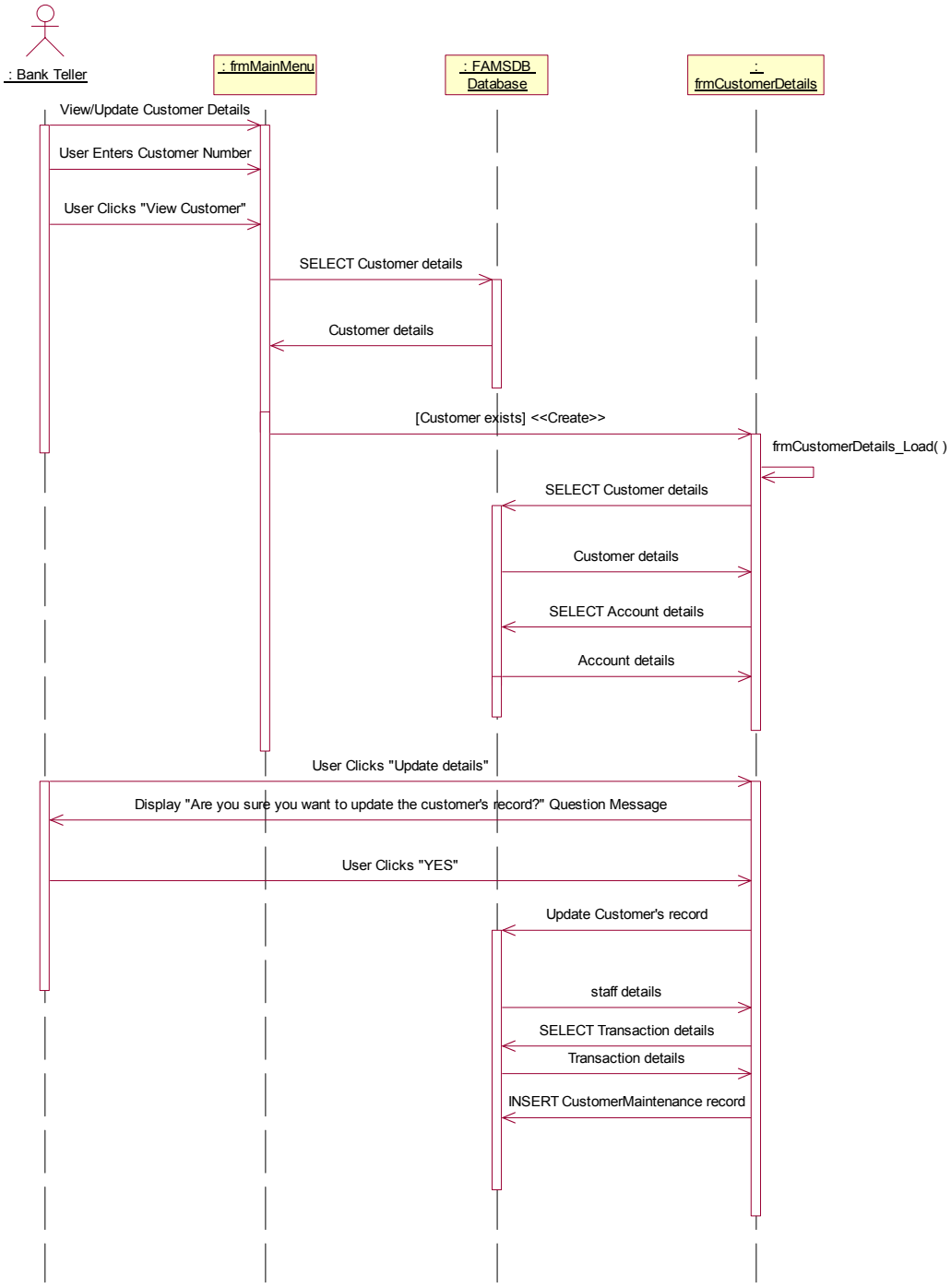


Figure 6.8 “View/Update Customer Details” Sequence Diagram

6.5.7 “View/Update Account Details” Sequence Diagram

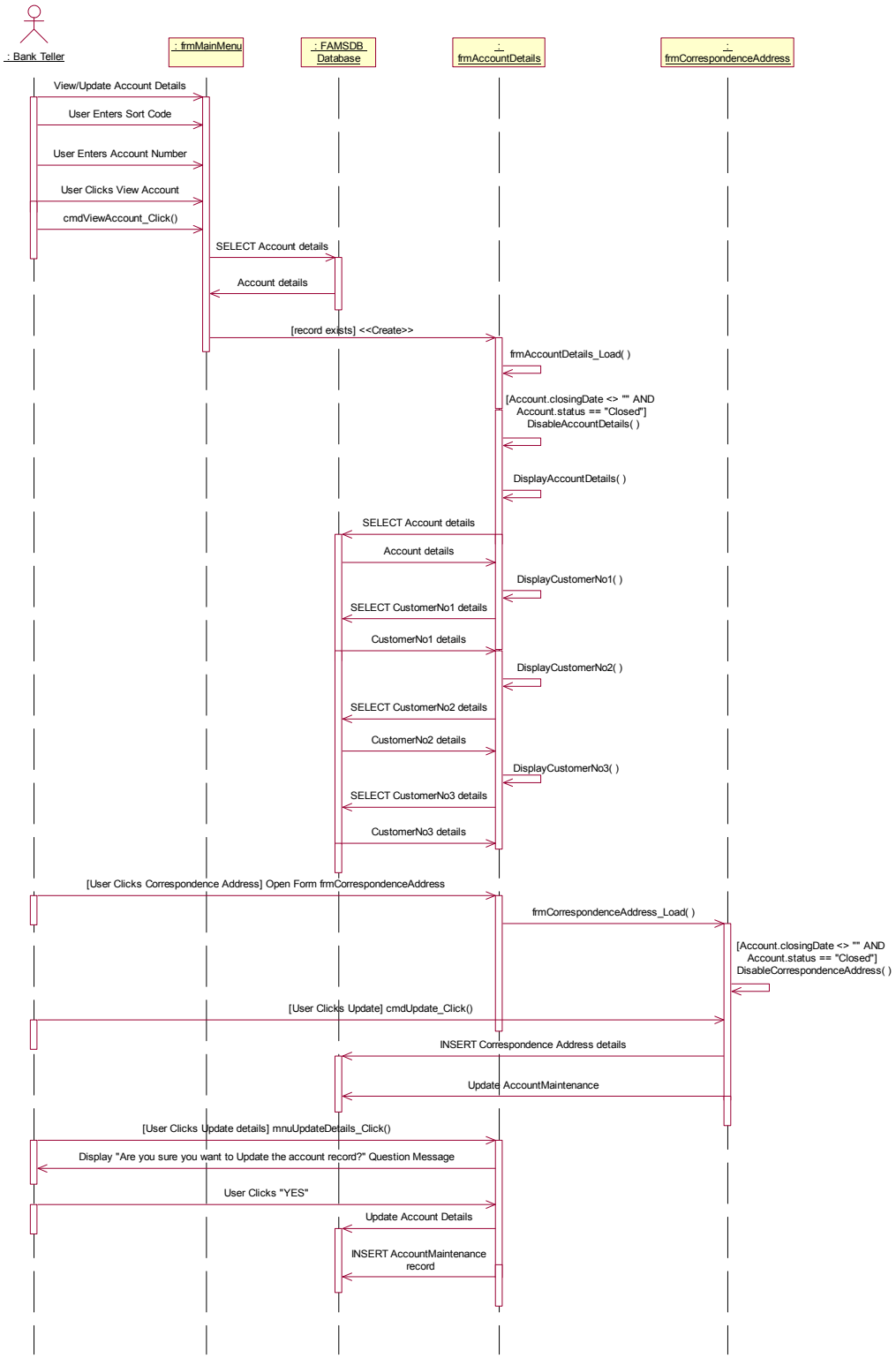


Figure 6.9 “View/Update Account Details” Sequence Diagram

6.5.8 “Make Cash Deposit” Sequence Diagram

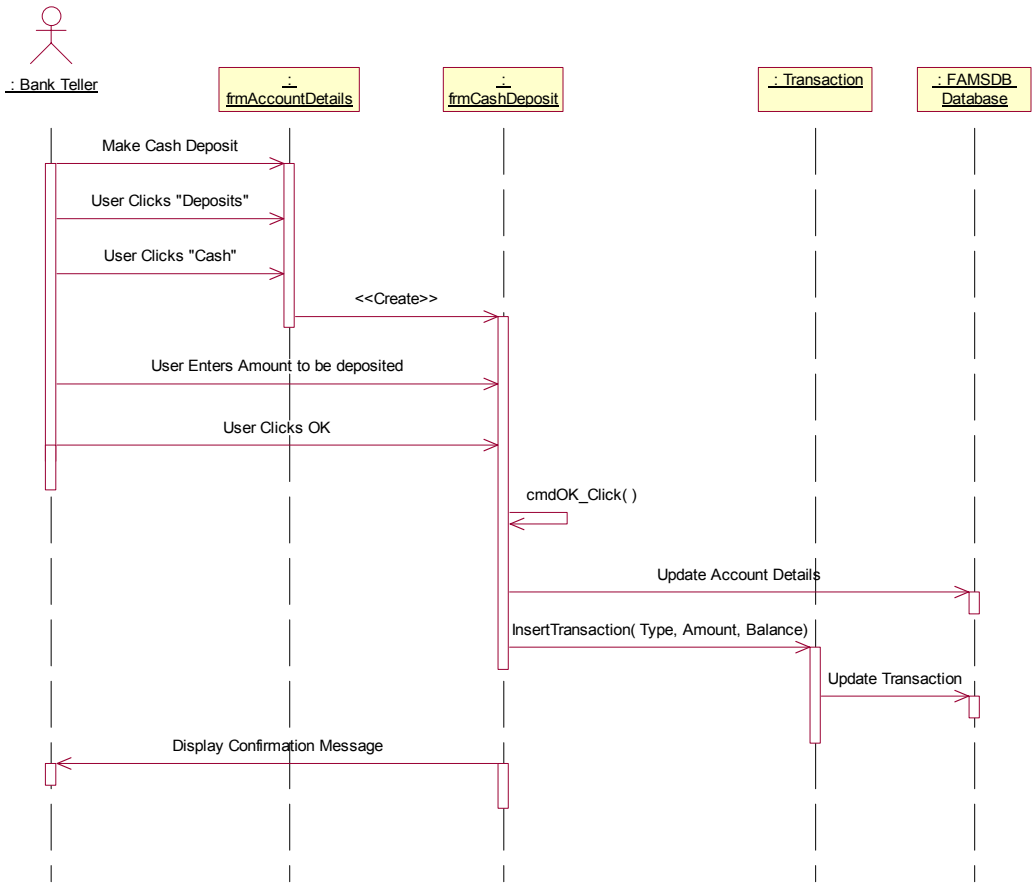


Figure 6.10 “Make Cash Deposit” Sequence Diagram

6.5.9 “Make Cheque Deposit” Sequence Diagram

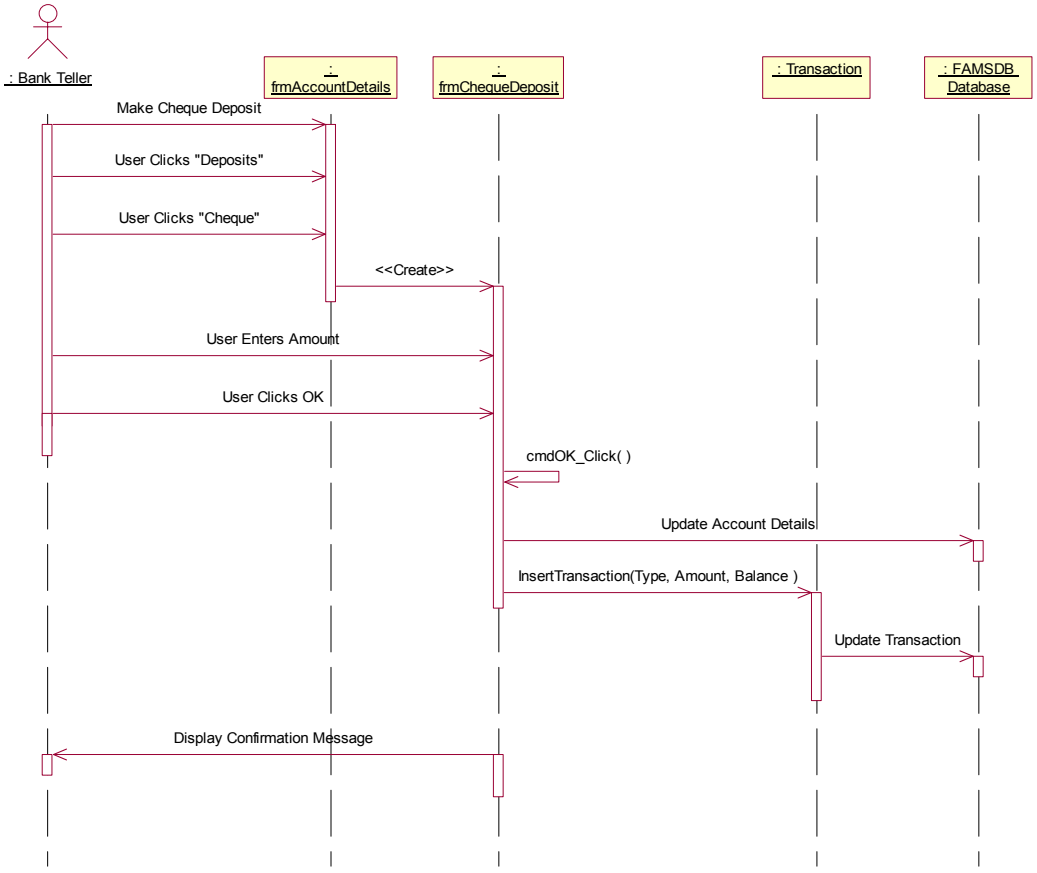


Figure 6.11 “Make Cheque Deposit” Sequence Diagram

6.5.10 “Post Interest” Sequence Diagram

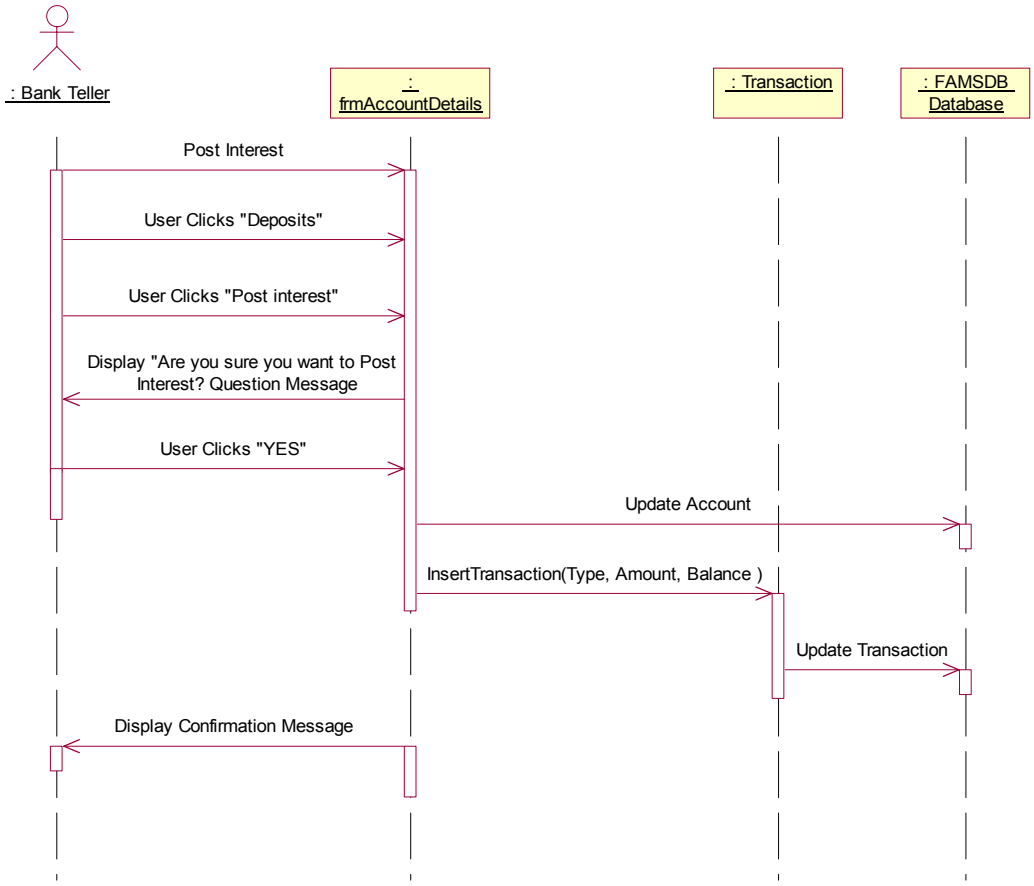


Figure 6.12 “Post Interest” Sequence Diagram

6.5.11 “Deduct Charges” Sequence Diagram

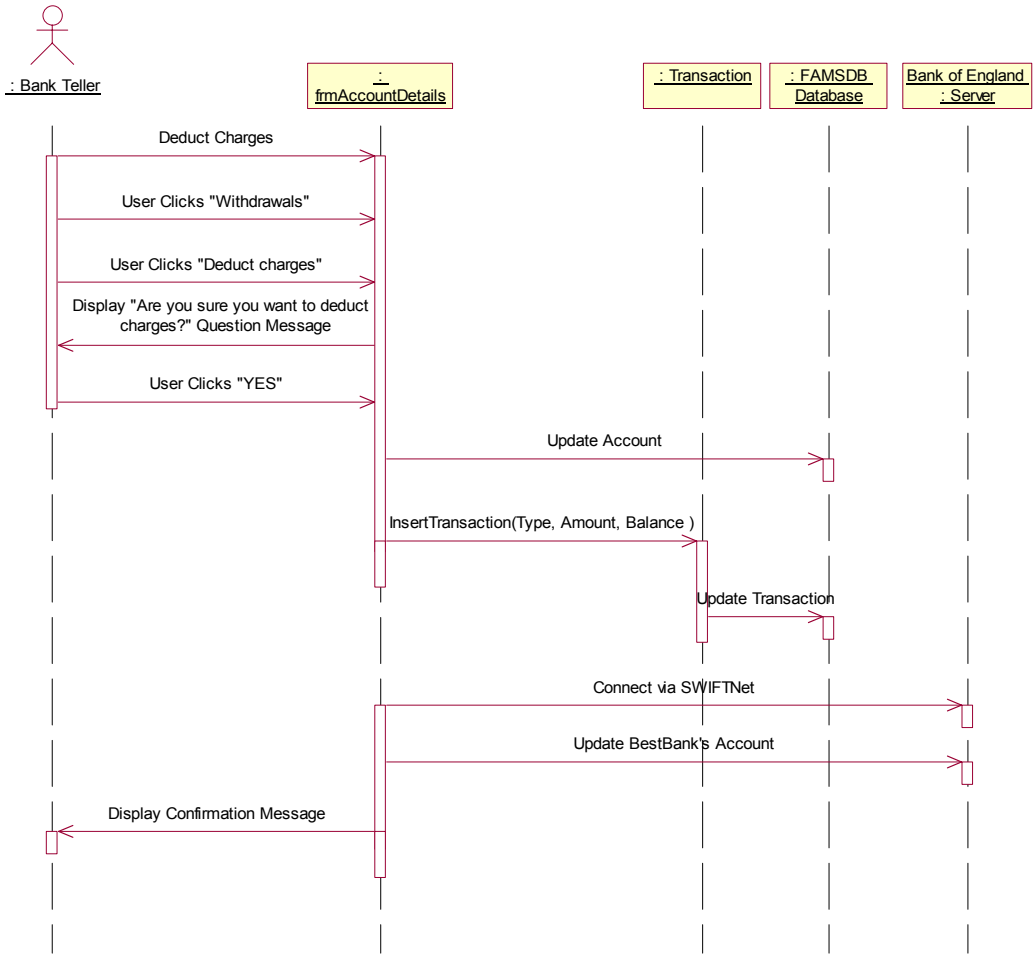


Figure 6.13 “Deduct Charges” Sequence Diagram

6.5.12 “Make Inter-Account Transfer” Sequence Diagram

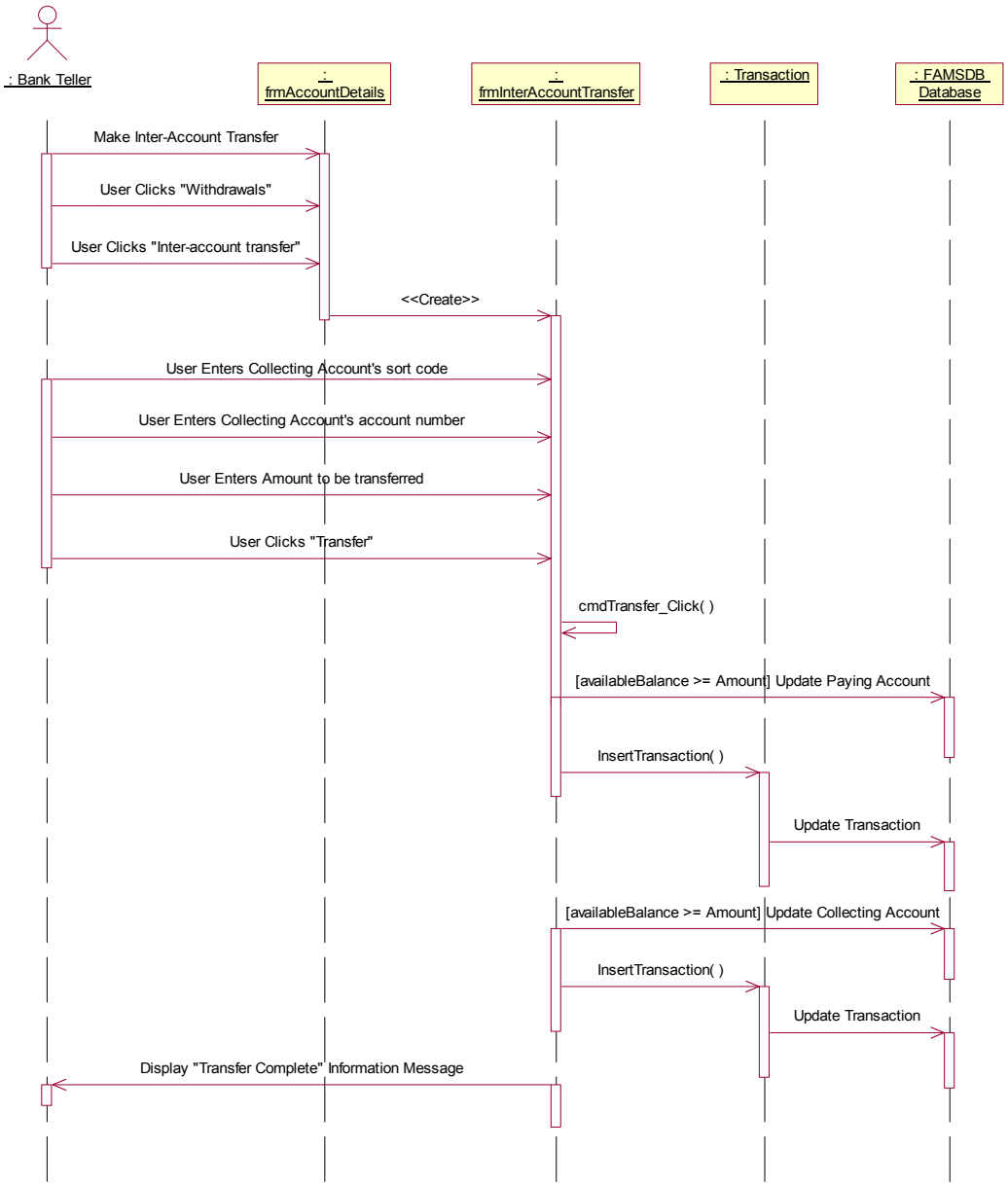


Figure 6.14 “Make Inter-Account Transfer” Sequence Diagram

6.5.13 “Issue Banker’s Draft” Sequence Diagram

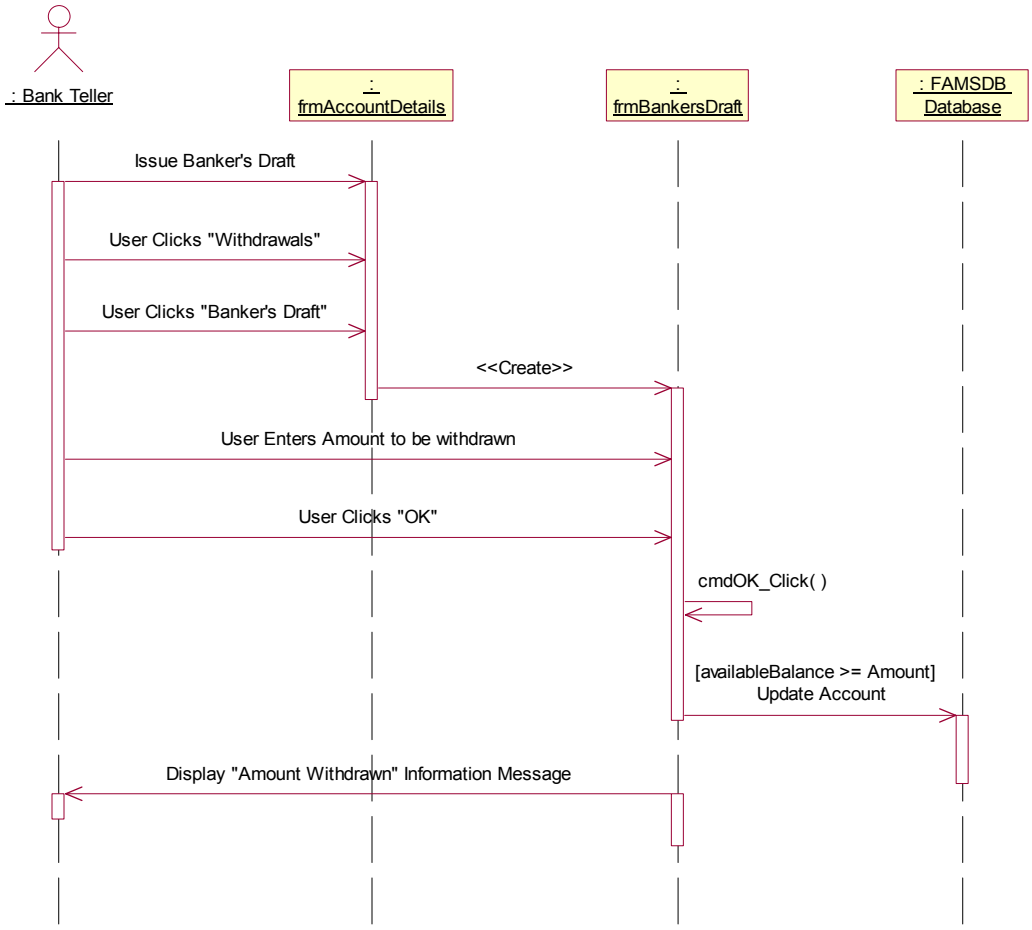


Figure 6.15 “Issue Banker’s Draft” Sequence Diagram

6.5.14 “Make Cash Withdrawal” Sequence Diagram

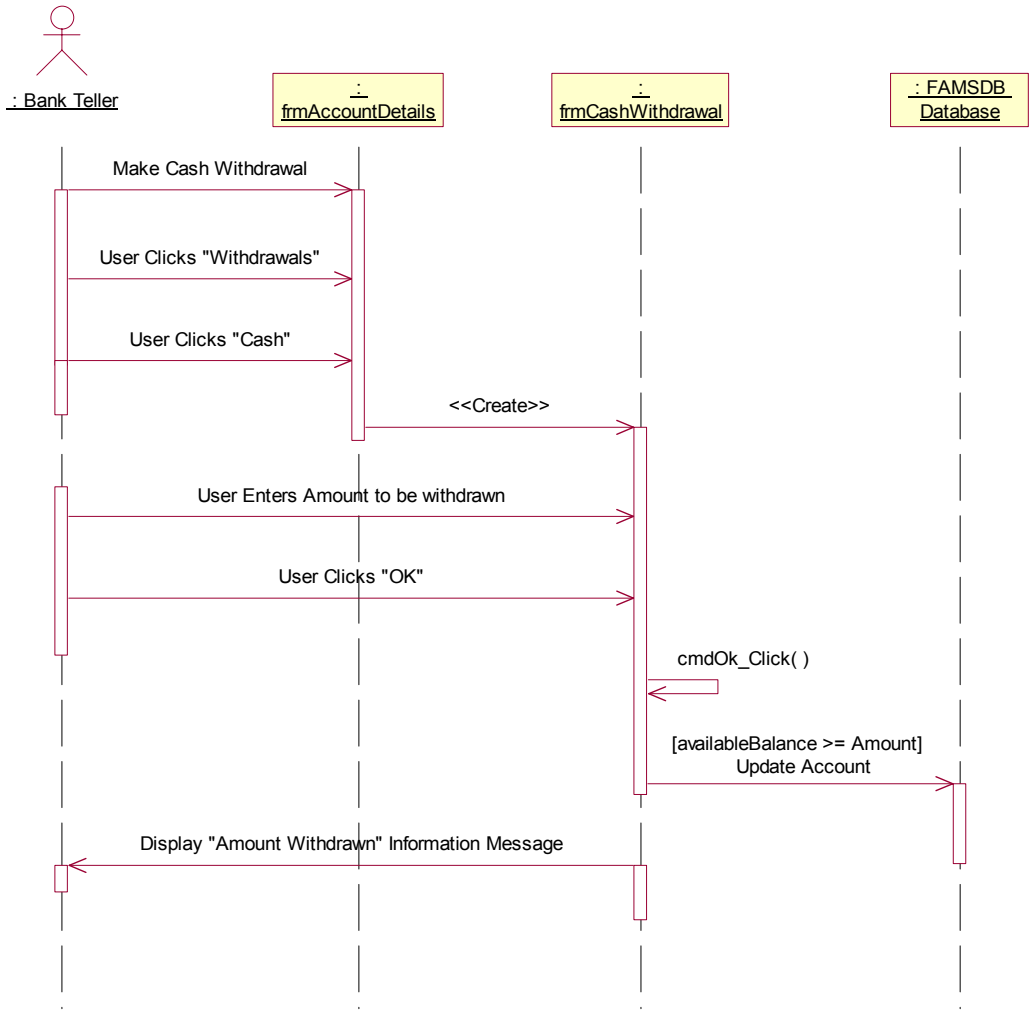


Figure 6.16 “Make Cash Withdrawal” Sequence Diagram

6.5.15 “View Statement” Sequence Diagram

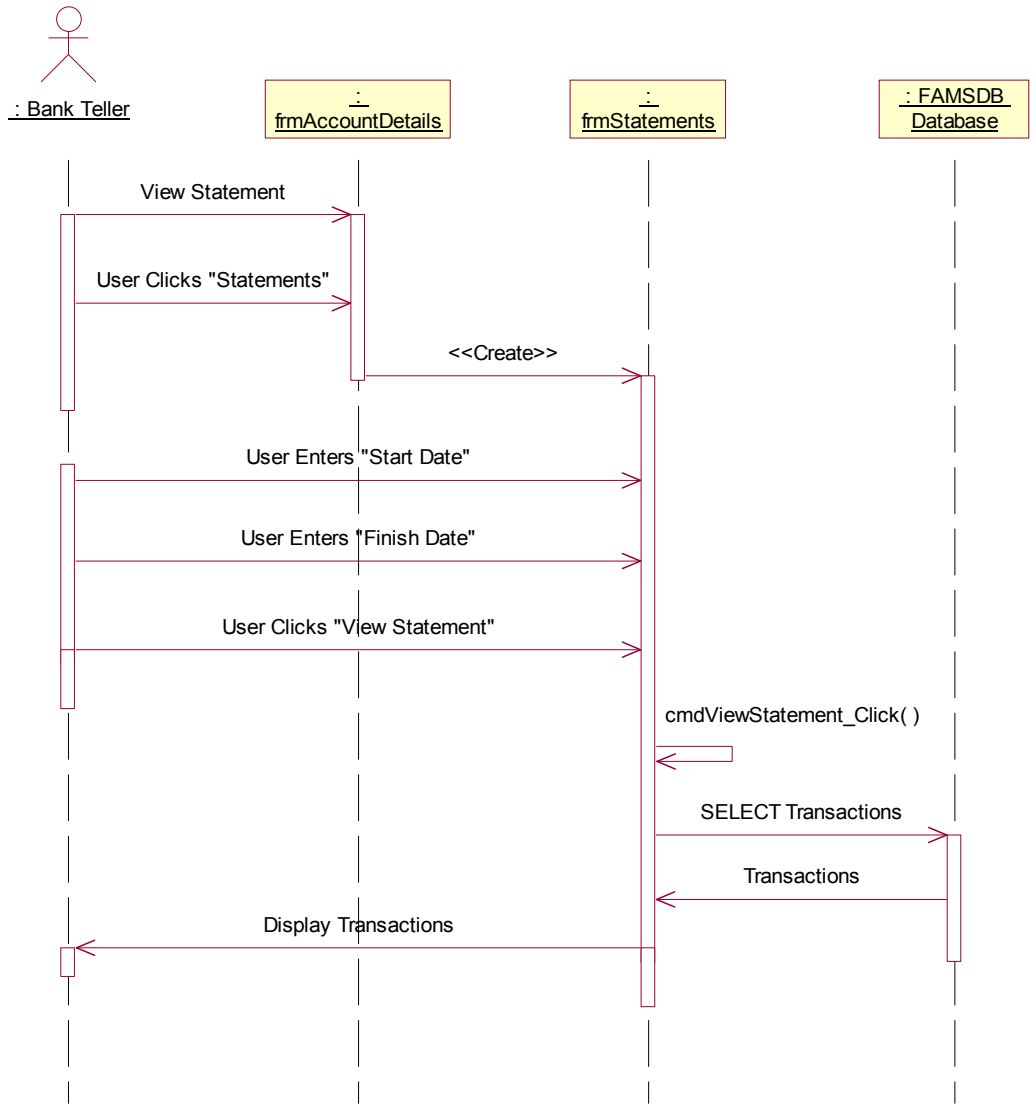
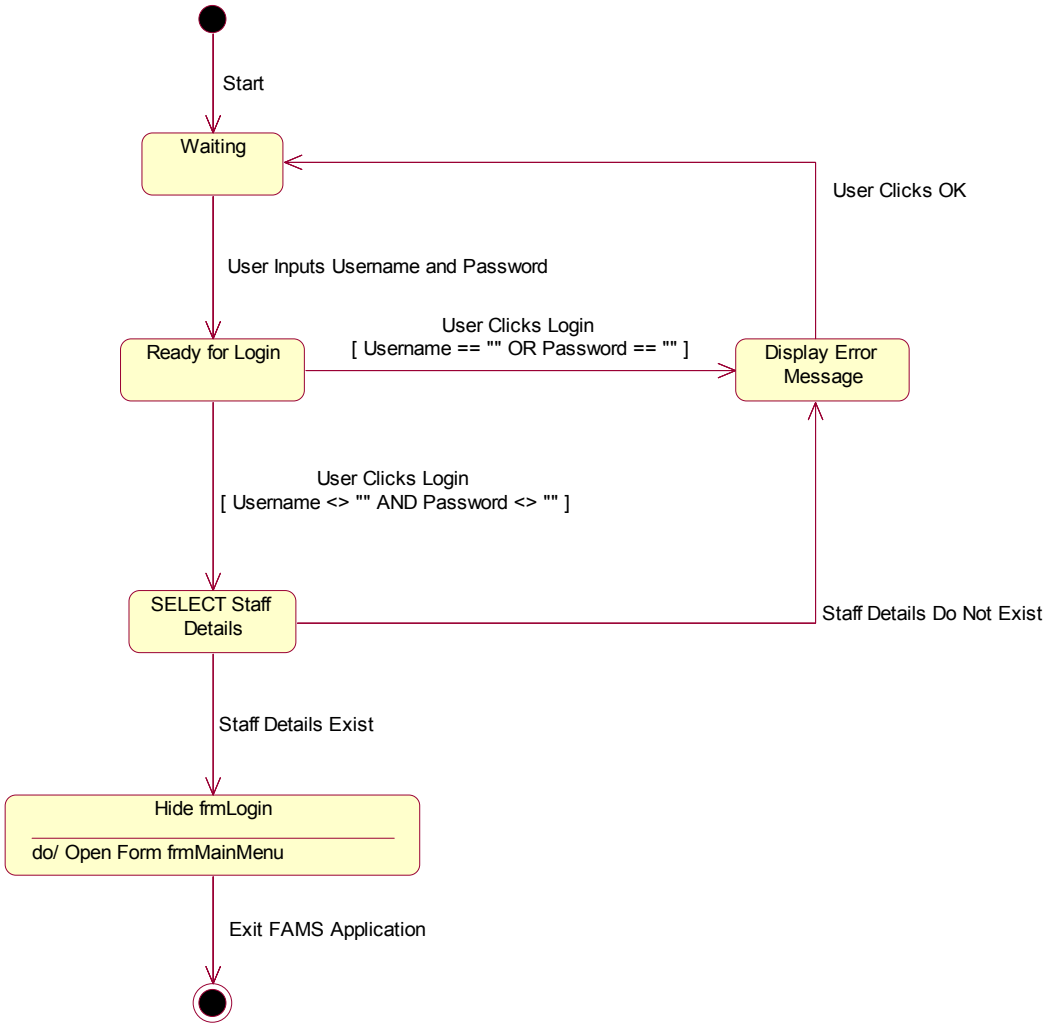


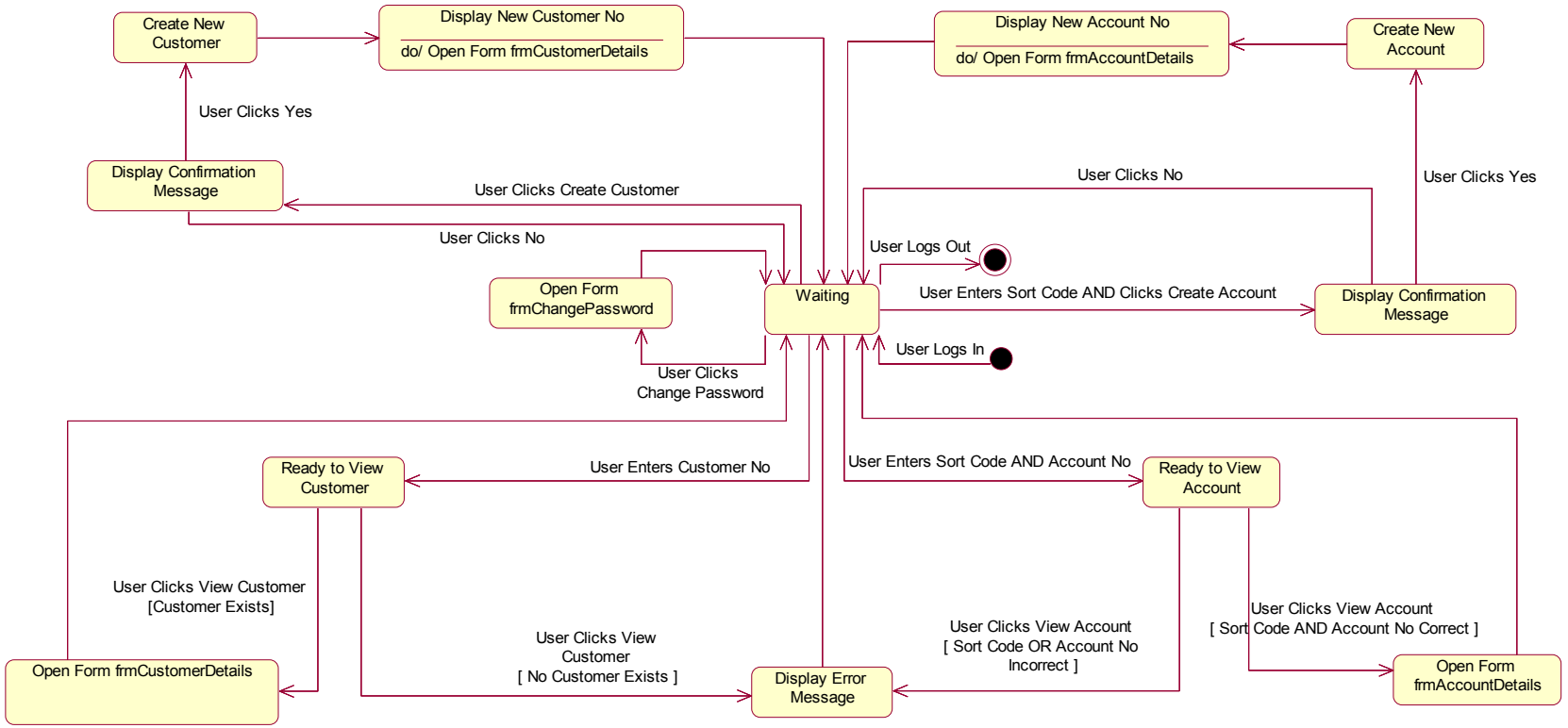
Figure 6.17 “View Statement” Sequence Diagram

6.6 Statechart Diagrams

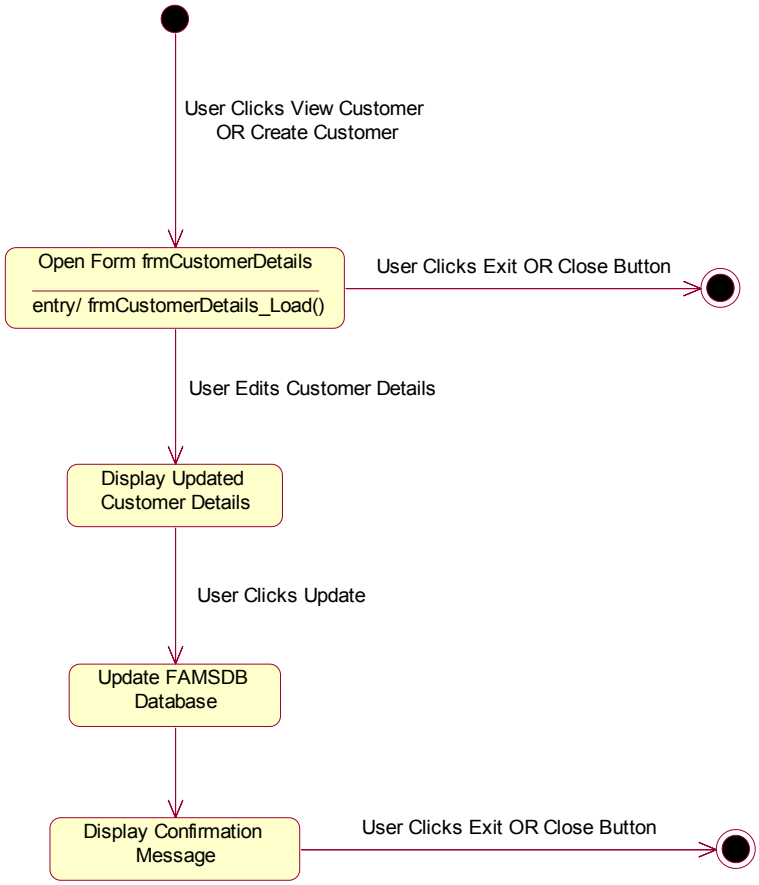
6.6.1 Statechart Diagram for frmLogin Object



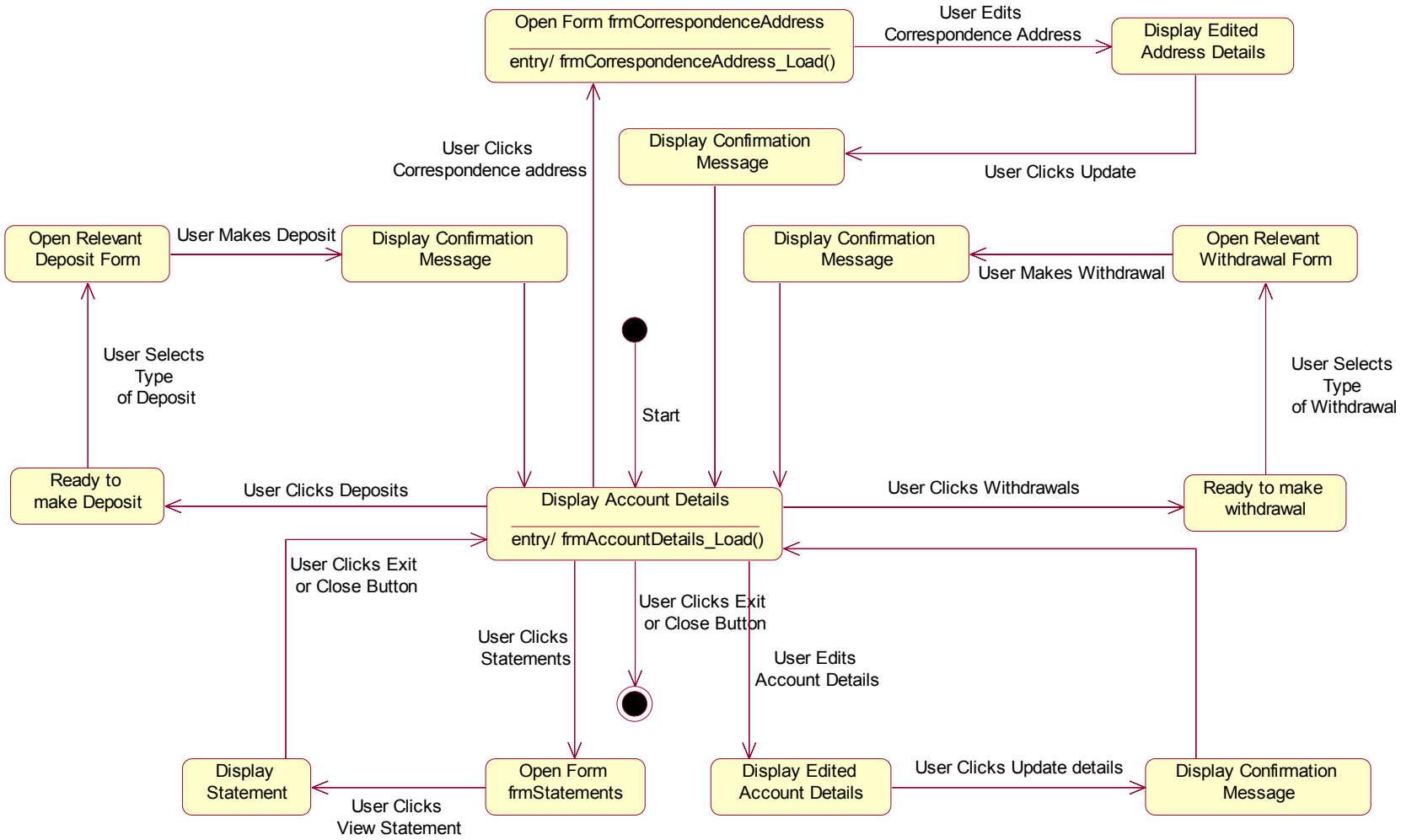
6.6.2 Statechart Diagram for frmMainMenu Object



6.6.3 Statechart Diagram for frmCustomerDetails Object



6.6.4 Statechart Diagram for frmAccountDetails Object



6.7 GUI Designs

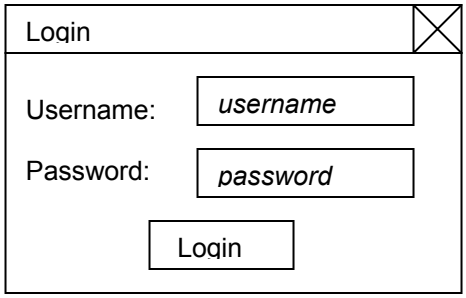


Figure 6.18: Login GUI

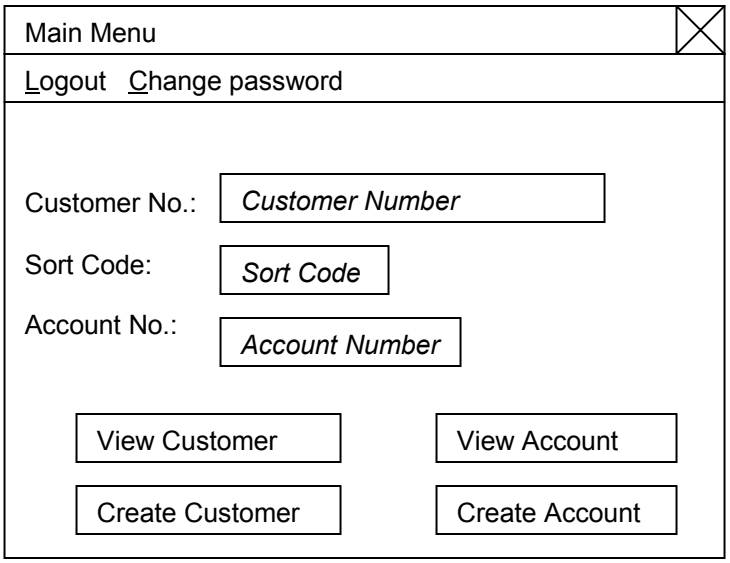


Figure 6.19: Main Menu GUI

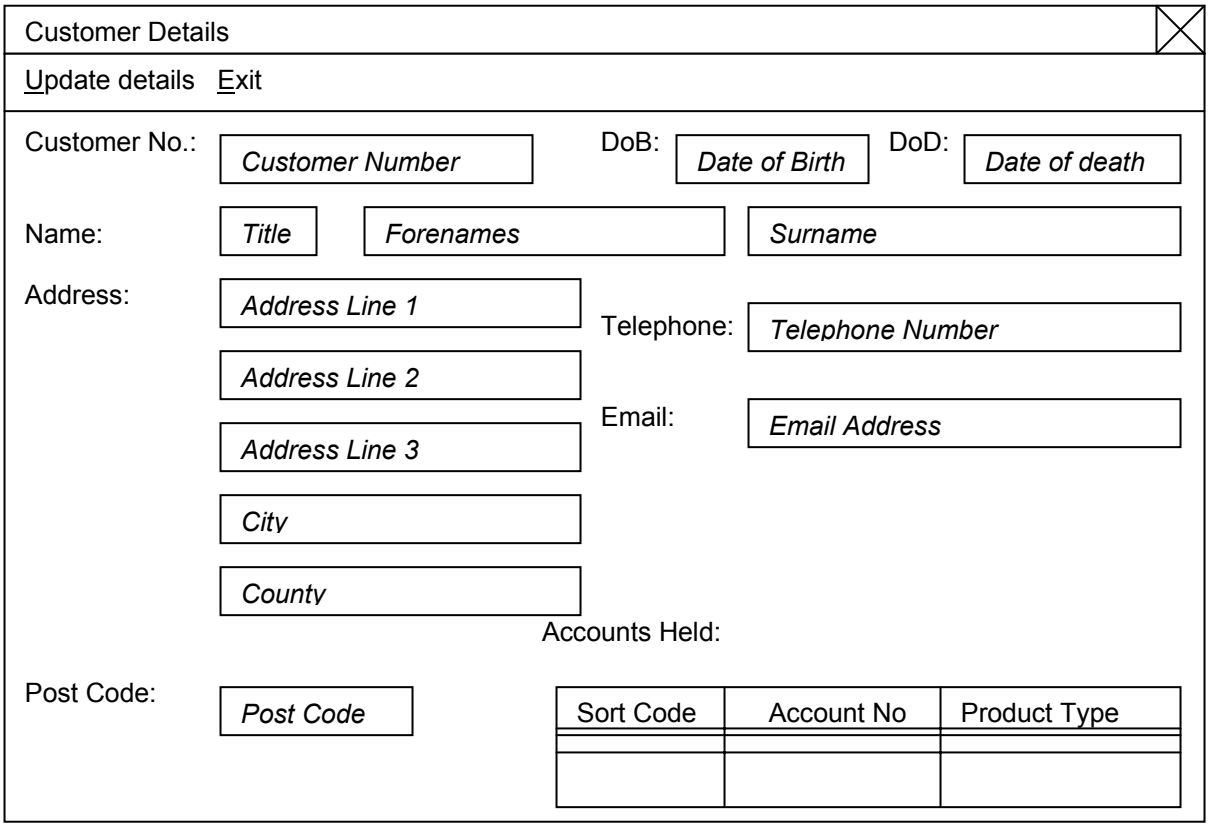


Figure 6.20: Customer Details GUI

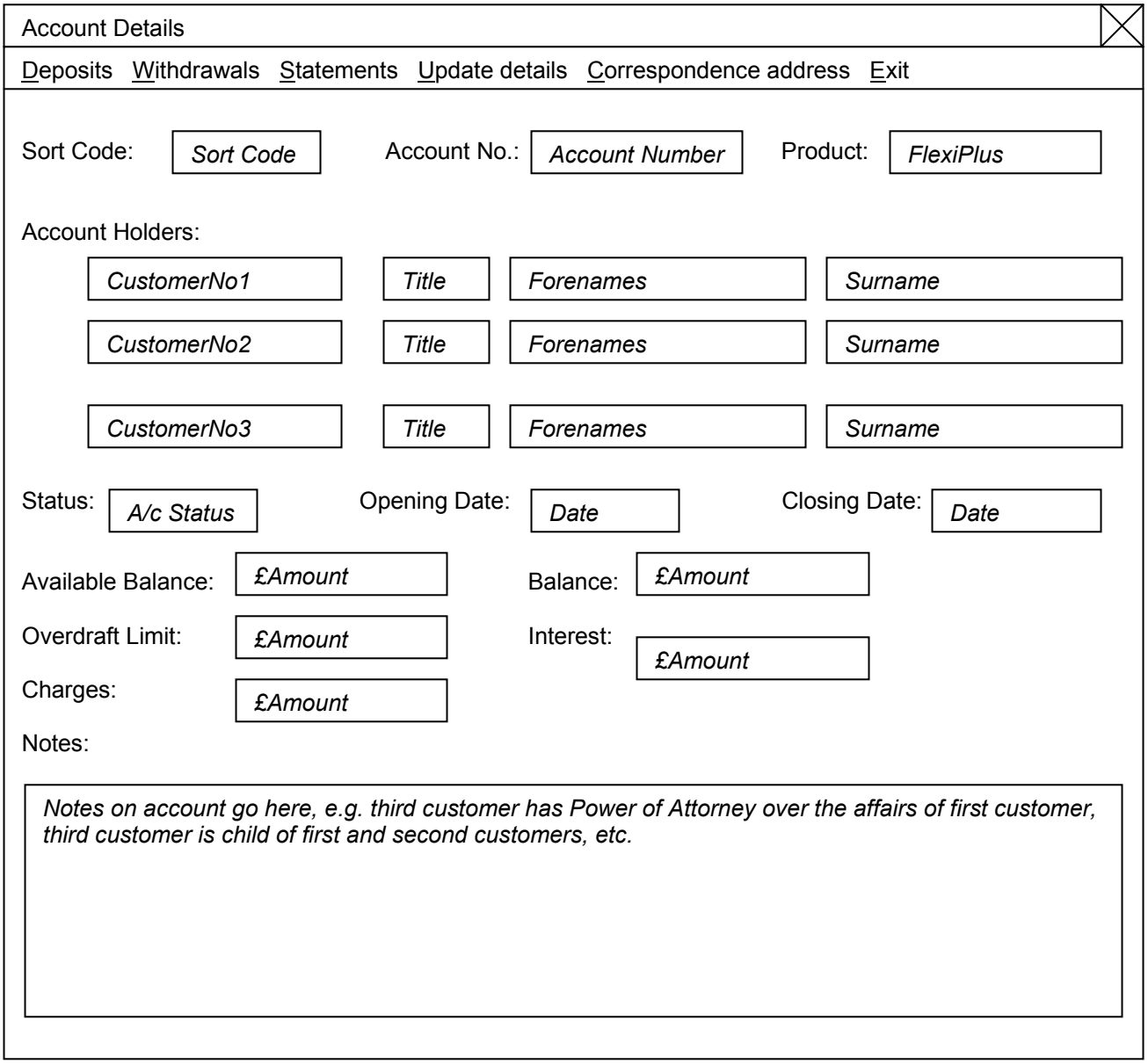


Figure 6.21: Account Details GUI

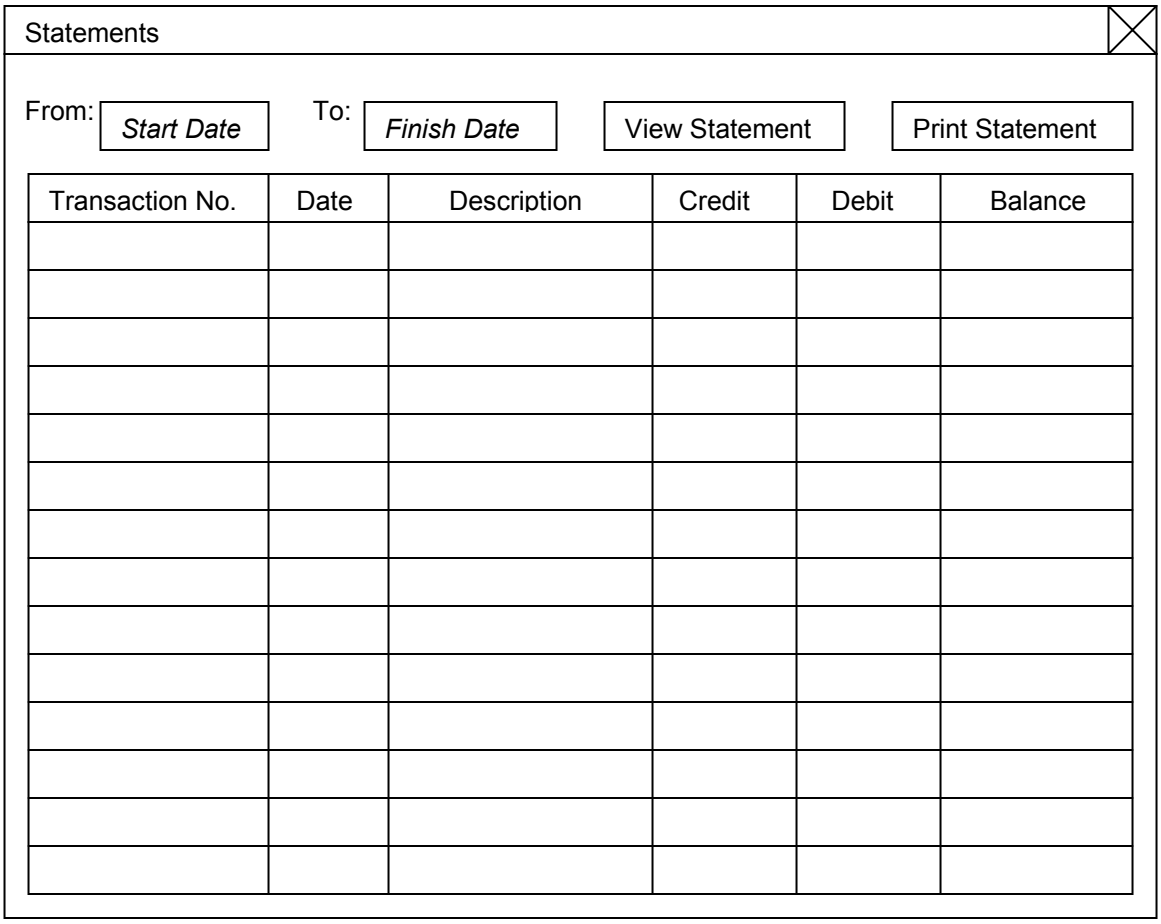


Figure 6.22: Statements GUI

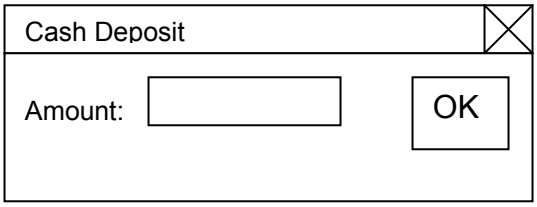


Figure 6.23: Cash Deposit GUI

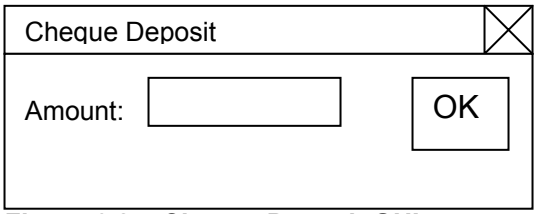


Figure 6.24: Cheque Deposit GUI

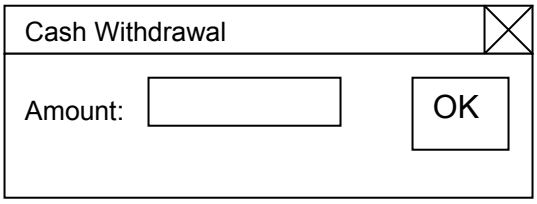


Figure 6.25: Cash Withdrawal GUI

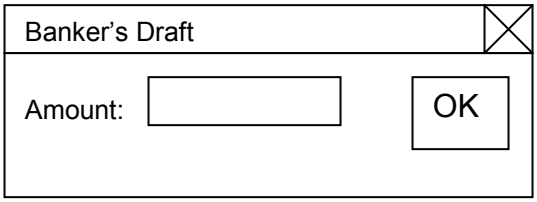


Figure 6.26: Banker's Draft GUI

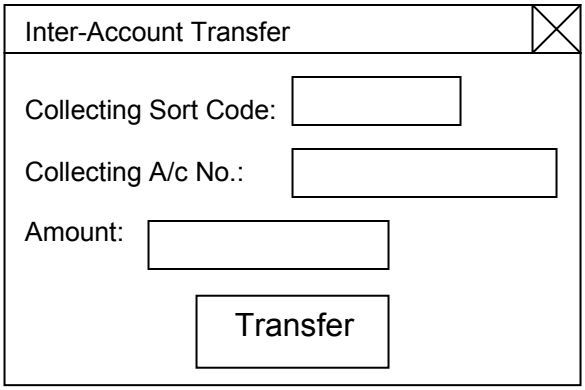


Figure 6.27: Inter-Account Transfer GUI

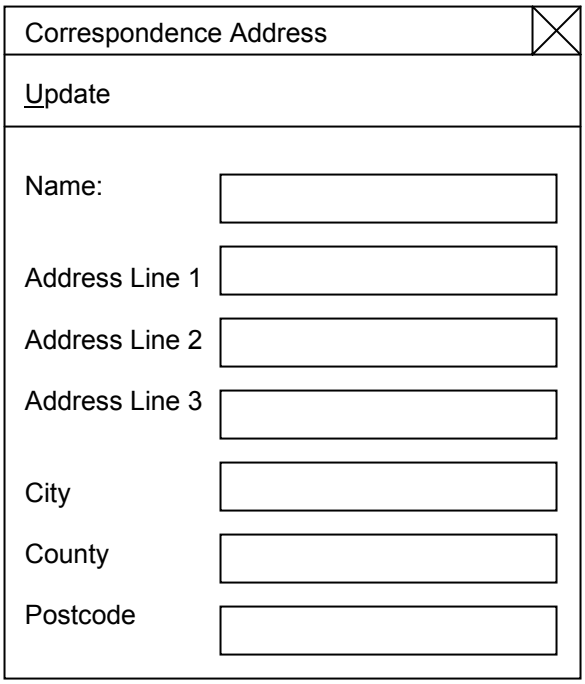


Figure 6.28: Correspondence Address GUI

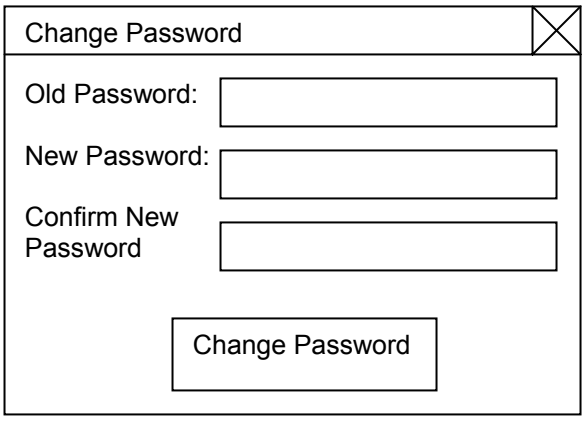
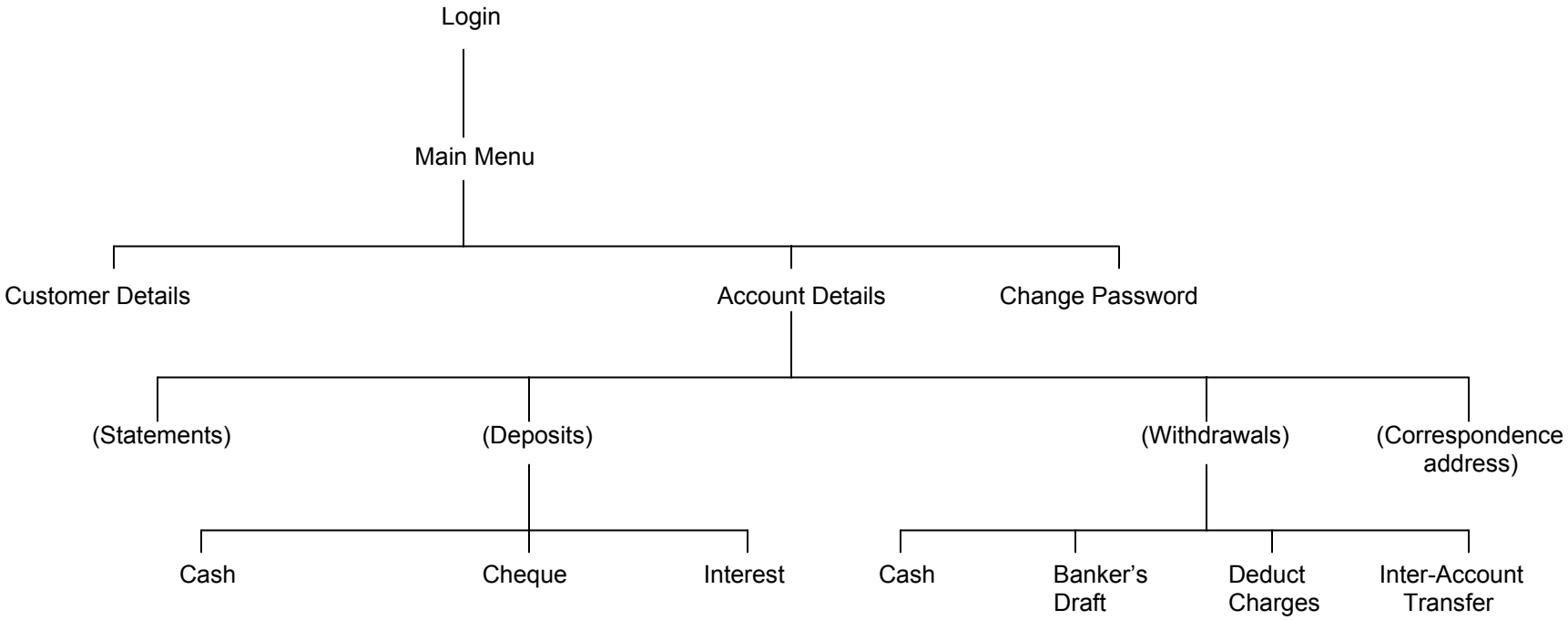


Figure 6.29: Change Password GUI

Figure 6.30: FAMS Application GUI Storyboard



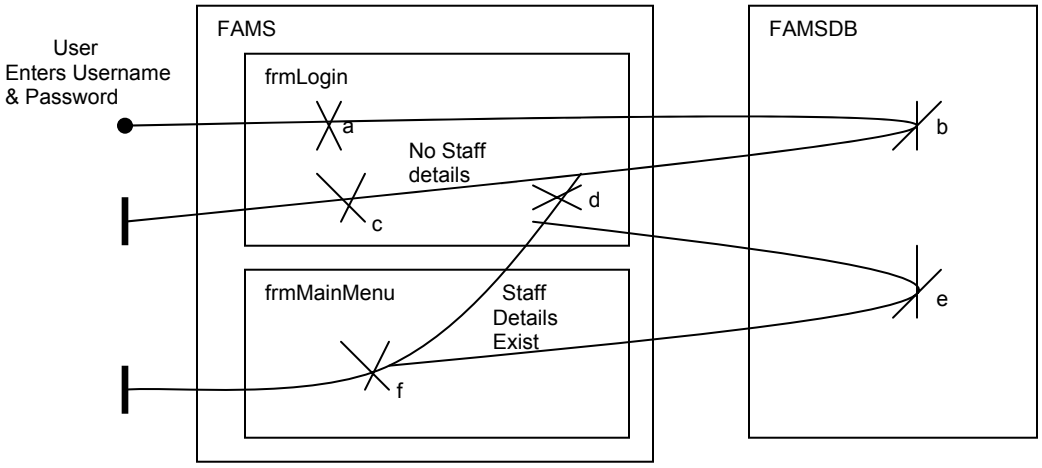
Chapter 7

UCM Views of the FAMS Application

This chapter contains the UCM views of the FAMS application (with the FAMSDB database). The chapter contains fifteen UCMs each of which models the scenarios of one of the use cases shown in Figure 6.1: Use Case Diagram for FAMS Application. The intension of this chapter is to discover if adding a UCM view of the system contributes to the overall design of the application.

7.1 "Login Staff" UCM

Figure 7.1: "Login Staff" UCM



Responsibilities

a. Read Username and Password Entered by the User
b. Select Staff Details
c. Display error message
d. Hide Form frmLogin
e. Add IP Address to database
f. Open Form frmMainMenu

Documentation for Figure 7.1: "Login Staff" UCM

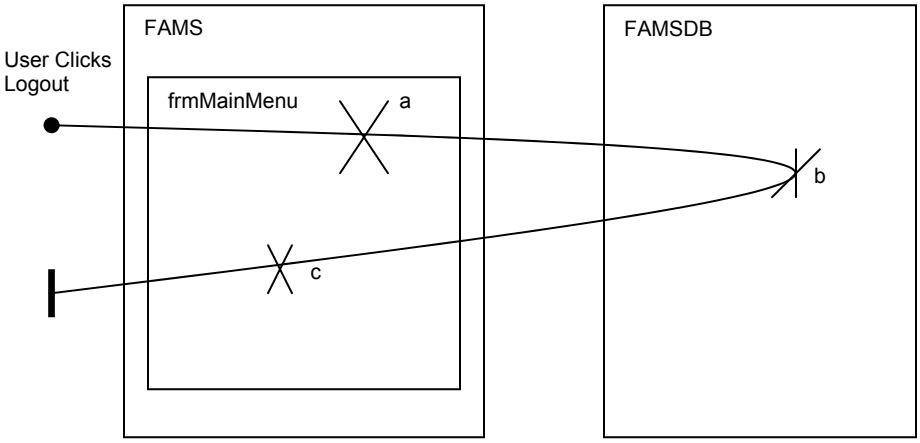
Figure 7.1 models the scenario where the user opens the FAMS application and is confronted with the Login Screen. The user then enters their username and password and clicks the Login button on the Login screen. The FAMS application then queries the FAMSDB database to see if there is a record in the Staff table which matches the username and password input by the user.

If no record exists an error message is displayed to the user.

If a record is found, the Login screen is hidden, an SQL UPDATE Command sends the IP address of the machine that the user is logged in on to the localIPAddress attribute of the Staff table of the FAMSDB database (so the DBA can tell the user is logged in) and the Main Menu screen of the FAMS application is opened.

7.2 “Logout Staff” UCM

Figure 7.2: “Logout Staff” UCM



Responsibilities

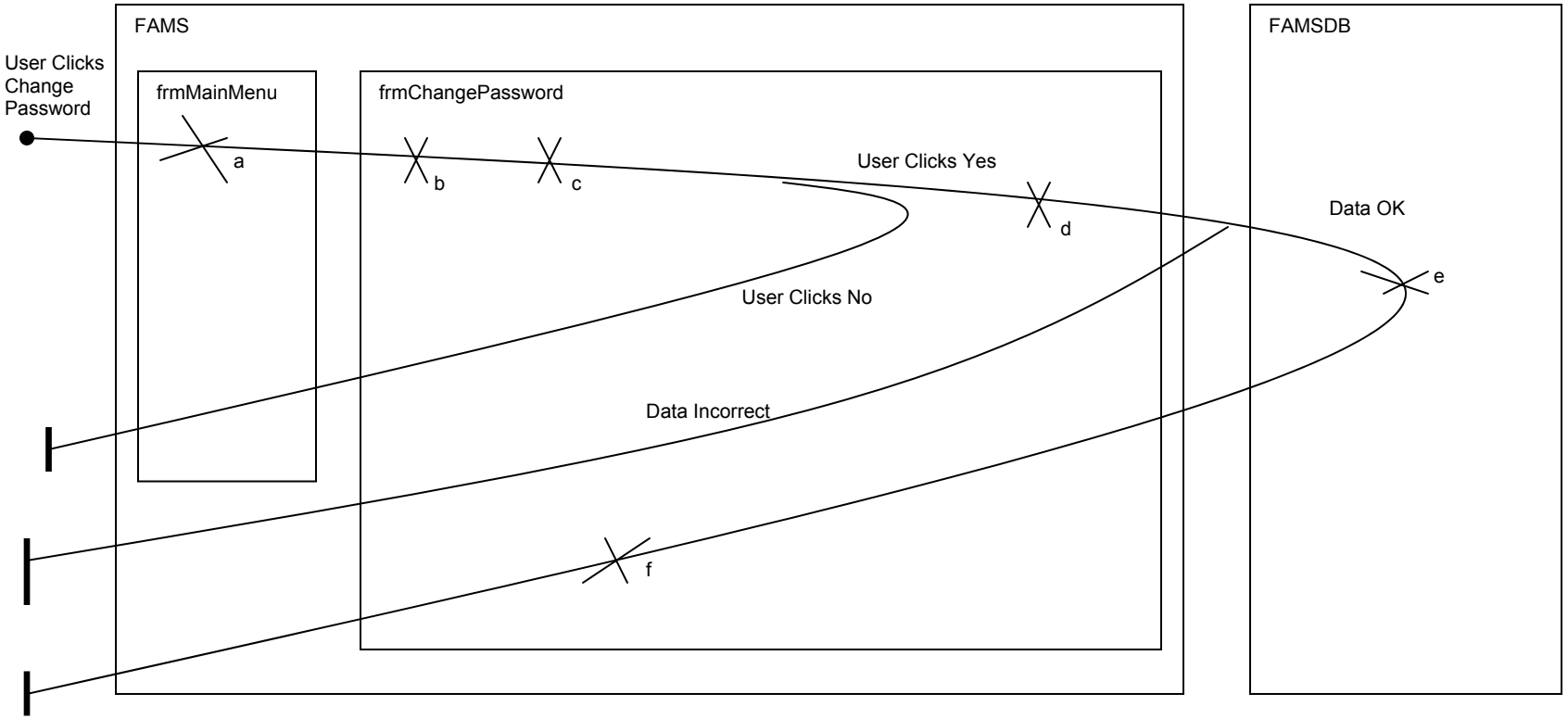
a. Call mnuLogout_Click() method
b. Delete IP Address from database
c. Exit Application

Documentation for Figure 7.2: “Logout Staff” UCM

Figure 7.2 models the scenario where the user clicks Logout in the menu in the Main Menu screen. When the user clicks Logout, an SQL UPDATE Command sets the localIPAddress attribute of the Staff table of the FAMSDB database to NULL (so the DBA can tell the user is not logged in), the Garbage Collector is called to free-up memory and the FAMS application is closed (this includes closing all FAMS application windows that are open).

7.3 “Change Password” UCM

Figure 7.3: “Change Password” UCM



Responsibilities

a. Open Form frmChangePassword
b. Read Old & New Passwords entered by user
c. Display Confirmation Message
d. Verify Data Entered by User
e. Update database
f. Display Password Changed Information Message

Documentation for Figure 7.3: “Change Password” UCM

Figure 7.3 models the scenario where the user clicks Change password in the menu in the Main Menu screen.

When the user clicks Change password the Change Password screen opens. The user then enters their Old Password, their New Password and they confirm their new password. The user then clicks the Change Password button on the Change Password screen. At this point the user is asked if they are sure they want to change their password.

If the user clicks No the change password screen closes.

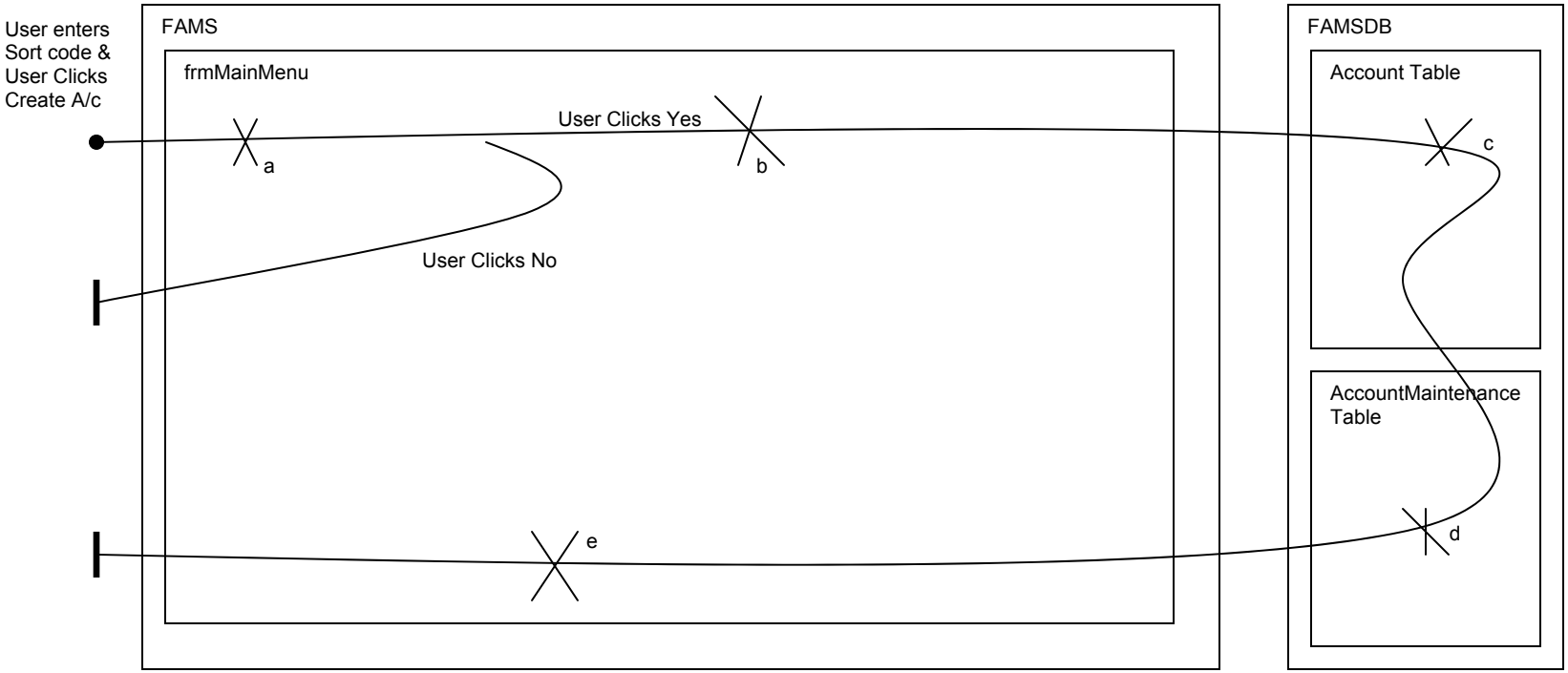
If the user clicks Yes, the data entered by the user is verified, i.e. the application checks that the Old Password, New Password and Confirm New Password fields are not empty, it verifies that the Old Password entered is in fact the user’s correct password and it checks that the user has entered the same values in the New Password and Confirm New Password fields.

Note that the SELECT query used to verify that the Old Password is the user’s correct password is elided in Figure 7.3.

Once the data entered by the user has been verified, if the data is incorrect an error message is displayed to the user. If the data entered by the user is OK an SQL UPDATE Command updates the Staff table of the FAMSDB database with the user’s new password and a confirmation message is displayed to the user.

7.4 “Open Account” UCM

Figure 7.4: “Open Account” UCM



Responsibilities

a. Display Confirmation Message
b. Randomly Generate New Account Number for the Sort Code Entered
c. Insert New Account Record
d. Insert New AccountMaintenance record
e. Display New Account Number in Main Menu screen

Documentation for Figure 7.4: “Open Account” UCM

Figure 7.4 models the scenario where the user enters an appropriate sort code and clicks the Create Account button in the Main Menu screen.

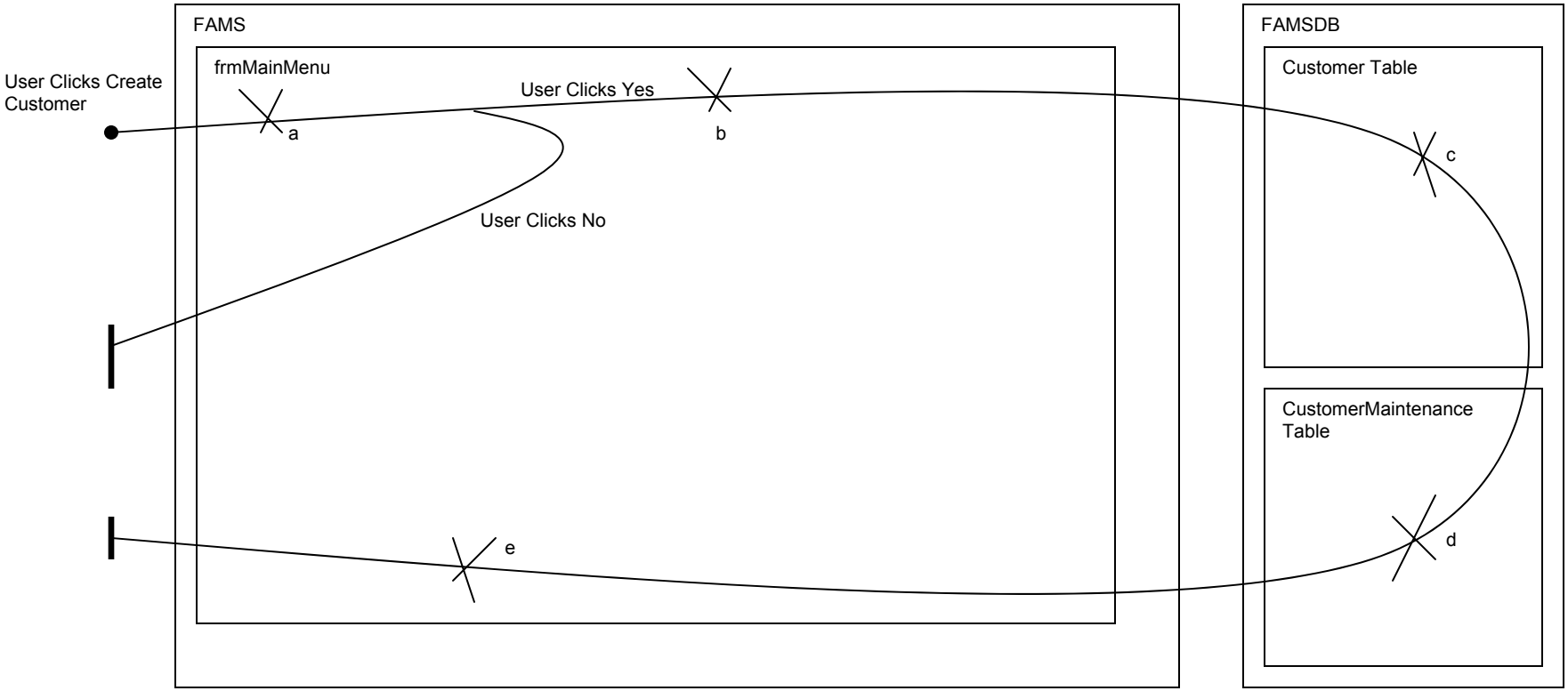
When the user clicks the Create Account button a confirmation message is displayed asking the user if they are sure they want to create a new account.

If the user answers No, no account number is created.

If the user answers Yes, a new account number (eight digit number in the range 10000000 to 99999999) for the sort code entered is randomly generated, an SQL INSERT Command inserts a new account record in the Account table of the FAMSDB Database and an SQL INSERT Command inserts a new record in the AccountMaintenance table of the FAMSDB database to indicate which member of staff created the new account and on which date they created it. The new account number is then displayed in the Account No.: field of the Main Menu screen.

7.5 “Create Customer” UCM

Figure 7.5: “Create Customer” UCM



Responsibilities

a. Display Confirmation Message
b. Randomly Generate New Customer Number
c. Insert New Customer Record
d. Insert New CustomerMaintenance Record
e. Display New Customer Number

Documentation for Figure 7.5: “Create Customer” UCM

Figure 7.5 models the scenario where the user clicks the Create Customer button in the Main Menu screen.

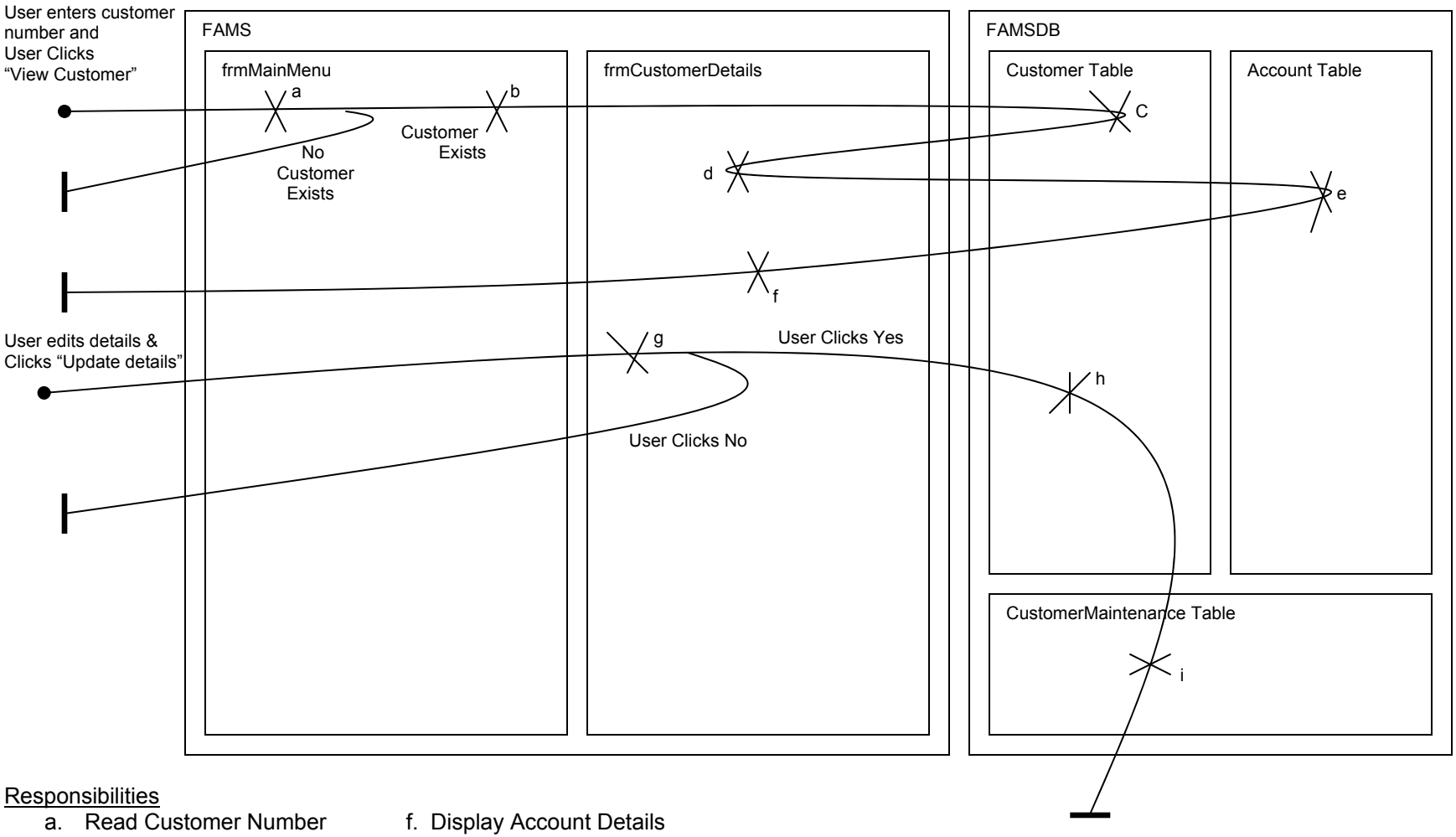
When the user clicks the Create Customer button a confirmation message is displayed asking the user if they are sure they want to create a new customer.

If the user clicks No, no customer number is created.

If the user clicks Yes, a new customer number (number in the range 1 to 2,147,483,647) is randomly generated, an SQL INSERT Command inserts a new customer record in the Customer table of the FAMSDB Database and an SQL INSERT Command inserts a new record in the CustomerMaintenance table of the FAMSDB database to indicate which member of staff created the new customer and on which date they created them. The new customer number is then displayed in the Customer No.: field of the Main Menu screen.

7.6 “View/Update Customer Details” UCM

Figure 7.6: “View/Update Customer Details” UCM



Responsibilities

a. Read Customer Number
b. Open frmCustomerDetails
c. Select Customer Details
d. Display Customer Details
e. Select Account Detail
f. Display Account Details
g. Display Confirmation Message
h. Update Customer Table
i. Update CustomerMaintenance Table

Documentation for Figure 7.6: “View/Update Customer Details” UCM

Figure 7.6 models the scenario where the user enters the customer’s number and then clicks the View Customer button in the Main Menu screen.

When the user clicks View Customer the application reads the customer number entered by the user and queries the FAMSDB database to see if there is a record for the customer number entered by the user. Note that the SQL SELECT Query for checking that there is a record corresponding to the customer number entered by the user is elided in Figure 7.6.

If no record corresponding to the customer number entered by the user if found an error message is displayed to the user.

If there is a record found, the Customer Details screen is opened and the FAMSDB database is queried in order to retrieve the customer’s details. The customer’s details are then displayed in the Customer Details screen. Then the Account table of the FAMSDB database is queried in order to find all the accounts that the customer is a signatory on, i.e. all the accounts where the customer is an account holder. The account details (sort code, account number and product type, e.g. *FlexiPlus*) are then displayed on the Customer Details screen.

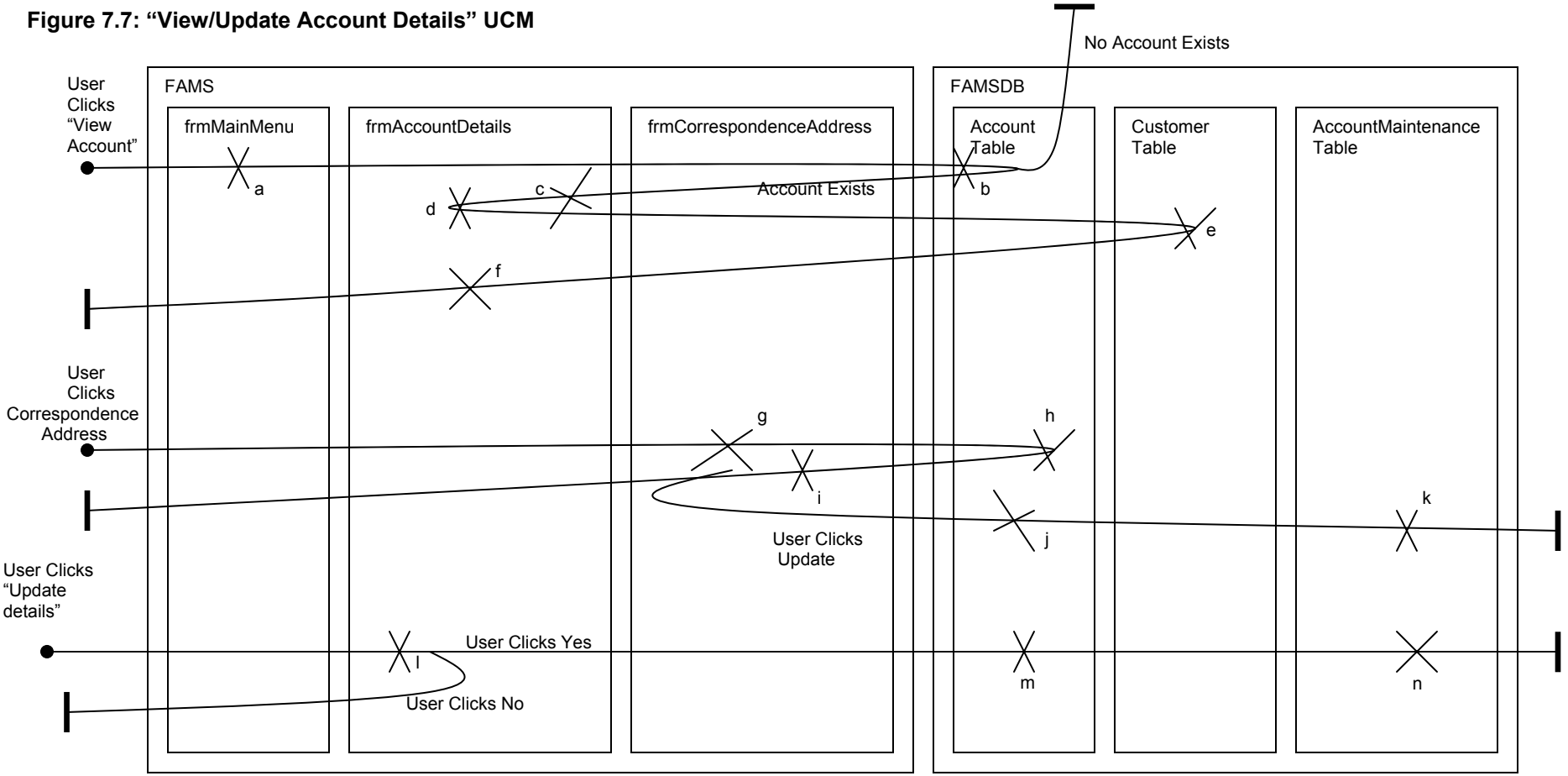
If the user edits the customer’s details and clicks Update details in the Customer Details screen, a confirmation message is displayed to the user asking if they are sure they want to update the customer’s details.

If the user answers No, the customer’s details are not updated.

If the user answers Yes, an SQL UPDATE Command updates the customer’s record in the Customer table of the FAMSDB database and an SQL INSERT Command inserts a record into the CustomerMaintenance table of the FAMSDB database indicating which member of staff updated the customer’s record and on which date they carried out the update.

7.7 “View/Update Account Details” UCM

Figure 7.7: “View/Update Account Details” UCM



Responsibilities

a. Read Sort Code & Account Number
b. Select Account Details
c. Open frmAccountDetails
d. Display Account Details
e. Select Customer Details
f. Display Customer Details
g. Open frmCorrespondenceAddress
h. Select Correspondence Address Details
i. Display Correspondence Address Details
j. Update Correspondence Address Details
k. Insert Account Maintenance Record
l. Display Confirmation Message
m. Update Account Details
n. Insert Account Maintenance Record

Documentation for Figure 7.7: “View/Update Account Details” UCM

Figure 7.7 models the scenario where the user enters the account sort code and the account number and clicks the View Account button on the Main Menu screen.

When the user clicks the View Account button the FAMS application reads the sort code and account number that the user entered and queries the FAMSDB database to select the account record that corresponds to the sort code and account number entered by the user.

If no record is found in the FAMSDB database, no account details are displayed.

If an account record is found, the Account Details screen is opened and the account details are displayed. Then, the Customer table of the FAMSDB database is queried to retrieve the account holder’s (signatories) names (titles, forenames, surnames) and they are displayed on the Account Details screen.

Then, if the user clicks Correspondence address on the menu in the Account Details screen, the Correspondence Address screen is opened which holds the account name and correspondence address details for the account. The Account table of the FAMSDB database is then queried to retrieve the account name and correspondence address data which is then displayed in the Correspondence Address screen.

If the user edits the account name and/or correspondence address details and clicks Update on the menu on the Correspondence Address screen an SQL UPDATE Command updates the Account table of the FAMSDB database and an SQL INSERT Command inserts a record in the AccountMaintenance table of the FAMSDB database to indicate which member of staff updated the details and on which date they did it.

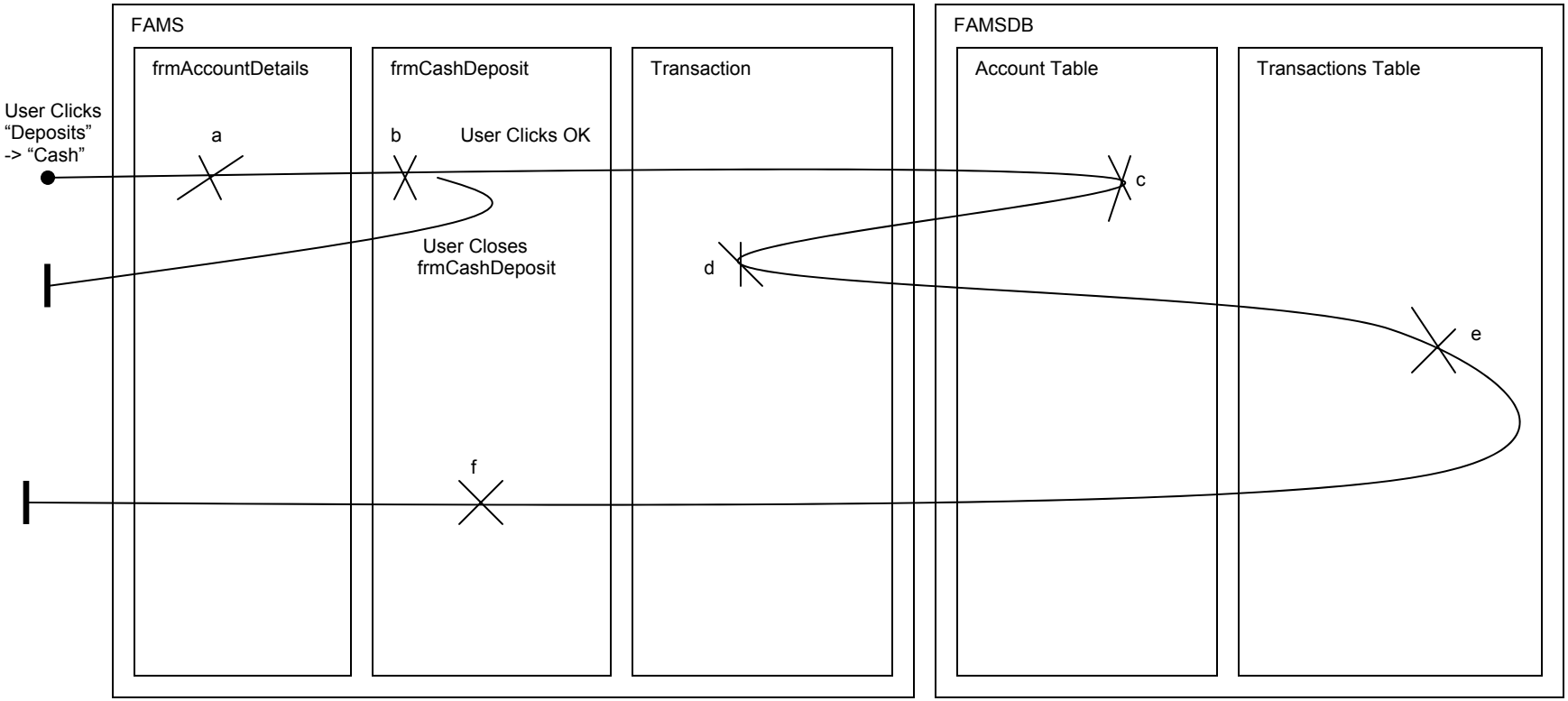
In the Account Details screen, if the user edits the account details and clicks Update details on the menu a confirmation message is displayed asking the user if they are sure they want to update the account details.

If they answer No, the account details are not updated.

If they answer Yes, an SQL UPDATE Command updates the Account table of the FAMSDB database with the new account details and an SQL INSERT Command inserts a new record in the AccountMaintenance table of the FAMSDB database to indicate which member of staff updated the account record and on which date they did it.

7.8 “Make Cash Deposit” UCM

Figure 7.8: “Make Cash Deposit” UCM



Responsibilities

a. Open frmCashDeposit
b. Read Amount Entered By User
c. Update Account Record
a. Select Transaction Type To Insert Into Transaction Table
b. Insert Transaction Record
c. Display Confirmation Message

Documentation for Figure 7.8: “Make Cash Deposit” UCM

Figure 7.8 models the scenario where the user clicks Deposits, then clicks Cash on the menu on the Account Details screen.

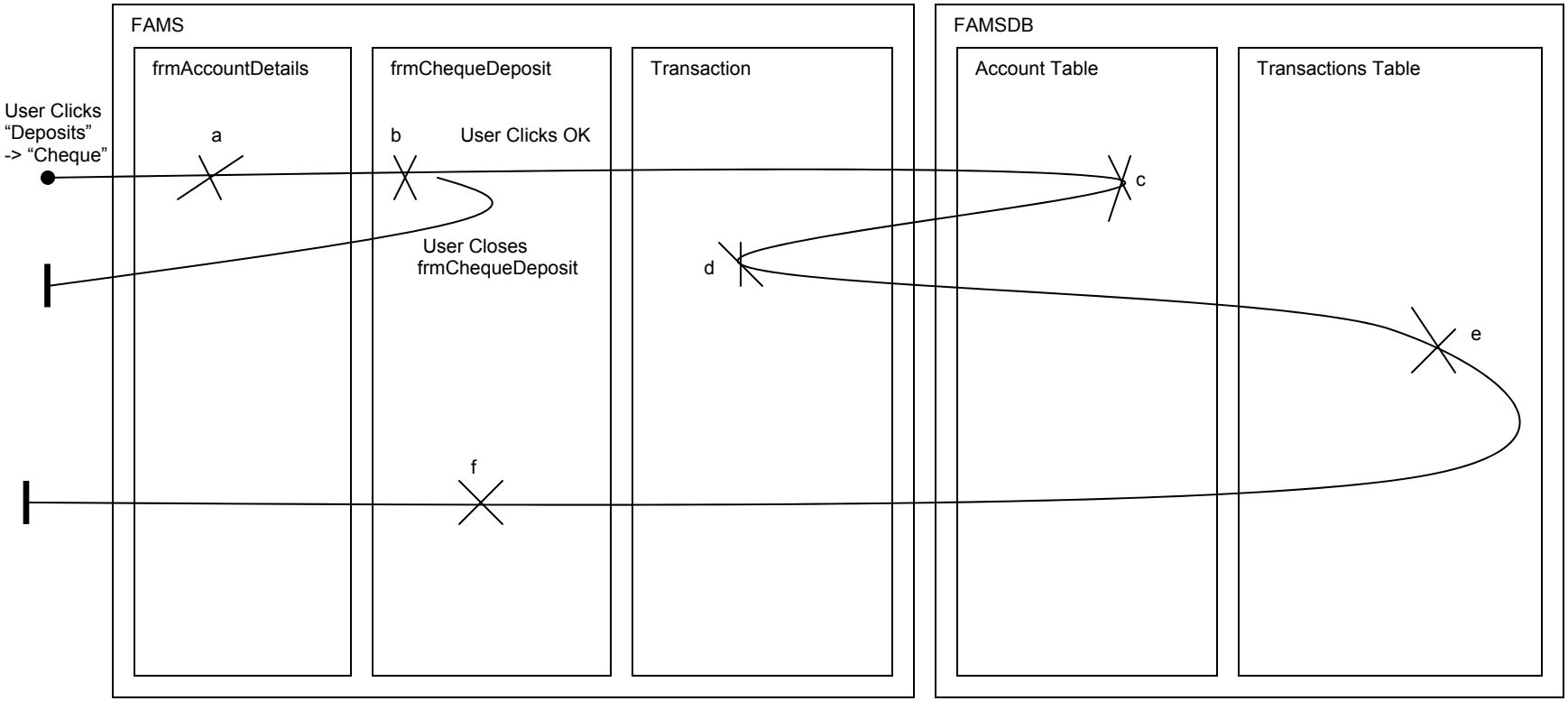
When the user clicks the Cash menu item on the Account Details screen the Cash Deposit screen opens and the user enters the amount that they wish to deposit in the Amount £ field of the Cash Deposit screen. The FAMS application then reads the amount that has been entered by the user.

If the user closes the Cash Deposit screen, no further action is taken.

If the user clicks OK, an SQL UPDATE Command updates the Account table of the FAMSDB database by adding the amount entered by the user to the account record’s balance and availableBalance attributes. Then, the InsertTransaction() method of class Transaction inserts a new transaction record (of type “CashDepsoit”) into the Transaction table of the FAMSDB database. Lastly, a confirmation message confirming the amount that has been deposited is displayed to the user.

7.9 “Make Cheque Deposit” UCM

Figure 7.9: “Make Cheque Deposit” UCM



Responsibilities

a. Open frmCashDeposit
b. Read Amount Entered By User
c. Update Account Record
d. Select Transaction Type To Insert Into Transaction Table
e. Insert Transaction Record
f. Display Confirmation Message

Documentation for Figure 7.9: “Make Cheque Deposit” UCM

Figure 7.9 models the scenario where the user clicks Deposits, then clicks Cheque on the menu on the Account Details screen.

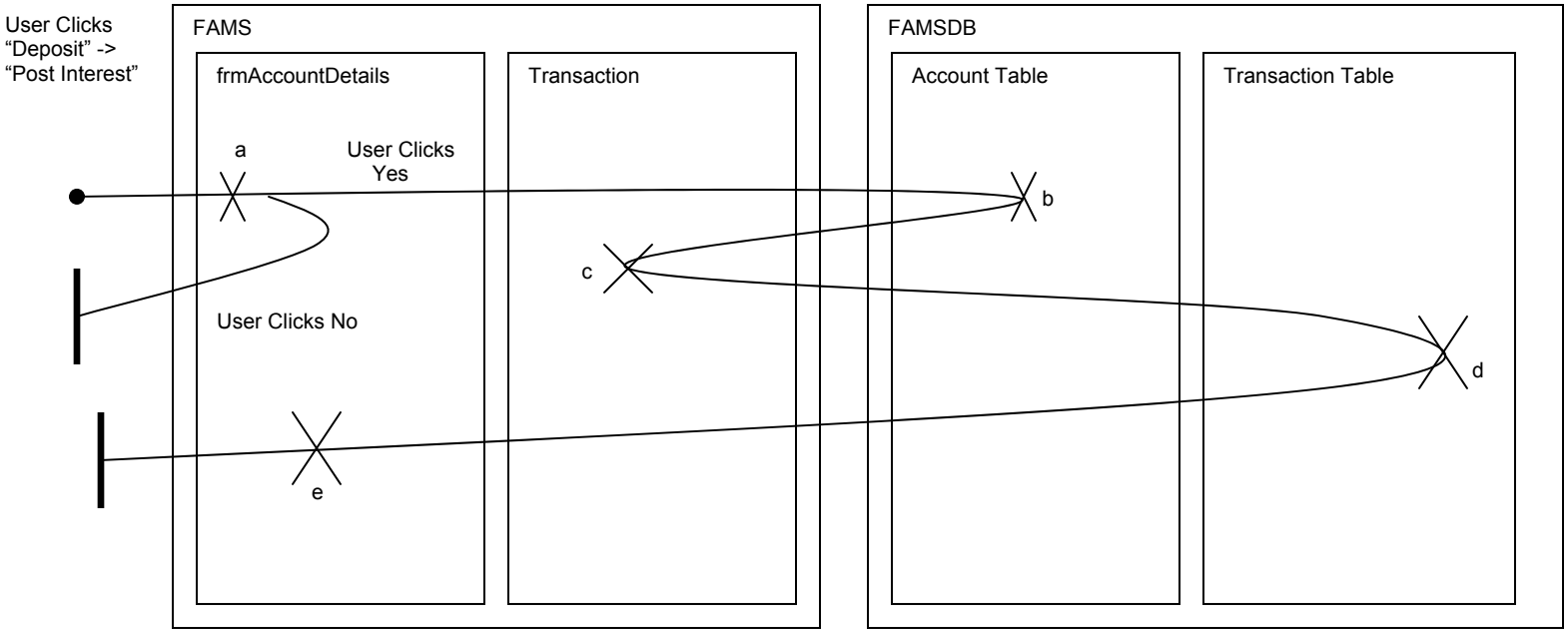
When the user clicks the Cheque menu item on the Account Details screen the Cheque Deposit screen opens and the user enters the amount that they wish to deposit in the Amount £ field of the Cheque Deposit screen. The FAMS application then reads the amount that has been entered by the user.

If the user closes the Cheque Deposit screen, no further action is taken.

If the user clicks OK, an SQL UPDATE Command updates the Account table of the FAMSDB database by adding the amount entered by the user to the account record’s balance attribute. Then, the InsertTransaction() method of class Transaction inserts a new transaction record (of type “ChequeDepsoit”) into the Transaction table of the FAMSDB database. Lastly, a confirmation message confirming the amount that has been deposited is displayed to the user.

7.10 “Post Interest” UCM

Figure 7.10: “Post Interest” UCM



Responsibilities

a. Display Confirmation Message
b. Update Account Record
c. Select Transaction Type To Insert Into Transaction Table
d. Insert Transaction Record
e. Display Confirmation Message

Documentation for Figure 7.10: “Post Interest” UCM

Figure 7.10 models the scenario where the user clicks Deposits, then clicks Post Interest on the menu on the Account Details screen.

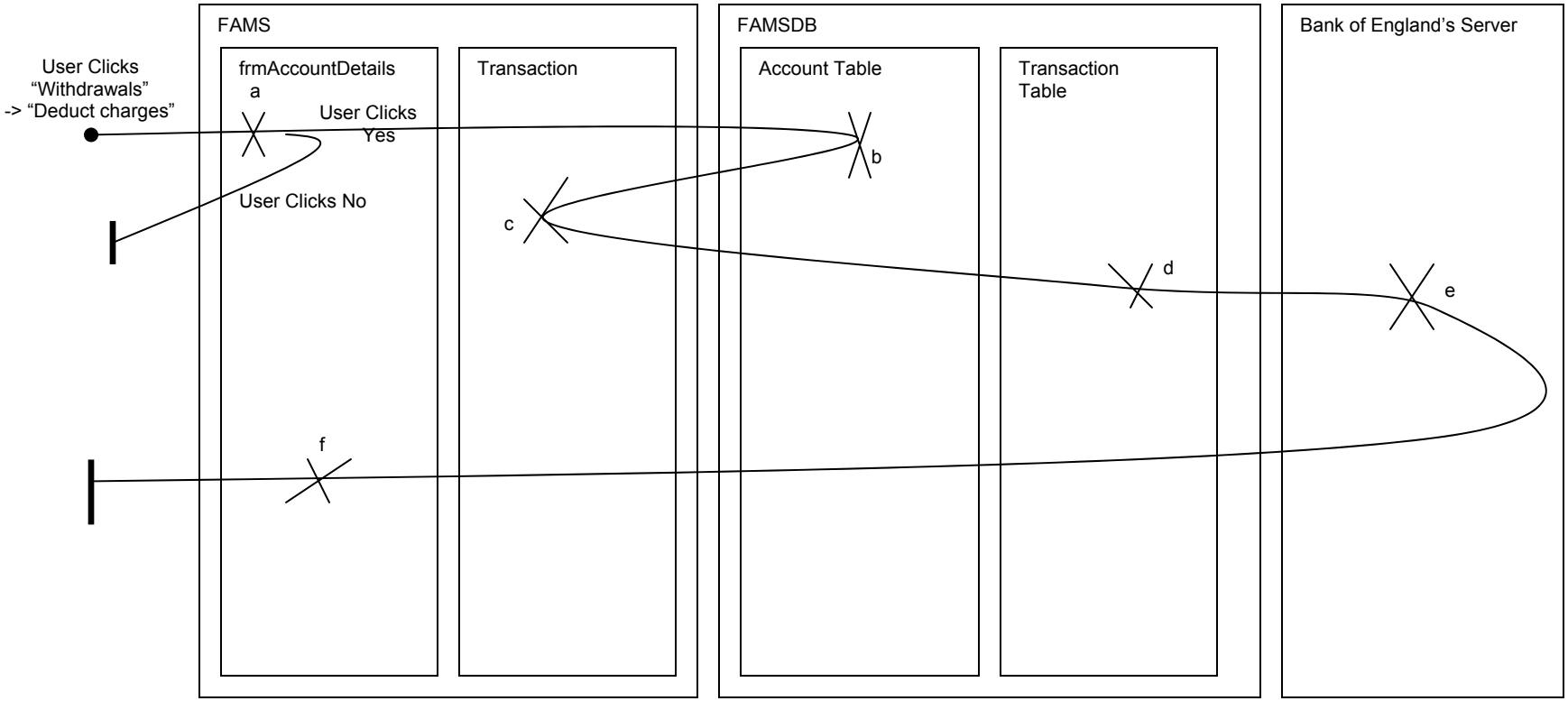
When the user clicks Post Interest a confirmation message is displayed asking the user if they are sure they want to post interest.

If the user clicks No, Interest is not posted to the account.

If the user clicks Yes, an SQL UPDATE Command updates the account record in the Account table of the FAMSDB database by adding the value of the account record’s totalInterest attribute to the account record’s balance and availableBalance attributes. The totalInterest attribute is then set to zero. Then, the InsertTransaction() method of class transaction inserts a transaction of type “Interest” into the Transaction table of the FAMSDB database and a message confirming that the interest has been posted is displayed to the user.

7.11 “Deduct Charges” UCM

Figure 7.11: “Deduct Charges” UCM



Responsibilities

a. Display Confirmation Message
b. Update Account Record
c. Select Transaction Type To Insert Into Transaction Table
d. Insert Transaction
e. Update *BestBank’s* Bank of England Account via SWIFTNet connection
f. Display Confirmation Message

Documentation for Figure 7.11: “Deduct Charges” UCM

Figure 7.11 models the scenario where the user clicks Withdrawals, then clicks Deduct charges on the menu on the Account Details screen.

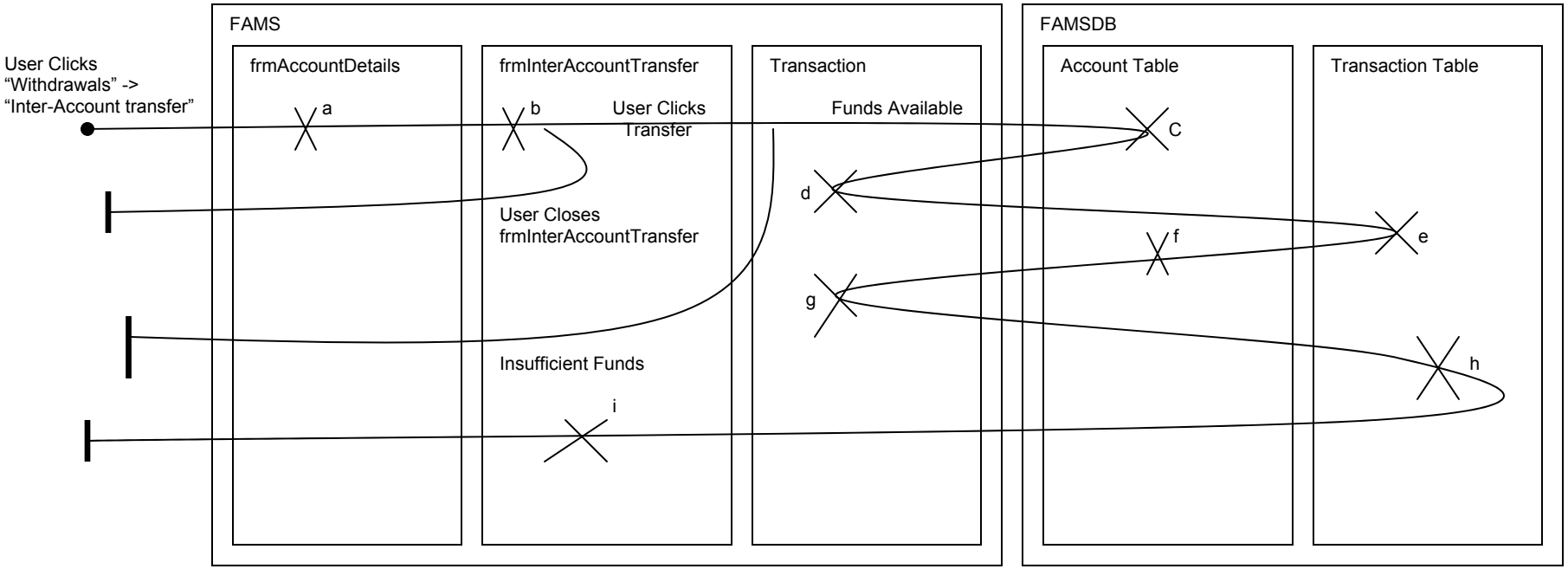
When the user clicks Deduct charges a confirmation message is displayed asking the user if they are sure they want to deduct charges.

If the user clicks No, charges are not deducted from the account.

If the user clicks Yes, an SQL UPDATE Command updates the account record in the Account table of the FAMSDB database by subtracting the value of the account record’s charges attribute from the account record’s balance and availableBalance attributes. The charges attribute is then set to zero. Then, the InsertTransaction() method of class Transaction inserts a transaction of type “Charges” into the Transaction table of the FAMSDB database and *BestBank’s* Bank of England account is updated to show that the charges have been taken. A message confirming that the charges have been deducted is then displayed to the user.

7.12 “Make Inter-Account Transfer” UCM

Figure 7.12: “Make Inter-Account Transfer” UCM



Responsibilities

a. Open frmInterAccountTransfer
b. Read Collecting Account’s Sort Code, Account Number And Amount To Be Transferred
c. Update Paying Account’s Record
d. Select Transaction Type (For Paying Account) To Insert Into Transaction Record
e. Insert Transaction Record (For Paying Account)
f. Update Collecting Account’s Record
g. Select Transaction Type (For Collecting Account) To Insert Into Transaction Record
h. Insert Transaction Record (For Collecting Account)
i. Display Confirmation Message

Documentation for Figure 7.12: “Make Inter-Account Transfer” UCM

Figure 7.12 models the scenario where the user clicks Withdrawals, then clicks Inter-account transfer on the menu on the Account Details screen.

When the user clicks Inter-account transfer the Inter-Account Transfer screen opens and the user enters the collecting account’s sort code and account number and the amount that is to be transferred. The FAMS application the reads the user’s input (collecting sort code and account number and amount).

If the user closes the Inter-Account Transfer form, no money is transferred.

If the user clicks Transfer and there are sufficient funds available in the paying account, firstly, the paying account’s record is updated in the Account table of the FAMSDB database by deducting the amount of the transfer from the paying account record’s balance and availableBalance attributes. Then, a transaction of type “PayTransfer” is inserted into the transaction table of the FAMSDB database for the paying account.

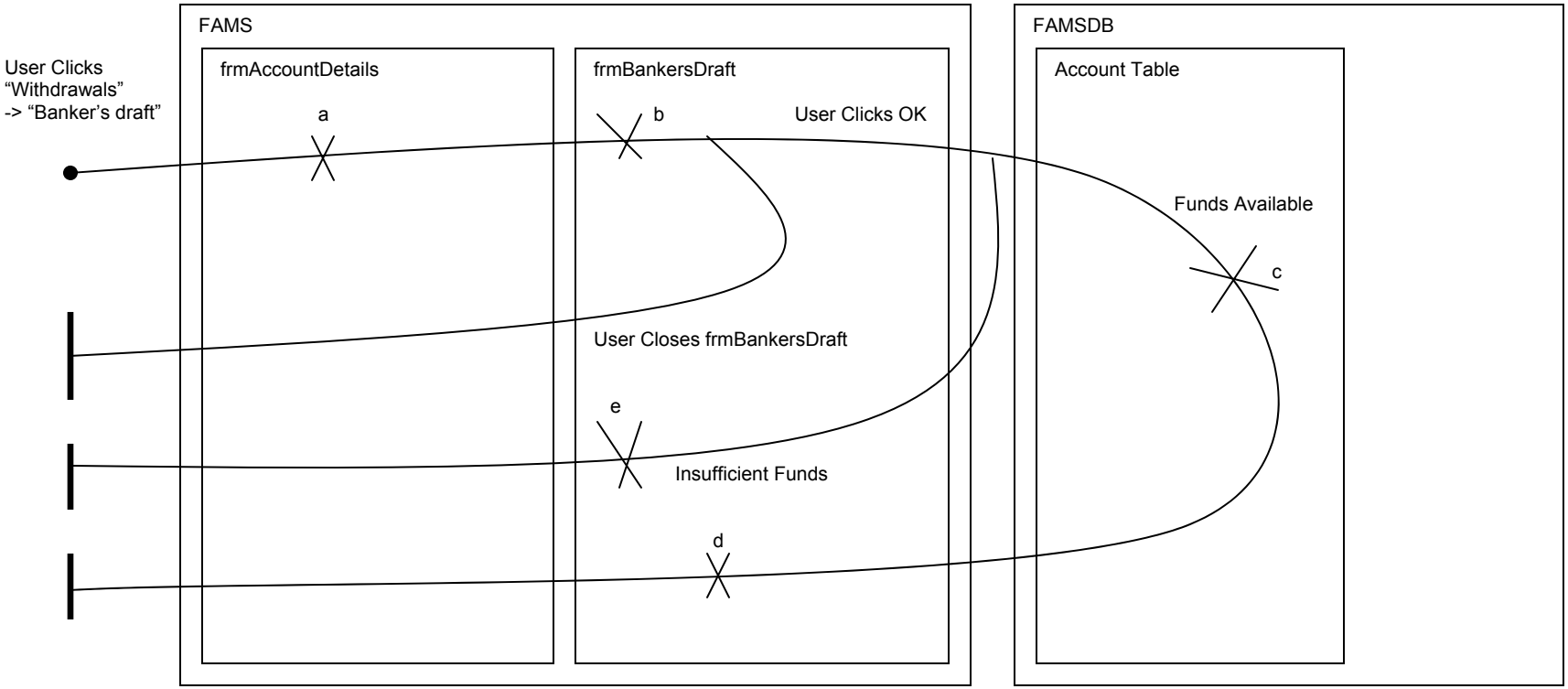
Secondly, the collecting account’s record is updated in the Account table of the FAMSDB database by adding the amount of the transfer to the collecting account record’s balance and availableBalance attributes. Then, a transaction of type “ReceiveTransfer” is inserted into the transaction table of the FAMSDB database for the collecting account.

Lastly, a message confirming that the transfer has successfully completed is displayed to the user.

If there are insufficient funds available in the paying account a message is displayed to the user telling them that there are insufficient funds for the transfer.

7.13 “Issue Banker’s Draft” UCM

Figure 7.13: “Issue Banker’s Draft” UCM



Responsibilities

a. Open frmBankersDraft
b. Read Amount Entered By User
c. Update Account Record
d. Display Confirmation Message
e. Display Warning Message

Documentation for Figure 7.13: “Issue Banker’s Draft” UCM

Figure 7.13 models the scenario where the user clicks Withdrawals, then clicks Banker’s draft on the menu on the Account Details screen.

When the user clicks Bankers draft the Banker’s Draft screen opens and the user enters the amount that is to be issued as a banker’s draft from the account. The FAMS application then reads the amount that is to be withdrawn.

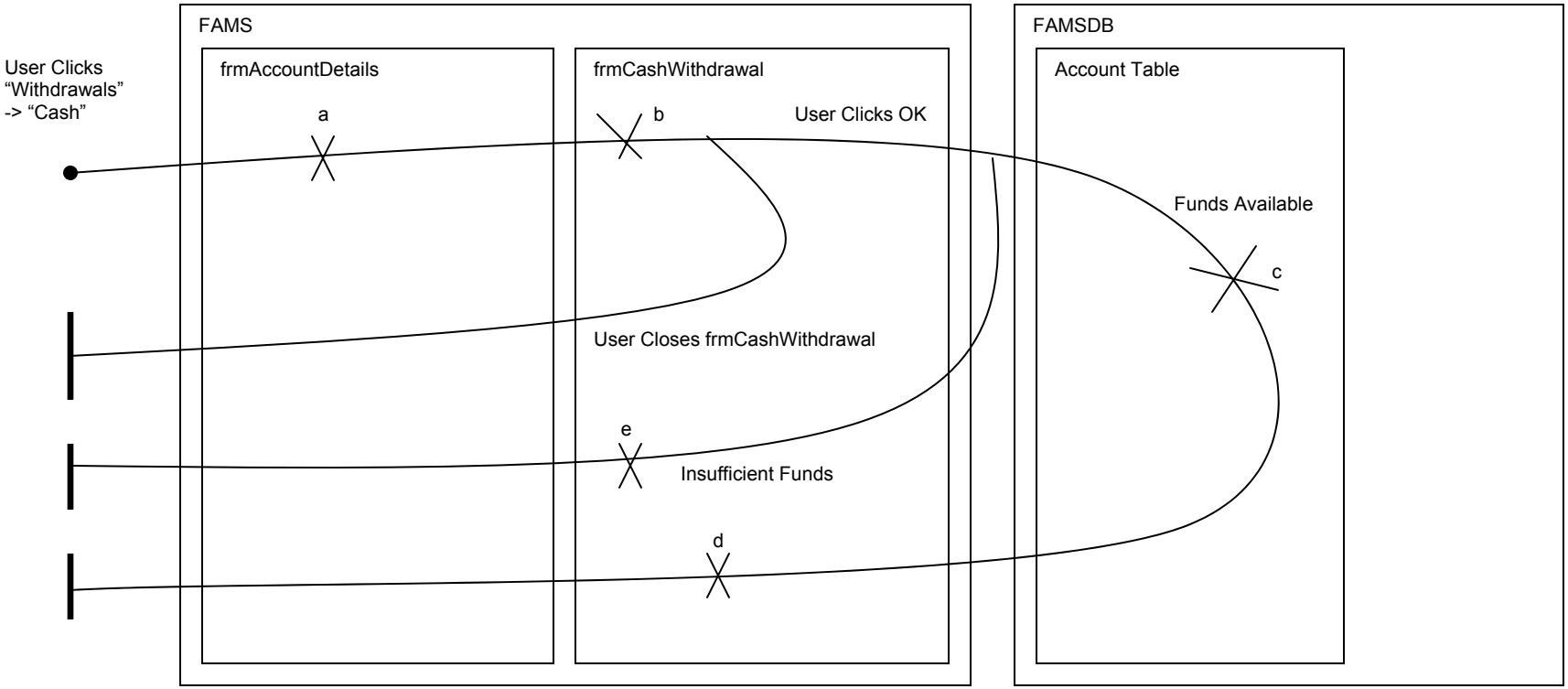
If the user closes the Banker’s Draft screen no money is withdrawn.

If the user clicks OK and there are sufficient funds available in the account, the account’s record is updated in the Account table of the FAMSDB database by deducting the amount of the withdrawal from the account record’s balance and availableBalance attributes. Then, a message confirming the withdrawal is displayed to the user.

If there are insufficient funds available, a message is displayed to the user indicating that there are not enough funds available on the account for the withdrawal.

7.14 “Make Cash Withdrawal” UCM

Figure 7.14 “Make Cash Withdrawal” UCM



Responsibilities

a. Open frmBankersDraft
b. Read Amount Entered By User
c. Update Account Record
d. Display Confirmation Message
e. Display Warning Message

Documentation for 7.14 “Make Cash Withdrawal” UCM

Figure 7.14 models the scenario where the user clicks Withdrawals, then clicks Cash on the menu on the Account Details screen.

When the user clicks Cash the Cash Withdrawal screen opens and the user enters the amount that is to be withdrawn from the account. The FAMS application then reads the amount that is to be withdrawn.

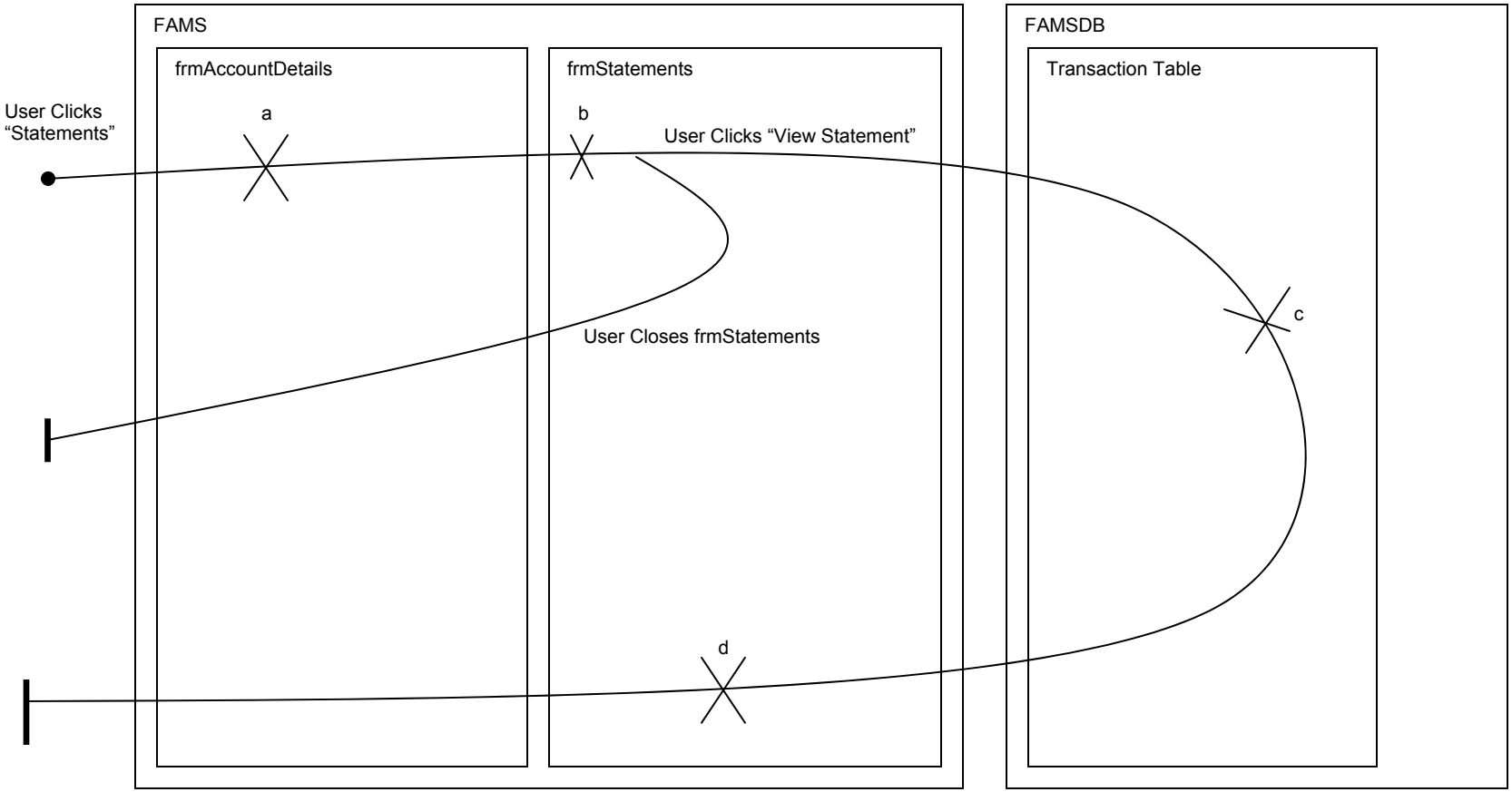
If the user closes the Cash Withdrawal screen no money is withdrawn.

If the user clicks OK and there are sufficient funds available in the account, the account’s record is updated in the Account table of the FAMSDB database by deducting the amount of the withdrawal from the account record’s balance and availableBalance attributes. Then, a message confirming the withdrawal is displayed to the user.

If there are insufficient funds available, a message is displayed to the user indicating that there are not enough funds available on the account for the withdrawal.

7.15 “View Statement” UCM

Figure 7.15: “View Statement” UCM



Responsibilities

a. Open frmStatements
b. Read “Start Date” And “Finish Date” If The User Has Entered Them
c. Select Transactions To Be Displayed
d. Display Transactions

Documentation for Figure 7.15: “View Statement” UCM

Figure 7.15 models the scenario where the user clicks Statements on the menu on the Account Details screen.

When the user clicks Statements, the Statements screen opens. The user can view all transactions carried out on the account by simply clicking View Statement or the user can view a subset of all transactions by entering a start date (in the from: field) and/or a finish date (in the To: field).

When the user clicks View Statement in the Statements screen, all transactions are selected from the Transaction table of the FAMSDB database and are displayed in the Statements screen or a subset of all transactions are displayed if the user has entered a start date and/or a finish date.

The user can close the Statements screen prior to clicking View Statements, in which case no transactions are viewed,

Chapter 8

Implementation of FAMS Application

This chapter covers issues relating to the implementation of the FAMS application.

8.1 Data Validation
In practice all data would be validated before being sent to the FAMSDB database. For example, the format of dates would be checked to confirm that the data entered was in fact a date and not some other string.

In order to simplify the implementation, data validation has been omitted in the prototype FAMS application.

8.2 Defensive Programming
In practice defensive programming would have been used when coding the FAMS application in order to protect against user errors and malicious users. However, while there has been some defensive programming carried out, it has not been rigorously employed during the coding of the prototype FAMS application.

8.3 Transaction Management
Concurrency issues must be considered. That is, we must prevent two or more account holders simultaneously carrying out transactions on one account as this could potentially cause data corruption in the FAMSDB database.

Microsoft Jet 4.0 (and Microsoft Access 2002) uses lock files to prevent data corruption. A lock file has extension .ldb and is used to keep multi-user databases from being changed in the same place by two people at the same time, resulting in data corruption. Thus, there is no need to worry about concurrency issues, such as the lost update problem, with this implementation. If we were using *BestBank's* HP NonStop SQL Database record locking would be used to prevent concurrency conflicts.

8.4 FAMSDB Database Tables and their Function

Table	Function
Account	Holds all data pertaining to customers' accounts including account balance, account name and correspondence address, overdraft limit, customer numbers of account signatories (holders), etc.
AccountMaintenance	Holds records of which members of staff created accounts or updated account details, such as overdraft limits, adding account holders, etc, and on which date they carried out the transaction on the account data.
Card	Holds data pertaining to cards issued to account holders (this table is not used in the implementation of the system as it would involve the use of card-reading devices which is beyond the scope of this paper).
Customer	Holds all data pertaining to customers, such as customers' names and addresses, dates of birth, dates of death if appropriate, telephone numbers, emails, etc.
CustomerMaintenance	Holds records of which members of staff created or updated customer details, such as customers' names and addresses, telephone numbers, emails, etc, and on which date they carried out the transaction on the customer data.

Table	Function
Staff	Holds records of members of staff that are authorise to use the FAMS application – this includes their names, usernames, passwords and staff numbers
Transaction	Holds records of every transaction carried out on every account managed by the FAMS application.

8.5 FAMS Class Methods and their Function

Class	Method	Function
frmLogin	cmdLogin_Click	Checks that the user is registered in the FAMSDB database and logs the user in if they are authorised users. Also, sends the IP Address of the machine that the user logs in on to the Staff.localIPAddress attribute of the FAMSDB.
frmLogin	GetIPAddress	Returns the IP address of the machine that the user is logged in on
frmMainMenu	mnuLogout_Click	Logs the user out of the FAMS application, closes all open application windows and deletes the local machines IP Address from the FAMSDB database.
frmMainMenu	frmMainMenu_Load	Disables the close button when the form is opened
frmMainMenu	frmMainMenu_SizeChanged	Disables the close button when the form is resized.
frmMainMenu	mnuChangePassword_Click	Opens instance of form frmChangePassword
frmMainMenu	cmdCreateCustomer_Click	Calls NewCustomerNo() method and inserts record in CustomerMaintenance Table of FAMSDB
frmMainMenu	cmdViewCustomer_Click	Opens form frmCustomerDetails and loads the data for the customer whose number is entered into the form so it can be viewed.
frmMainMenu	cmdCreateAccount_Click	Calls NewAccountNo() method and inserts record in AccountMaintenance Table of FAMSDB.
frmMainMenu	cmdViewAccount_Click	Opens form frmAccountDetails and loads the data for the account whose sort code and account number are entered into the form so it can be viewed.
frmMainMenu	NewCustomerNo	Randomly generates a new customer number, inserts record in Customer table of FAMSDB and displays New Customer Number in frmMainMenu
frmMainMenu	NewAccountNo	Randomly generates a new account number, inserts record in Account Table of FAMSDB and displays New Account Number in frmMainmenu
frmChangePassword	cmdChangePassword_Click	Allows the user to change their password
frmChangePassword	frmChangePassword_Load	Sets all passwords to the empty string “”.
CloseButton	Disable	Allows the close button on forms to be disabled
frmCustomerDetails	mnuExit_Click	Closes instance of form frmCustomerDetails
frmCustomerDetails	mnuUpdateDetails_Click	Updates the database with the customer’s new data after the user has edited the customer’s details and enters a record in the CustomerMaintenance Table of the FAMSDB.

Class	Method	Function
frmAccountDetails	mnuCorrespondneceAddress_Click	Opens form frmCorrespondenceAddress and loads the account name and correspondence address data into the form so it can be viewed.
frmAccountDetails	mnuExit_Click	Closes the form
frmAccountDetails	mnuUpdate_Click	Checks that the account balance is zero if the user is closing the account, Updates account details such as overdraft limit, notes, etc and inserts a record in the AccountMaintenance Table of the FAMSDB database.
frmAccountDetails	mnuItmCashDeposit_Click	Opens instance of frmCashDeposit and passes the account's sort code and account number to it.
frmAccountDetails	mnuItmCheque_Click	Opens instance of frmChequeDeposit and passes the account's sort code and account number to it.
frmAccountDetails	mnuItmPostInterest_Click	Allows the user to post interest to the account by adding the value of the account record's totalInterest attribute to the account record's balance and availableBalance attributes and then setting the totalInterest attribute to zero.
frmAccountDetails	mnuItmCashWithdrawal_Click	Opens instance of frmCashWithdrawal and passes the account's sort code and account number to it.
frmAccountDetails	mnuItmBankersDraft_Click	Opens instance of frmBankersDraft and passes the account's sort code and account number to it.
frmAccountDetails	mnuItmInterAccountTransfer_Click	Opens instance of frmInterAccountTransfer and passes the account's sort code and account number to it.
frmAccountDetails	mnuItmDeductCharges_Click	Allows the user to deduct charges from the account by subtracting the value of the account record's charges attribute from the account record's balance and availableBalance attributes and then setting the charges attribute to zero.
frmAccountDetails	mnuStatements_Click	Opens instance of frmStatements and passes the account's sort code and account number to it.
frmCorrespondenceAddress	mnuExit_Click	Closes the form
frmCorrespondenceAddress	mnuUpdate_Click	Updates the database with the new account name and correspondence address after the user has edited the data and enters a record in the AccountMaintenance Table of the FAMSDB database
Transaction	InsertTransaction	Inserts records for all types of transaction in the Transaction table of the FAMSDB database
frmCashDeposit	cmdCashDeposit_Click	Allows the user to make a cash deposit by Updating the Account Table and Inserting a record in the Transaction Table (by calling Transaction.InsertTransaction() method) of the FAMSDB database.

Class	Method	Function
frmChequeDeposit	cmdChequeDeposit_Click	Allows the user to make a cheque deposit by Updating the Account Table and Inserting a record in the Transaction Table (by calling Transaction.InsertTransaction() method) of the FAMSDB database. Note that Cheque deposits do not alter the availableBalance attribute of the account table as Cheques take 3 days to clear and BestBank's Clearing Centre are responsible for making cleared funds available.
frmInterAccountTransfer	cmdTransfer_Click	Allows the user to transfer money out of the (paying) account and into another (collecting) account, provided there are sufficient funds available. It also Inserts transactions for each account.
frmBankersDraft	cmdBankersDraft_Click	Allows the user to withdraw funds from accounts by subtracting the amount entered by the user from the account records balance and availableBalance attributes, provided there are sufficient funds available. It also inserts a transaction record in the Transaction Table.
frmCashWithdrawal	cmdCashWithdrawal_Click	Allows the user to withdraw funds from accounts by subtracting the amount entered by the user from the account records balance and availableBalance attributes, provided there are sufficient funds available. It also inserts a transaction record in the Transaction Table.
frmStatements	frmStatements_Load()	Initialises TextBoxes txtStartDate and txtFinishDate on frmStatements to the empty string “”.
frmStatements	cmdViewStatement_Click	Allows the user to view all transactions carried out on an account or a subset of these determined by the dates entered in From and To TextBoxes.

8.6 Deviations from the design

1) The class frmCustomerDetails was designed to have a method frmCustomerDetails_Load(). However, all the code that was to be incorporated into this method was included in the method cmdViewCustomer_Click of class frmMainmenu. Hence, there is no frmCustomerDetails_Load() method.

2) In method mnuUpdateCustomerDetails_Click() of class frmCustomerDetails the following piece of pseudo-code was to be implemented:

“Set transactionNumber = SELECT MAX(transactionNo)
FROM CustomerMaintenance
WHERE staffNo = StaffNumber AND Customer No = “Customer Number”

However, as the variable transactionNo was a string this did not work in practice and so the following piece of Visual Basic code was used:

```
CustomerMaintenanceDataAdapter.SelectCommand.CommandText = _
"SELECT COUNT(*) FROM CustomerMaintenance " & _
"WHERE staffNo = '" & staffNumber & "' AND customerNo = '" & _
txtCustomerNo.Text & "'"
```

3) The class frmAccountDetails was designed to have a method frmAccountDetails_Load(). However, all the code that was to be incorporated into this method was included in the method cmdViewAccount_Click() of class frmMainMenu. Hence, there is no frmAccountDetails_Load() method.

4) The class frmCorrespondenceAddress was designed to have a method frmCorrespondenceAddress_Load(). However, all the code that was to be incorporated into this method was included in the method mnuCorrespondenceAddress_Click() of class frmAccountDetails. Hence, there is no frmCorrespondenceAddress_Load() method.

5) In method mnuUpdate_Click of class frmCorrespondenceAddress the following piece of pseudo-code was to be implemented:

Set transactionNumber = SELECT MAX(transactionNo)
FROM AccountMaintenance
WHERE staffNo = StaffNumber AND sortCode = “Sort Code” AND
accountNo = “Account Number”

However, as the variable transactionNo was a string this did not work in practice and so the following piece of Visual Basic code was used:

```
AccountMaintenanceDataAdapter.SelectCommand.CommandText = _
"SELECT COUNT(*) FROM AccountMaintenance WHERE " & _
"sortCode = '" & txtSortCode.Text & "' AND " & _
"accountNo = '" & txtAccountNo.Text & "' AND " & _
"staffNo = '" & StaffNumber & "'"
```

6) Method cmdOK_Click() of class frmCashDeposit has been called cmdCashDeposit_Click().

7) Method cmdOK_Click() of class frmChequeDeposit has been called cmdChequeDeposit_Click().

8) Method cmdOK_Click() of class frmBankersDraft has been called cmdBankersDraft_Click()

9) Method cmdOK_Click() of class frmCashwithdrawal has been called cmdCashWithdrawal_Click().

10) The design of some of the GUIs has been altered in that all transaction forms, such as frmCashDeposit, frmChequeDeposit, etc, display the account’s sort code and account number on them.

11) The TextBoxes containing the Customer Numbers and names of Account Holders in the form frmAccountDetails have not been disabled. This is for ease of reading.

12) Method mnuUpdate_Click of class frmAccountDetails prevents the user from closing accounts if their balances are not zero, i.e. it stops the user closing accounts with non-zero balances, as well as performing the other tasks which were set out in the design.

8.7 Installing the FAMS Application with the FAMSDB Database

The user should, firstly, go to folder “FAMSDB and FAMS Application”, then folder “FAMS Installer”, then folder “Debug” and finally double-click “Setup.exe”. Follow the instructions in the Setup Wizard to install the FAMS Application with the FAMSDB Database in the folder of the user’s choice.

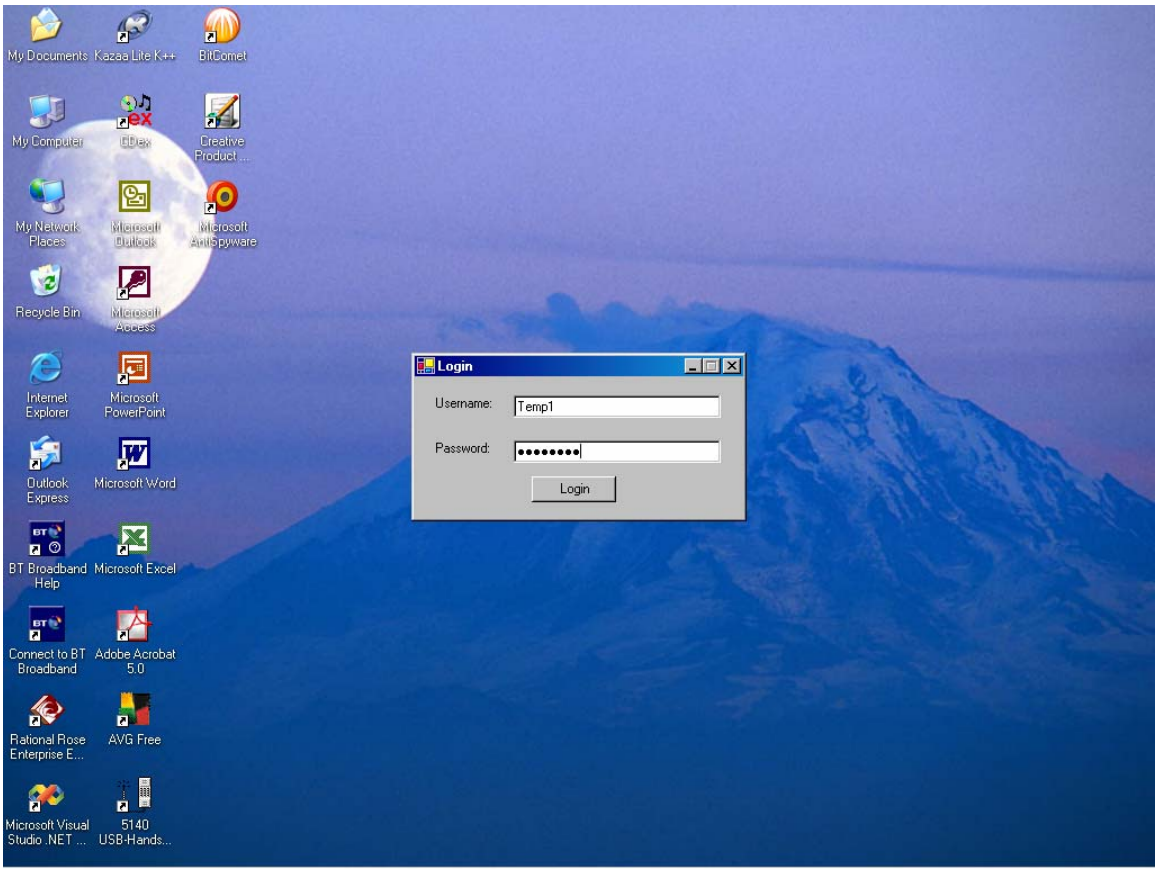
Once installed, the FAMS.exe file and the FAMSDB.mdb file will be in the same folder.

Double-click the “FAMS.exe” file to start the application.

8.8 Examples of the FAMS Application in use

8.8.1 Logging In

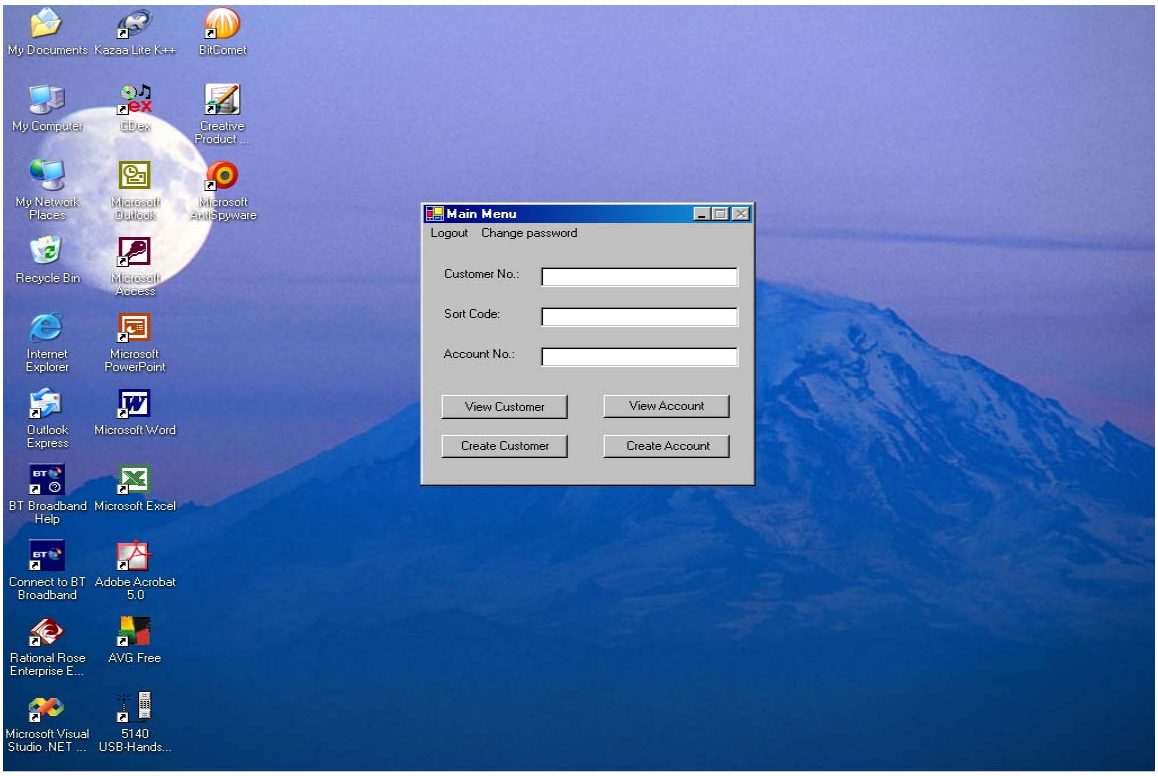
When the user double-clicks the FAMS.exe file the following login screen opens:



The username to be used is **Temp1** and initially the password is **password**.

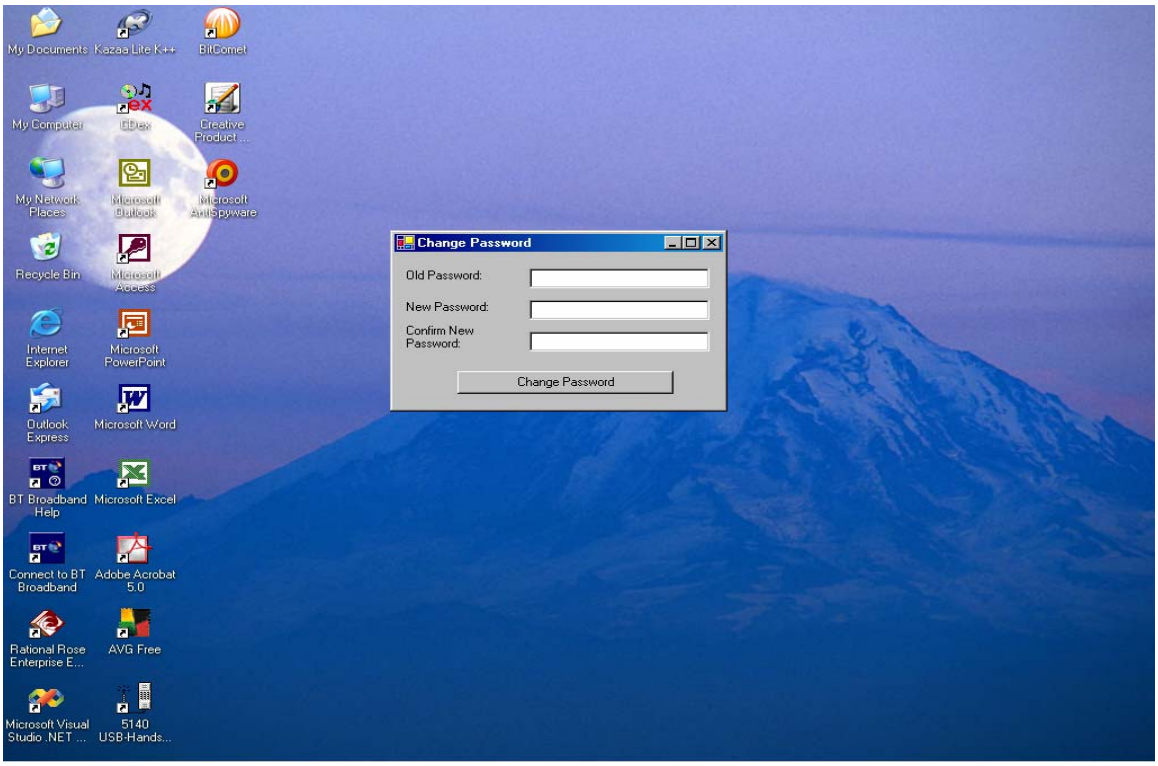
Enter the username and password and click the **Login** button to login.

The user then sees the Main Menu screen of the FAMS Application (see below).



8.8.2 Changing Password

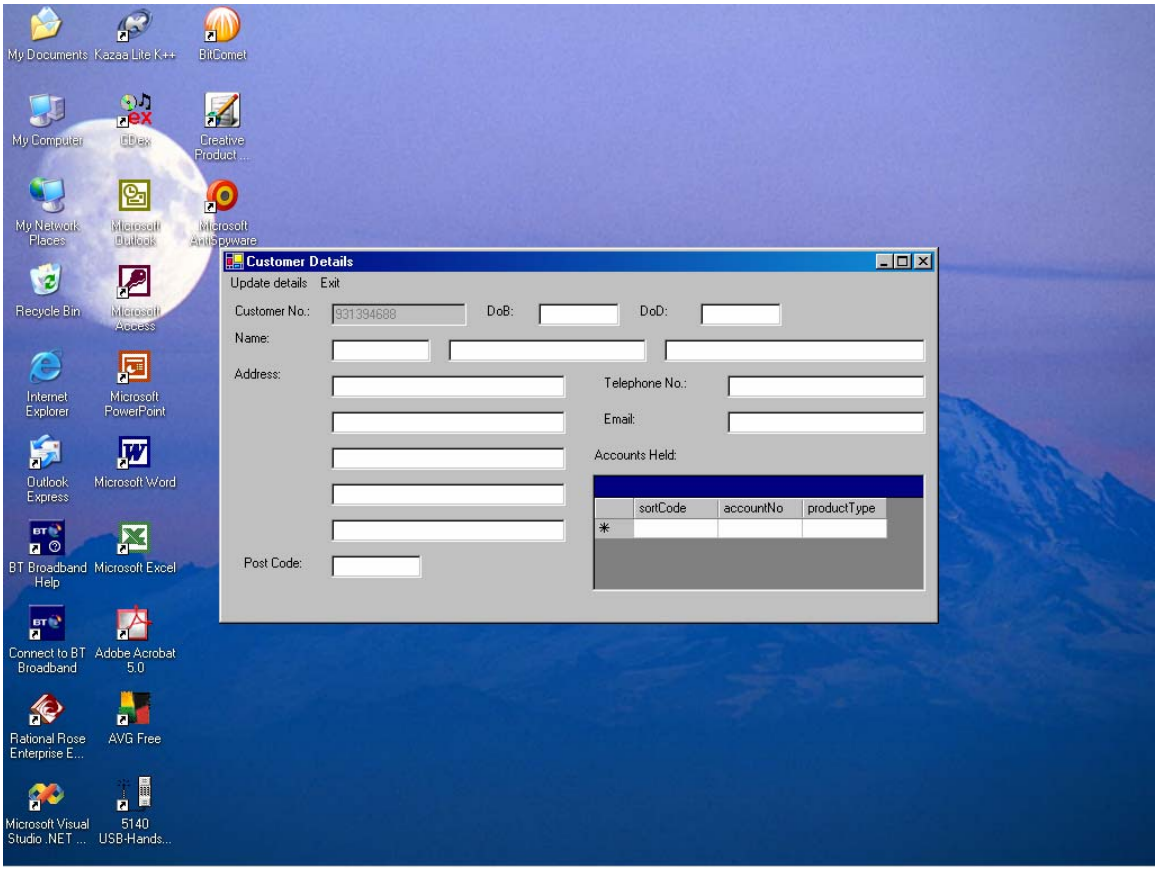
In the Main Menu screen click “Change password” at the top of the window. The following window opens:



Simply enter the old password and the new password (twice) and click “Change Password”.

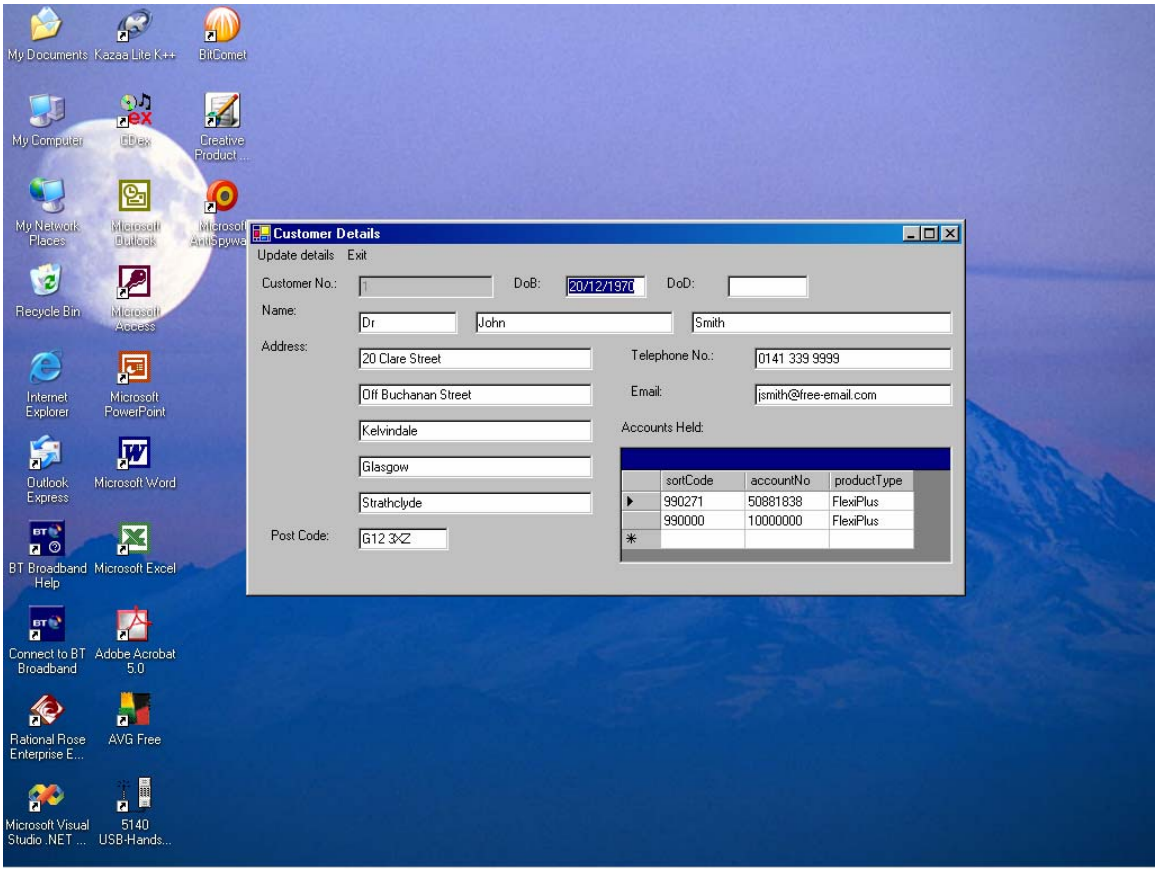
8.8.3 Creating a Customer and Editing/Viewing their details

To create a customer, click **Create Customer** in the Main Menu screen. This will randomly generate a customer number and display it in the **Customer No.:** field of the Main Menu. Then, click **View Customer** to Open the **Customer Details** screen (for the new customer). See Below.



Now enter the customer's details and click **Update details** to save the changes to the form.

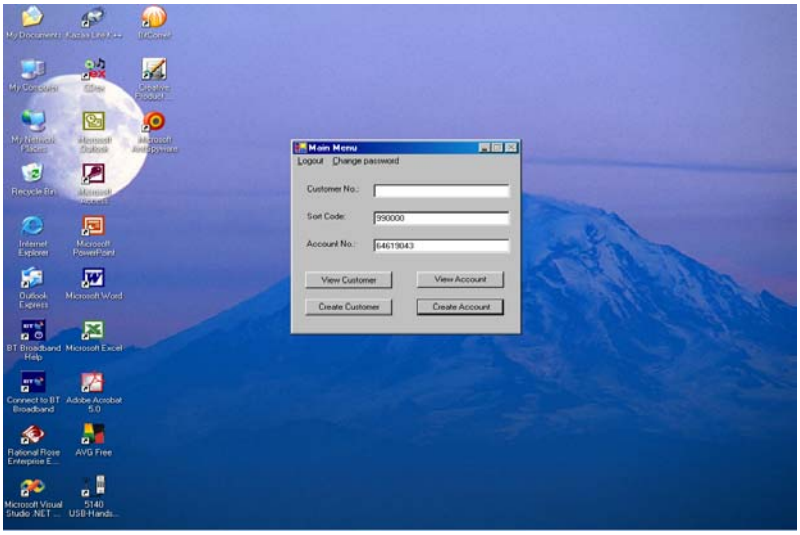
To view a customer's details simply enter their customer number in the **Customer No.:** field of the **Main Menu** and click **View Customer** (See Below).



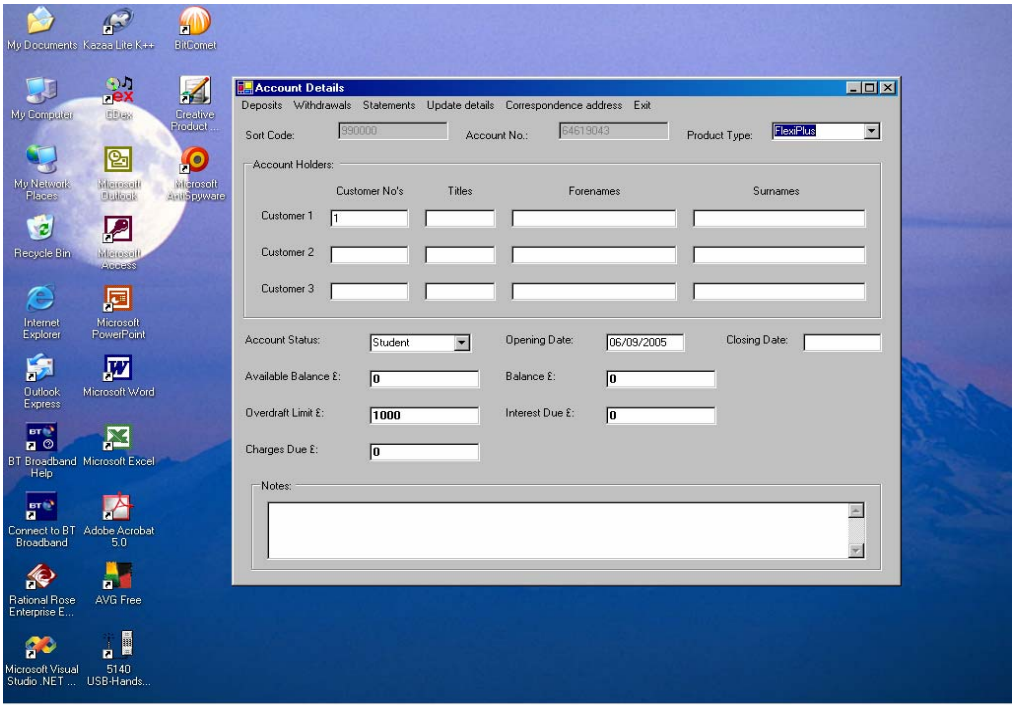
To edit the customer's details simply edit the form and click **Update details** - when the form is re-opened the changes will remain.

8.8.4 Creating an Account and Editing/Viewing the Account Details

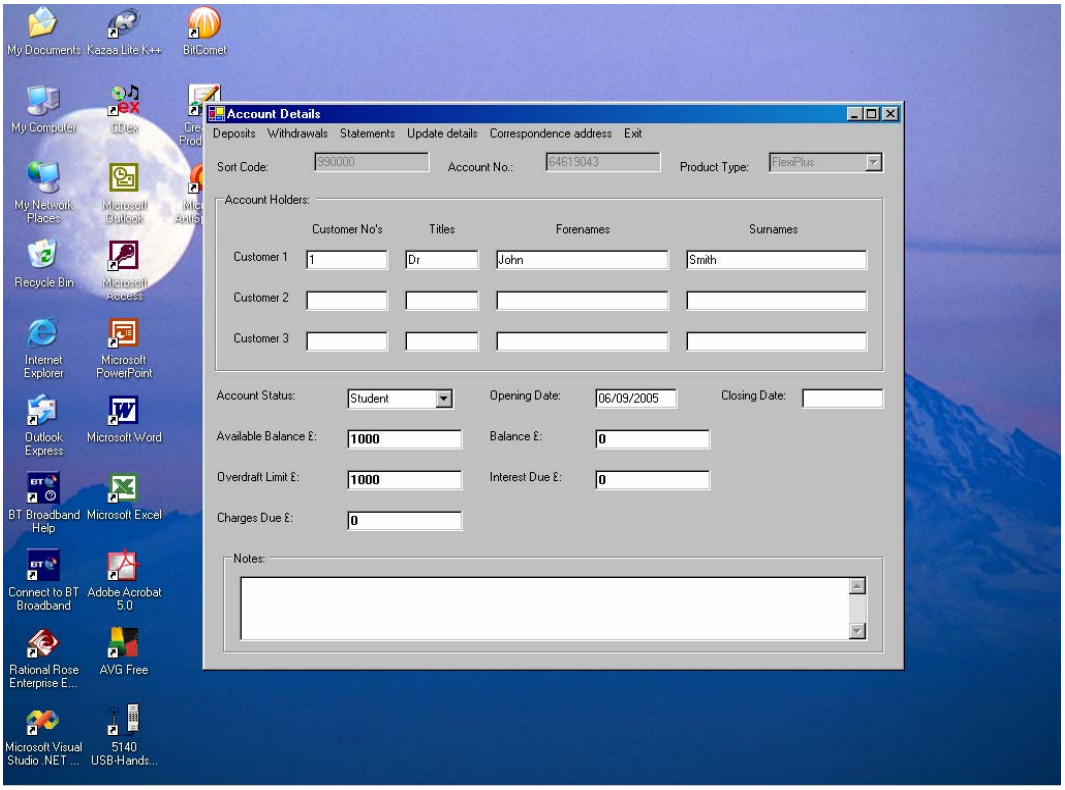
To create an Account, enter the appropriate sort code for the branch the customer is opening the account with (this will be in the range 990000 to 990271 as BestBank has 272 branches in the UK). Then, click **Create Account** in form **Main Menu**. This will randomly generate an Account Number for the given sort code and display it in the **Account No.:** field of the **Main Menu**. See Below.



Then, click **View Account** in the **Main Menu** to view and edit the account details (such as the overdraft limit). See below.



To add a signatory to the account, enter their customer number in the appropriate Customer No field and click **Update details**. When the form is re-opened the customer's name will appear on the form. See below.

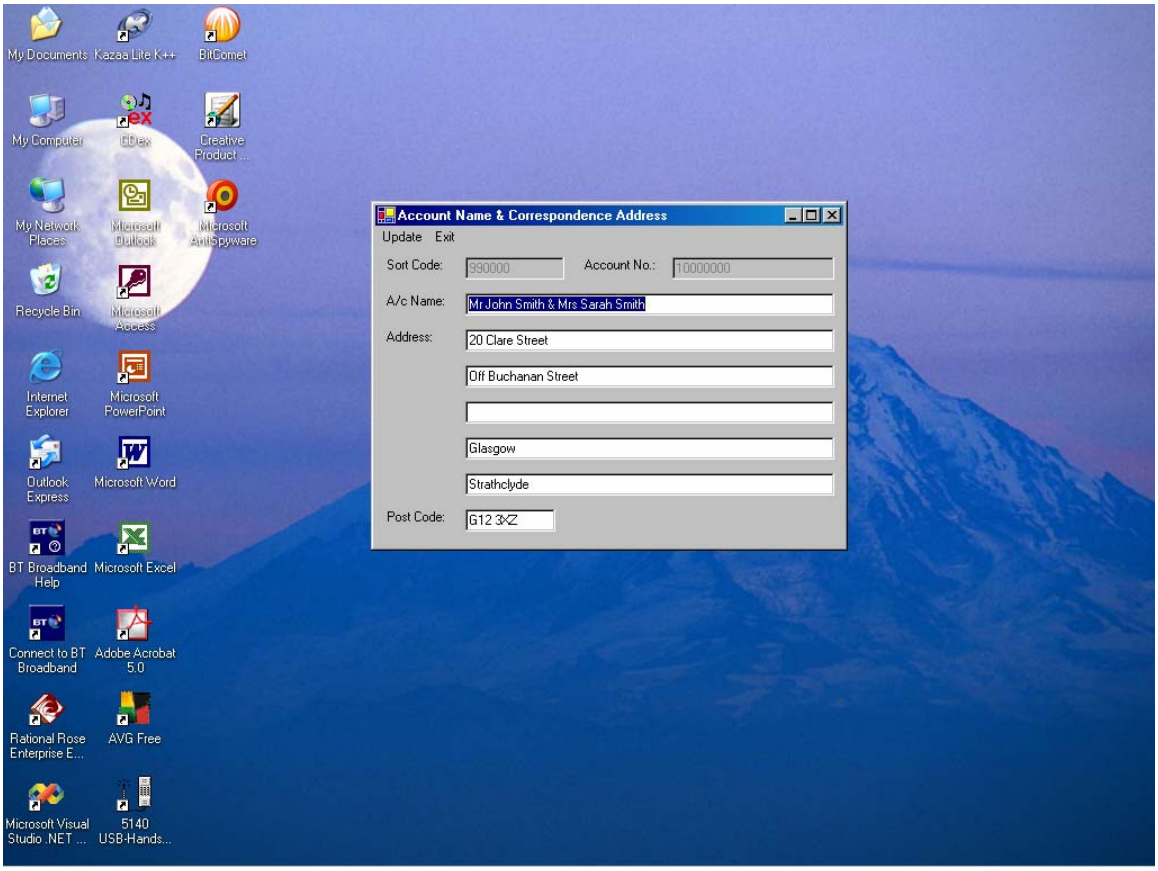


The following fields are updatable in the Account Details screen:

- Product Type (only when creating an account)
- Account Holders (by changing the value of the Customer No field and clicking **Update details**)
- Account Status (by Selecting the status from the drop down menu and clicking **Update details**)
- The account can be closed by selecting status **Closed** and entering a **Closing Date** then clicking **Update details.** Note that an account can only be closed if it has a zero balance.
- Notes may be entered in the **Notes** field – this would include information such as "Customer 3 has Power of Attorney over Customer 2", "Customer 3 is child of Customer 1 and Customer 2", etc. Once the Notes have been edited, click **Update details** to save the changes.

All changes made to forms will only appear on all the forms in the application when the forms are re-opened.

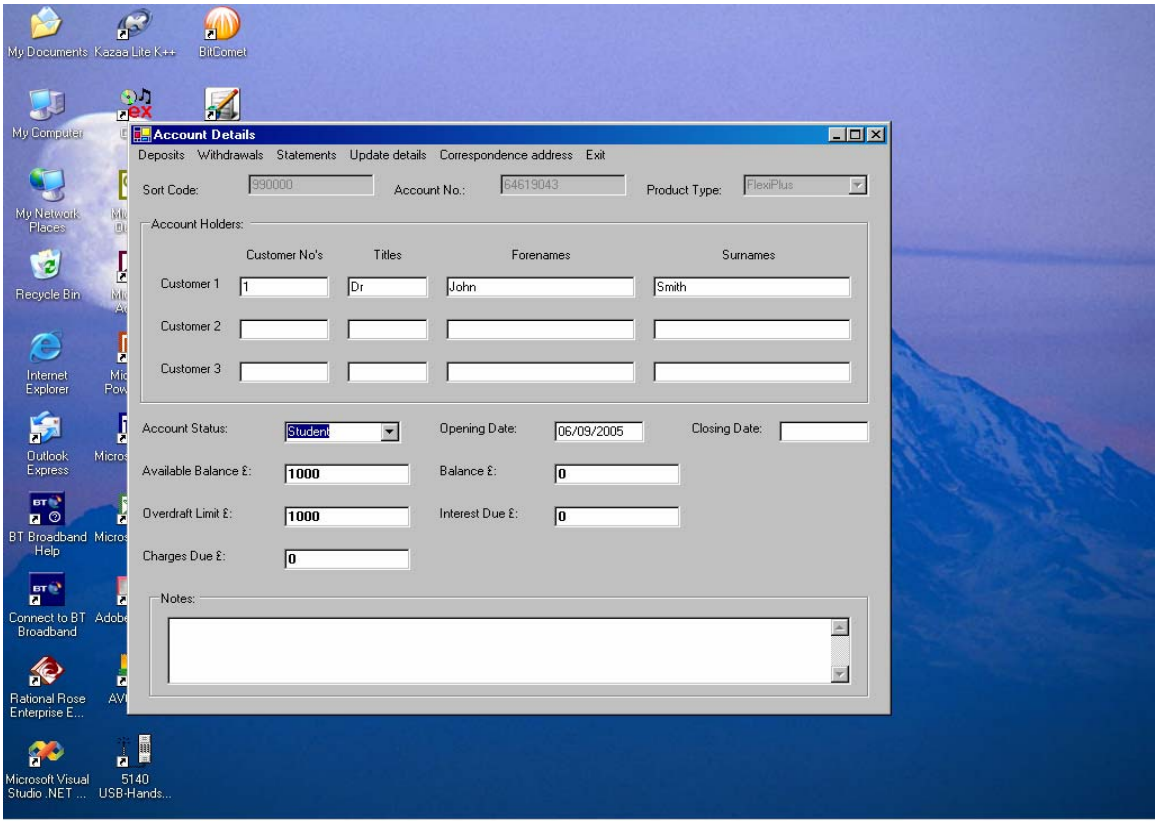
To enter the Account Name and Correspondence Address click **Correspondence address** in the **Account Details** screen. The following screen is opened:



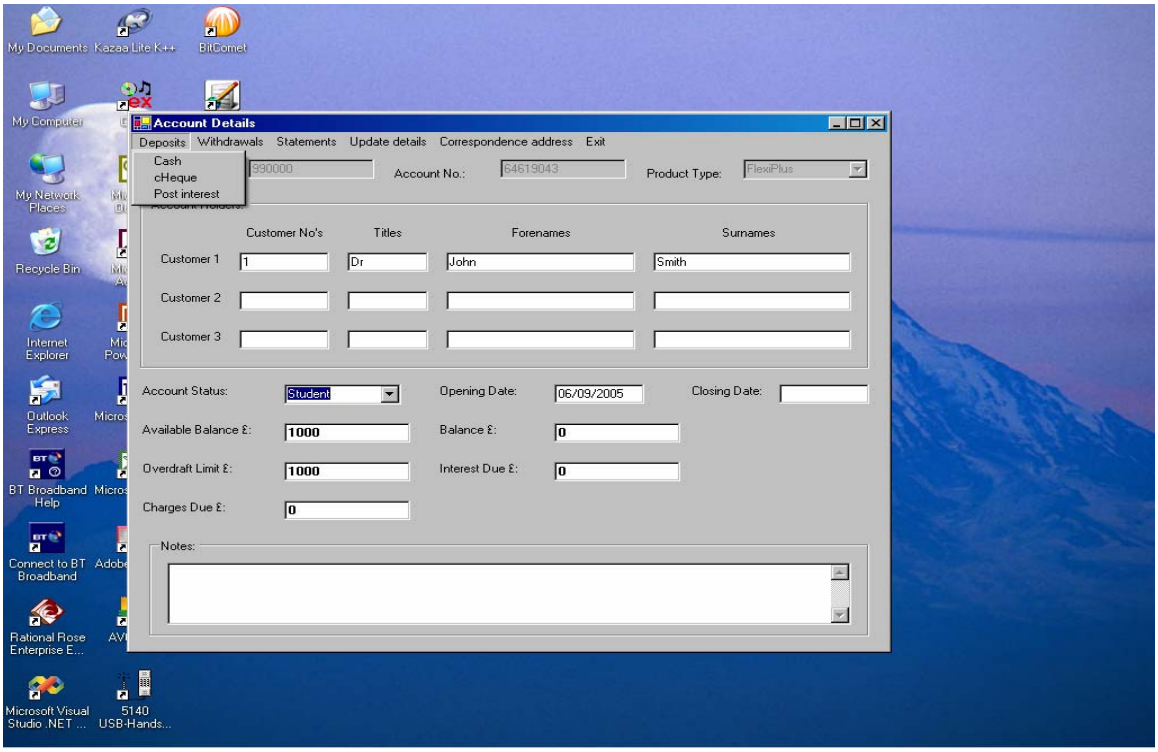
Simply enter the details and click **Update**. The changes will be saved and will appear on the form when it is re-opened.

8.8.5 Making a Cash Deposit.

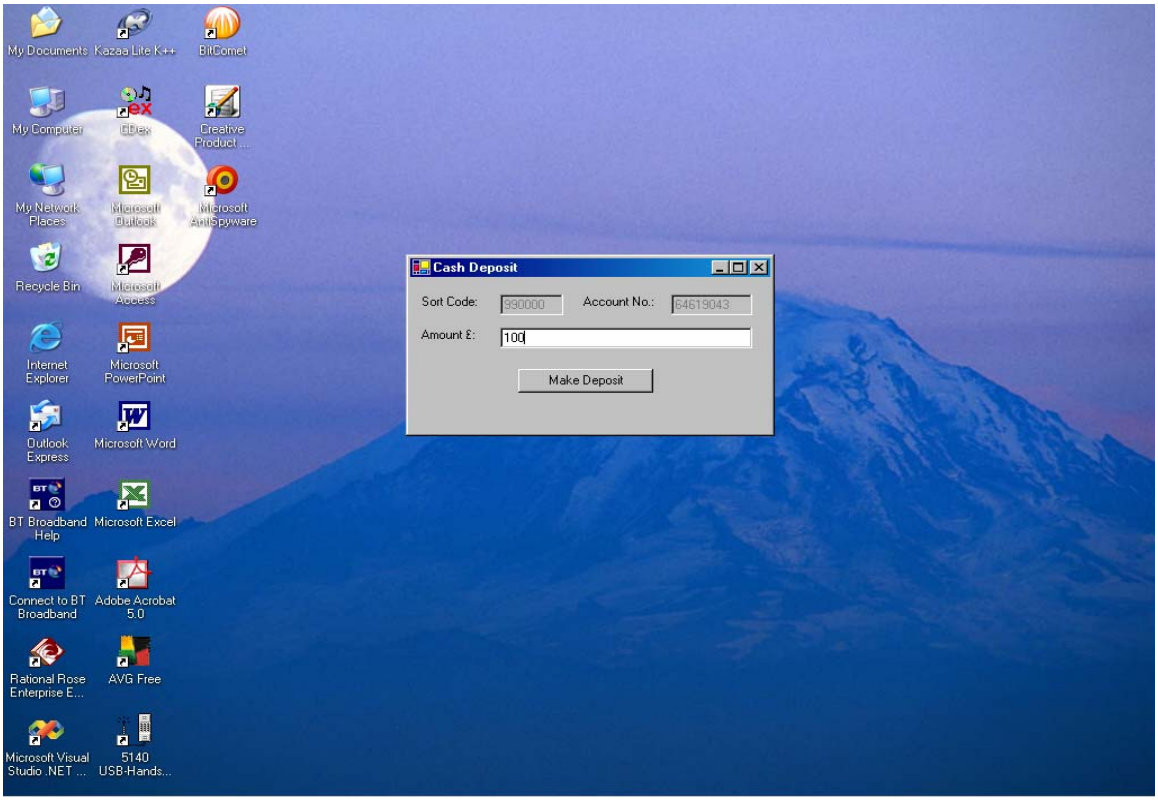
In the **Main Menu** screen enter the **Sort Code:** and **Account No.:** of the account that the sum of money has to be deposited into. Then click **View Account**. The following screen will appear:



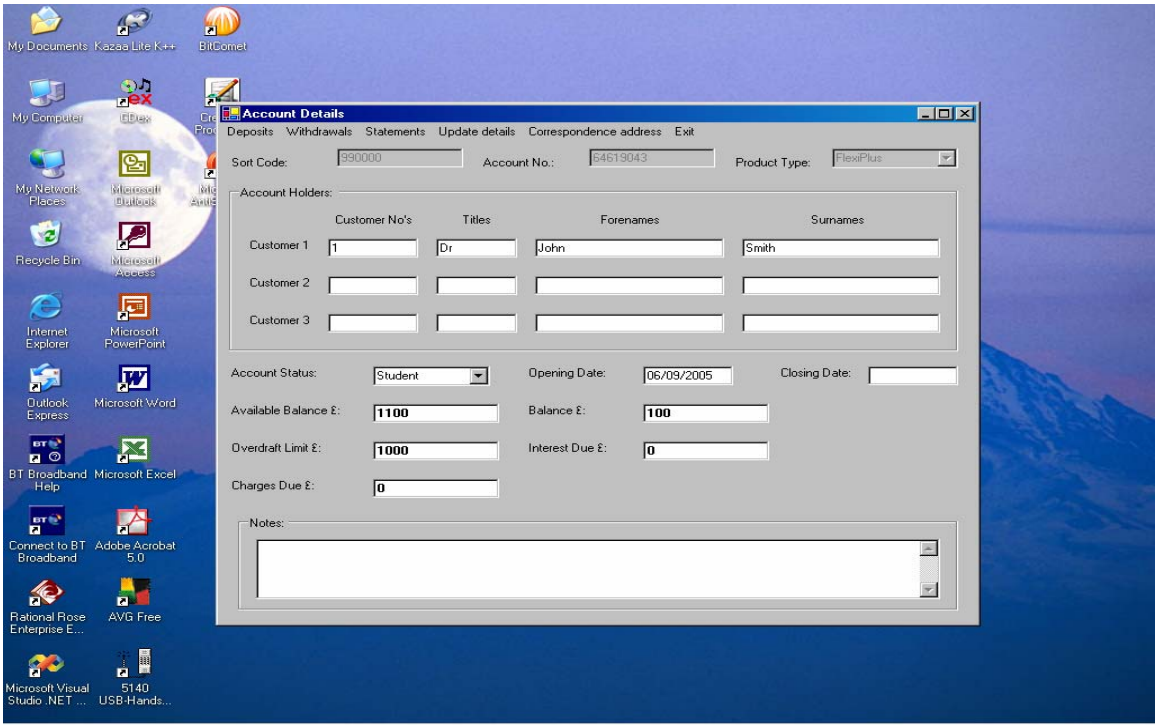
Then click on Deposits and then click on Cash, as shown below:



The following screen opens:



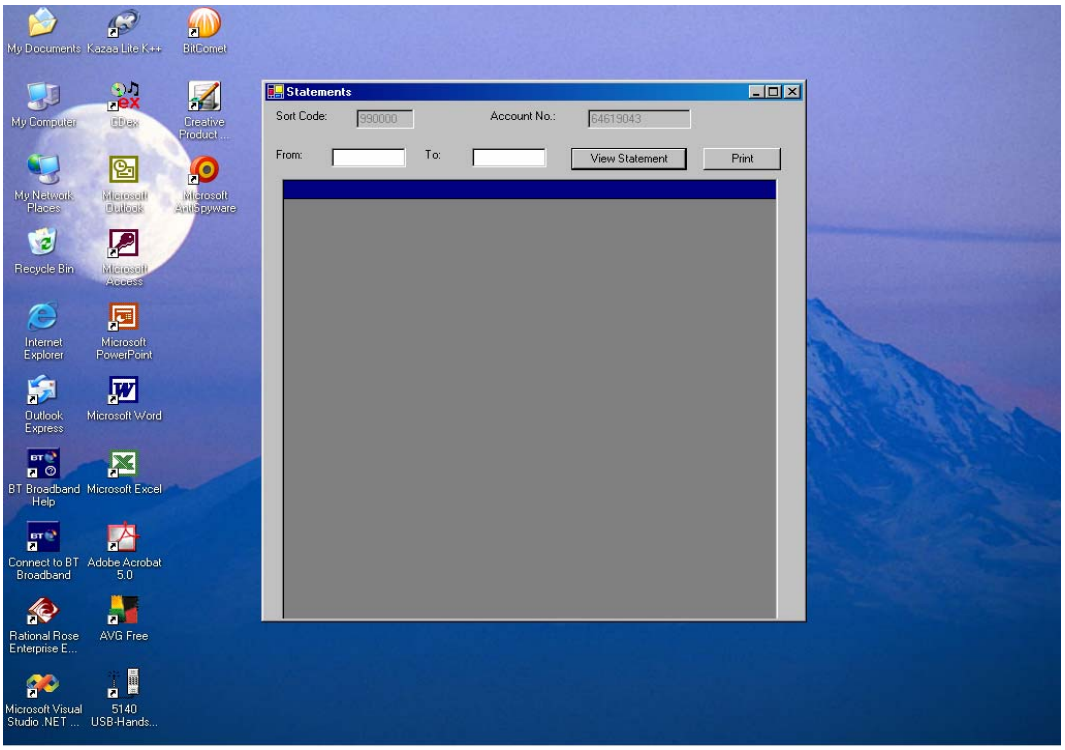
Simply enter the sum to be deposited and click **Make Deposit.** When the **Account Details** screen is re-opened the balance will be altered accordingly. See Below.



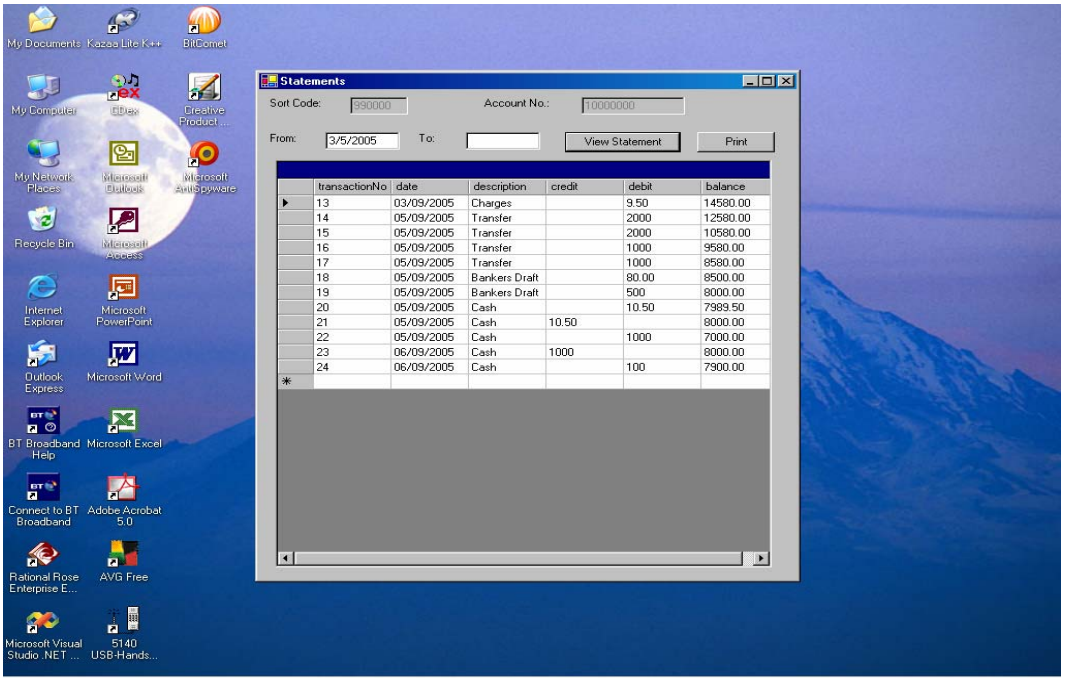
The transaction can be seen in the account's statements in the following way:

8.8.6 Viewing Statements

Open the appropriate Account Details screen and click Statements. The following screen will open:



If the user wishes to view all transactions they simply need to click **View Statement.** The user can also specify a start date and a finish date if they wish to view a subset of all the transactions. See below.



Chapter 9

Testing Strategy for the FAMS Prototype

This chapter is adapted from chapter 4 of [30]; for full coverage of test-case design, see Chapter 4 – Test-Case Design of [30].

It is impossible to completely test a system. Therefore, we must design good test-cases which, although they cannot guarantee the absence of all errors, seem likely to find errors if they do indeed exist. We need to find the subset of all possible test-cases which has the highest probability of detecting the most errors [30].

Selecting a set of test cases randomly will be unlikely to yield a set of test-cases which will have a high probability of finding the most errors. We need to develop a train of thought which will allow us to select a set of test data that is more likely to find errors than if we simply pick the test data at random.

Table 9.1 below lists some techniques for choosing this test data. Some are black-box or functional testing techniques; others are white-box or structural techniques. In [30] Myers says "*The recommended procedure is to develop test cases using the black-box methods and then develop supplementary test cases as necessary by using the white-box methods."* We briefly discuss these techniques in this chapter and outline a strategy for choosing test data.

Black Box	**White Box**
Equivalence partitioning	Statement coverage
Boundary-value analysis	Decision coverage
Cause-effect graphing	Condition coverage
Error guessing	Decision/condition coverage
	Multiple-condition coverage

Table 9.1 (taken from [30])

9.1 Equivalence Partitioning
A well selected test case should:

1) have a reasonable probability of finding an error
2) reduce by more than a count of one the number of other test cases that must be developed to achieve some predefined goal of "reasonable" testing; and
3) tell us something about the presence or absence of errors over and above the specific set of input values [30].

The last of these conditions suggests that we should partition the input domain into *equivalence classes* so that one can assume that using one value from the class to test the program will tell us as much as using any other value from the same equivalence class, i.e. if one test-case in an equivalence class finds an error, all the other test-cases in the same equivalence class would also find the same error. Conversely, if a test case from an equivalence class does not find an error, we would expect that no other test-case in the class would find an error [30].

In [30], Myers states that "*The equivalence classes are identified by taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups….Notice that two types of equivalence classes are identified: valid equivalence classes represent valid inputs to the program, and invalid equivalence classes represent all other possible states of the condition (i.e. erroneous input values)."*

The table shown in Table 9.2 below can be used when identifying equivalence classes (taken from [30]).

External Condition	Valid Equivalence Class	Invalid Equivalence Class

Table 9.2

In [30], Myers says the second step is to use the equivalence classes to identify test-cases. He says "*The process is:*

1. *Assign a unique number to each equivalence class.*
2. *Until all valid equivalence classes have been covered by (incorporated into) test-cases, write a new test-case covering as many of the uncovered valid equivalence classes as possible.*
3. *Until all invalid equivalence classes have been covered by test-cases, write a test-case that covers one, and only one, of the uncovered invalid equivalence classes."*

9.2 Boundary-Value Analysis

It has been shown that test-cases that investigate boundary conditions usually have greater success in identifying errors than test-cases that do not investigate boundary conditions. Boundary conditions are found around the edges of equivalence classes, i.e. directly on, or directly under or directly above the edges of input and output equivalence classes.

In [30] Myers says that *"Boundary-value analysis differs from equivalence partitioning in two respects*

1. *Rather than selecting any element in an equivalence class as being representative, boundary-value analysis requires that one or more elements be selected such that each edge of the equivalence class is the subject of a test.*
2. *Rather than just focusing attention on the input conditions (input space), test cases are also derived by considering the result space (i.e. output equivalence classes).*

There is no specific algorithm for boundary-value analysis but some general guidelines are to be found on page 51 of [30].

9.3 Cause-Effect Graphing

Equivalence partitioning and boundary-value analysis are limited in the sense that they do not consider combinations of input into a program. For example, a program that grades student's exam papers may work well when the maximum number of students is entered and, again, may work well when the maximum scores are entered but may fail when both the maximum number of students *and* the maximum scores are entered simultaneously, i.e. when the input is combined the program fails, perhaps due to lack of storage [30].

Testing combinations of input is usually an extremely complex task due to the vast number of possible groupings. One needs a systematic way of choosing input conditions. Cause-effect graphing is a method of choosing suitable test-cases which have a high probability of finding errors.

A cause-effect graph is in actual fact a combinatorial logic network (normally used to model digital electronics circuits) with a simplified notation. One does not need knowledge of electronics to construct a cause-effect graph; all that is required is a knowledge of Boolean logic (AND, OR, NOT, etc. operators) [30].

Test-cases can be derived using cause-effect graphing in the following way:

1. Divide the specification into manageable pieces – cause-effect graphing quickly becomes unmanageable when used on large specifications.
2. Identify the causes and effects in the specification. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a change in the state of the system, e.g. a file being updated. Causes and effects are identified by reading the system specification and picking out words or phrases that describe causes and effects [30].
3. The semantic content of the specification is used to create a Boolean graph which links the causes and effects – this is the cause-effect graph.
4. Constraints are added to the graph that highlight causes and/or effects that are impossible due to the syntax of the cause-effect graph language or the environment.
5. The graph is then converted into a limited-entry decision table, where each column in the table is a test-case, by tracing state conditions in the graph.
6. The columns in the limited-entry decision table are converted into test-cases [30].

For full details of how to construct and work with cause-effect graphs, see page 56 of [30].

9.4 Error Guessing

There is no particular recipe for error guessing. It is done in an intuitive and ad hoc manner that mainly comes down to experience. Testers who use error guessing identify probable errors using a combination of intuition and experience and then design test-cases to expose these errors.

For several examples of situations where error guessing has been used, see page 73 of [30].

9.5 Logic-Coverage Testing (White-Box Testing)

In [30], Myers says "*White-box testing is concerned with the degree to which test cases exercise or cover the logic (source code) of the program…..the ultimate white-box test is the execution of every path in the program, but since in a program with loops the execution of every path is usually infeasible, compete path testing is not considered here as a viable testing goal."* A program could contain an infinite number, or near infinite number, of possible paths and so it would be impossible to test them all.

The execution of every statement in a program (statement coverage) may seem like a good criterion but, while being necessary, it is actually a very poor testing criterion. In actual fact, statement coverage is so weak it is usually considered to be useless.

Decision coverage (branch coverage) is a stronger criterion than statement coverage. In order to fulfil decision coverage the tester must write enough test-cases so that each decision in the program has at least one true outcome and at least one false outcome, i.e. each branch direction of the source code must be traversed at least once. Decision coverage usually satisfies statement coverage, although there are exceptions to this, e.g. if a program has several entry points a particular statement may only be executed if the program is entered at a particular entry point. Since statement coverage is necessary, decision coverage requires that each decision has at least one true outcome and at least one false outcome *and* that every statement in a program be executed at least once. Decision coverage, although stronger than statement coverage, is still a fairly weak testing criterion.

Condition coverage is a stronger testing criterion than decision coverage. In order to fulfil condition coverage the tester writes enough test cases such that each condition in a decision takes on all possible outcomes at least once *and* each point of entry is invoked at least once (so that every statement in a program is executed at least once).

Condition coverage does not always satisfy decision coverage. Hence, we have decision/condition coverage which requires sufficient test-cases to be written such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once and each point of entry is invoked at least once. A weakness with decision/condition coverage is that it does not always exercise the outcomes of all conditions in a program. A testing criterion that covers this problem is multiple-condition coverage.

Multiple-condition coverage requires the tester to write enough test-cases to invoke all possible combinations of conditions at least once and all points of entry at least once. Multiple-condition coverage satisfies the decision-coverage, condition-coverage and decision/condition coverage testing criterion.

9.6 A Testing Strategy

The testing techniques discussed so far should be combined into a testing strategy, as no one technique on its own yields a complete set of test-cases.

In [30], Myers outlines the following strategy:

1. Start with cause-effect graphing, if the system specification contains combinations of input conditions.
2. Always use boundary-value analysis.
3. If necessary, supplement the test-cases identified so far by identifying valid and invalid equivalence classes and writing test-cases from them.
4. Use error guessing to add to the test-cases already identified.
5. Add test-cases to satisfy the white-box testing criterion outlined in section 9.5 above.

Chapter 10

Evaluation

10.1 UCMs and Requirements Gathering

As this project has been conducted by case study, where there would normally be the requirements gathering phase of the project, the case study was designed and written.

As little was initially known about banking systems, considerable time was spent gathering information about them. One of the major problems in this phase of the project was not finding answers to questions but knowing what questions to ask in the first place. The UCM shown in Figure 4.2 was of immense help in this. Initially, it was imagined that the FAMS application would need to implement BACS, CHAPS, ATM Transactions, etc. This was not the case. However, it was only when the UCM in Figure 4.2 was constructed that the separation of software entities needed in order to provide all the services required for the system was realised. That is, it was only when the UCM in Figure 4.2 was constructed that it was realised that a multi-agent system was required in order to provide all the services needed by a bank accounts system. In this sense, UCMs proved to be invaluable.

As UCMs model scenarios (use cases) and scenarios are closely related to requirements, UCMs are easily understood by different stakeholders in a project [3]. This means that everyone involved in a project can talk the same language, which is a great benefit. Also, it was found that UCMs are easily explained to the uninitiated, which is extremely helpful.

In [3] Amyot and Eberlein say "*…UCMs…support component-independent scenarios, which are useful for early descriptions of requirements and help avoiding early commitments to a specific architecture."* In other words, one may construct a use case path without binding responsibilities to specific components which is very useful when initially describing system requirements. This was found to be the case here. The greatest benefit derived from constructing Figure 4.2 was that it facilitated the identification of the software agents needed to carry out the system's responsibilities. This was because one was able to speculate about what software agents were required rather than having to immediately allocate responsibilities to specific entities (software agents).

10.2 UCMs vs. UCDs

In [3], Amyot and Eberlein highlight the most important differences between UCMs and UCDs as follows:

- UCDs are component-independent whereas UCMs can be either Component-independent or component-centred. That is to say, UCDs describe scenarios (use cases) in a purely functional style, independent from components, whereas UCMs can describe scenarios in a purely functional style or in terms of communication events between system components.
- UCDs are represented by a graph, i.e. the scenarios are described by a graph, whereas UCMs are represented by paths (use case paths) on two-dimensional components.
- UCDs have no ordering whereas UCMs have causal ordering, i.e. scenarios represent a collection of events which can be ordered either sequentially or causally – UCDs show no ordering of events whereas UCMs show causal ordering of events (pre-conditions and triggering events causing other post-conditions and resulting events (responses) to take place in the system).
- UCDs do not have any mechanism for showing time constraints whereas UCMs have some support for showing time constraints (timers and response time requirements).
- UCDs use dependencies to decompose scenarios into smaller re-usable pieces whereas UCMs takes a hierarchical approach to decomposition, using static stubs, dynamic stubs and pools.

- UCDs are static in that they do not enable the description of system behaviour that modifies itself at run-time whereas UCMs are dynamic in that they use dynamic stubs, pools and dynamic responsibilities to show system behaviour that modifies itself at run-time.

However, UCDs are grey-box in that they use dependencies to show internal information about the system, e.g. an extends dependency (relationship) between two use cases would indicate that one use case represented optional system behaviour and therefore we would expect a decision (e.g. an If/Then statement) in the program code. UCMs are also grey-box but show this kind of information with OR-forks and OR-joins [2] [3].

Both UCMs and UCDs support abstract scenarios rather than concrete scenarios. Abstract scenarios are generic, with formal parameters, whereas concrete scenarios focus on one specific instance, with concrete data values [3].

Also, both UCMs and UCDs can focus on several actors in a system at once [3].

10.3 UCMs vs. Sequence Diagrams
Sequence diagrams can describe scenarios (use cases).

In [3], Amyot and Eberlein highlight the most important differences between UCMs and sequence diagrams as follows:

- Sequence diagrams are component-centred whereas UCMs can be either component-centred or component-independent.
- Sequence diagrams are represented as sequence diagrams (patterns of interactions among objects) whereas UCMs are represented as paths (use case paths) on two-dimensional components.
- Sequence diagrams show sequential ordering of events whereas UCMs show causal ordering of events.
- Sequence diagrams have no provision for showing time constraints whereas UCMs do have some provision for showing time constraints.
- Sequence diagrams have no provision for decomposing scenarios into smaller re-usable pieces whereas UCMs can (hierarchically) decompose scenarios into smaller re-usable pieces.
- Sequence diagrams can represent both abstract and concrete scenarios whereas UCMs only represent abstract scenarios, i.e. sequence diagrams can represent generic scenarios that have formal parameters or a specific instance of a scenario that has concrete data values whereas UCMs only represent generic scenarios that have formal parameters.
- Sequence diagrams are static in that they do not enable the description of system behaviour that modifies itself at run-time whereas UCMs are dynamic in that they use dynamic stubs, pools and dynamic responsibilities to show system behaviour that modifies itself at run-time.

In [3], the authors say *"…UML Sequence Diagrams are very useful for single scenarios, especially when describing lengthy black-box interactions between actors and a given system (something that….UCMs do not do well)".* However, sequence diagrams model components in a one-dimensional manner, i.e. they do not allow decomposition of components (components cannot contain sub-components). UCMs show two-dimensional components (components can contain sub-components).

Also, sequence diagrams are grey-box in that they show patterns of interactions among objects; these interactions can represent method calls in the program's source code. UCMs are also grey-box but use responsibilities to model method calls in the program's source code.

Lastly, both UCMs and sequence diagrams can focus on several actors in a system at once [3].

10.4 UCMs vs. Statechart Diagrams

Statechart diagrams model scenarios by focusing on component behaviour (the behaviour of objects in response to messages they receive throughout the execution of the scenarios).

In [3], Amyot and Eberlein highlight the most important differences between UCMs and statechart diagrams as follows:

- Statechart diagrams are component-centred whereas UCMs can be either component-centred or component-independent.
- Statechart diagrams are black-box, i.e. they describe system behaviour with respect to their environment only, whereas UCMs are grey-box, i.e. they include internal information about the systems behaviour.
- Statechart diagrams are represented by a state machine whereas UCMs are represented as paths (use case paths) on 2-dimensional components.
- Statechart diagrams have no provision for showing time constraints whereas UCMs do have some provision for showing time constraints.
- Statechart diagrams are static in that they do not enable the description of system behaviour that modifies itself at run-time whereas UCMs are dynamic in that they use dynamic stubs, pools and dynamic responsibilities to show system behaviour that modifies itself at run-time.

However, both UCMs and statechart diagrams show causal ordering of events [3].

Also, statechart diagrams allow scenarios to be decomposed into smaller re-usable pieces by using substates (nested states). UCMs also provide for decomposition with the use of static stubs, dynamic stubs and pools.

Both UCMs and statechart diagrams support abstract scenarios rather than concrete scenarios [3].

Also, both UCMs and statechart diagrams can focus on several actors in a system at once [3].

10.5 Bridging the gap between Requirements and Use Cases and more detailed views

In [2], Amyot says "*Use Case Maps allow for the early evaluation of architectural alternatives by acting as a link between function (use cases) and form (structure.)"* and in [4] Amyot and Mussbacher say "*By combining behaviour and structure in one view in an explicit and visual way, UCM enable the understanding of scenario paths as well as high-level architectural reasoning."*

Use case paths model the function of a system while the underlying component substrates model the structure. It is considered here that as interaction diagrams and statechart diagrams rely on knowledge of the underlying structure of the system, UCMs successfully bridge the gap between requirements and use cases (modelled by use case paths and responsibilities) and more detailed views, such as sequence diagrams, which rely on knowledge of the structure of the system (modelled by the component substrates and responsibilities). This is illustrated by the way in which the UCMs of chapter 7 link the scenarios of section 6.2 to the classes in the class diagram (Figure 6.2) in section 6.3.

10.6 Extending UML with UCMs

Firstly, let us consider the possibility of substituting UCDs with UCMs. Although one can use static stubs in UCMs to represent the includes relationship of a UCD, OR-forks/OR-joins or dynamic stubs in UCMs to represent the extends relationship of a UCD and OR-forks/OR-joins or multiple dynamic stubs to represent the generalization relationship of a UCD [2], it is considered here to be the case that UCMs are a poor substitute for UCDs as they do not represent the functional requirements of a system in as succinct a way as UCDs do. Although one can construct an unbound UCM (one that is not bound to a component substrate), it is not considered here to be as informative as a UCD, particularly to the uninitiated. Alternatively, if one constructs a bound UCM (one which *is* bound to a component substrate) the resultant diagram contains far more information about the internal workings of the system than a UCD would show.

The class diagram is at the heart of object-oriented analysis and design. There are four types of component in the UCMs of chapter 7; namely, the FAMS application, classes that form the FAMS application, the FAMSDB database and the tables that form the FAMSDB database. When constructing these UCMs it was found that the process of binding responsibilities to components was invaluable for identifying classes which could then be added to the class diagram. Thus, it is considered here that one should initially construct the UCMs before attempting to construct the class diagram; once both the UCMs *and* the class diagram have been constructed, one can update them both in an opportunistic fashion as the development of the system proceeds.

UML object diagrams were not used in this project. However, object diagrams are generally used to confirm the necessity of relationships between classes in the class diagram and are also used to establish the multiplicities of relationships between classes in the class diagram. UCMs have a completely different role. Hence, it is considered here that UCMs cannot replace object diagrams.

UCMs cannot replace sequence diagrams as sequence diagrams explicitly show message exchanges between components whereas UCMs have a different emphasis. However, in [2] Amyot says *"Use Case Maps can help deriving interaction diagrams (in the analysis and design models) from use cases (in the use case model). UCMs do not explicitly define message exchanges between components, but messages need to be constructed in such a way that the causal relationships between responsibilities from different components are satisfied."* This would indicate that UCMs should be used as an aid to the construction of interaction diagrams. In this project, it can be seen that the UCMs of chapter 7 could certainly act as an aid to constructing the sequence diagrams of section 6.5. Hence, it would be advisable to construct the UCMs before attempting to construct the interaction diagrams for a system. (For full coverage of how to construct interaction diagrams from UCMs, see [2]).

UML collaboration diagrams were not used in this project. However, as collaboration diagrams model the same information as sequence diagrams (with the emphasis on the organizational structure of the objects that pass messages rather than on the time ordering of the messages, as in sequence diagrams), it is considered here that UCMs cannot replace them.

In [2] Amyot says "*UCMs do not replace these* [statechart] *diagrams, but they can guide in their construction."* The statechart diagrams in this project were constructed mainly from the use case descriptions of section 6.2. However, it can again be seen that the UCMs of chapter 7 could certainly act as an aid to constructing the statechart diagrams of section 6.6. Hence, it would be advisable to construct the UCMs before attempting to construct the statechart diagrams for a system. (For full coverage of how to construct statechart diagrams from UCMs, see [2]).

UML activity diagrams were not used in this project. However, activity graph metaclasses support many, but not all, UCM concepts [5]. Thus, because of their similarity, one might suppose that UCMs could replace activity diagrams. However, UML activity diagrams are usually used for business-oriented modelling such as workflows or for modelling the details of a computation (an operation) whereas UCMs have a different role. In [5], Amyot and Mussbacher say *"activity diagrams focus on internal processing, often found in business-oriented models (e.g. workflows), whereas UCMs are also concerned with external (asynchronous) events…"* Hence, it is considered here that it would be inappropriate to attempt to replace activity diagrams with UCMs.

Component diagrams focus on modelling the physical aspects of a system such as modelling source code, executable releases, physical databases, etc. whereas UCMs have a different role. Thus, it is considered here that it would be inappropriate to attempt to replace component diagrams with UCMs.

Deployment diagrams show a set of run-time processing nodes and, possibly, the components housed on them. UCMs have a different emphasis. Therefore, again it is considered here that it would be inappropriate to attempt to replace deployment diagrams with UCMs.

Thus, it is considered here to be the case that UCMs cannot replace any of the nine UML diagrams but adding UCM views of a system to existing UML views can lead to a far greater understanding of the system under investigation and can greatly improve the systems analysis and design, by contributing to the construction of the class diagram, sequence diagrams and statechart diagrams, as well as providing another view of the system.

10.7 Project Review

Firstly, the project could have been more concise. However, once it was written there was insufficient time to re-visit each chapter and reduce it to the salient points.

Chapter 2 – Use Case Maps and the Unified Modeling Language included explanations of the various symbols in the UCM notation as well as explanations of the various symbols in the UML. Examples were given of the symbols in use. However, it may have been better if concrete examples were used rather than abstract ones. Again, there was insufficient time to re-visit Chapter 2.

It is believed by the author that Chapter 3 – The Case Study is as realistic as it could be bearing in mind that the project was not carried out in conjunction with industry. HSBC Bank, Abbey National Bank and LogicaCMG (IT consultancy) were written to and asked to comment on the realism of the case study. Their responses are in Appendix 9 – Case Study Validation Letters. The case study was altered in light of their responses but there will undoubtedly be points that the reader feels should have been included in the case study and, conversely, there will be points that have been included in the case study which the reader will feel should have been treated differently.

The mechanism by which bill payments are made was omitted for the UCM of Figure 4.2. It would have been better if some attempt had been made to model this.

There was insufficient time in which to test the implementation, i.e. the FAMS application and the FAMSDB database that were built with VB .NET and Jet 4.0 and Access 2002 were not rigorously tested. Given more time they would have been thoroughly tested. One problem that the author is aware of is that once the FAMS (with FAMSDB) application is installed, using the FAMS Installer, the user is unable to view new records added to the database via the FAMS application by opening the FAMSDB database (FAMSDB.mdb). There was insufficient time to investigate this and make any necessary corrections.

Given more time the FAMS application would have included data validation and defensive programming. As was said in Chapter 8 – Implementation, this has been omitted in order to simplify the implementation.

The FAMS application did not make any provision for data gathered by card-reading devices as this was considered to be beyond the scope of this project. It would have been better if this had been included.

Chapter 9 – Testing Strategy for the FAMS Prototype did not give any concrete examples of test-cases. It would have been better if some concrete examples were given.

Lastly, the presentation of the project would be done differently if the project were to be re-written, in that the writing style would be changed. This project was written in a business letter style, with a line space between paragraphs; if it were to be re-written, rather than using a line space to mark a new paragraph, the first line of the new paragraph would be indented.

Chapter 11

Conclusions and Recommendations

It is considered here that there is no doubt that adding UCM views to existing Unified Modeling Language views of a system, contributes a great deal to one's understanding of the system under investigation.

It is believed that this project has clearly shown that UCMs contribute significantly in two respects:

1. The train of thought invoked by constructing course grained UCMs leads to a far better understanding of the structure of the required system. This was shown here when Figure 4.2 identified the fact that a multi-agent system was required in order to provide all of the services required by a bank accounts application.
2. The (fine-grained) UCMs of chapter 7 clearly show that UCMs link Use Cases (scenarios) to classes (in the class diagram). Thus, they not only bridge the gap between requirements and use cases and more detailed views of the system, which rely upon knowledge of the underlying structure of the system, but they also act as an aid to the construction of the (UML) class diagram, sequence diagrams and statechart diagrams.

As stated in section 10.6, it is considered here to be the case that UCMs cannot replace any of the nine UML diagrams. Hence, the following analysis and design process, which incorporates UCMs *and* UML diagrams, is recommended here:

1. Construct a Deployment Diagram (to model the platform that the applications are to be deployed on)
2. Construct UCMs (course grained - for requirements gathering and mapping applications to hardware)
3. Construct Component Diagrams (for modelling the general physical structure of the system)
4. Construct UCDs (for modelling the functional aspects of the system/applications)
5. Write Use Case Descriptions (scenarios)
6. Construct UCMs (fine grained - for mapping scenarios to classes and aiding the construction of the class diagram, sequence diagrams and statechart diagrams)
7. Construct the Class Diagram
8. Construct Sequence Diagrams
9. Construct Statechart Diagrams

Note that Activity Diagrams have not been used in this paper. However, it is imagined that they would be placed between 1. and 2. above when modelling workflows as seen by actors in the system and would be placed at the end of the process when being used to model operations (the details of computations).

Also, there were no Object Diagrams in the analysis and design of the FAMS application. If one wished to include Object Diagrams (e.g. to verify the multiplicities of a relationship in the Class Diagram), they would be placed between 7. and 8. in the above process.

If Collaboration Diagrams were required, they would be placed between 8. and 9. in the above process.

11.1 The Way Forward

UCMs are restricted to functional requirements modelling and do not represent non-functional requirements in any way, except perhaps time constraints (see below). This is overcome in the *User Requirements Notation* (URN) where Goal-oriented Requirements Language (GRL) is used to represent non-functional requirements and UCMs are used to represent functional requirements.

Using GRL in conjunction with UCMs provides a complete representation of system requirements (both functional and non-functional).

In November 2000, ITU-T Study Group 17 started work towards the standardization of URN [36].

This paper has investigated the possibility of integrating UCMs and UML. It is hoped that in the future the possibility of integrating GRL with UML can also be considered; the aspiration being that the URN could be integrated with the UML to provide one standard language for object-oriented analysis and design.

As a side note, time constraints (response time requirements) were not included in the design of the FAMS application because they were considered to be a non-functional requirement which would be dealt with by *BestBank's* Group Technology Services Department. It is imagined that whither or not time constraints are a functional requirement or a non-functional requirement is a grey area. In this paper they have been treated as a non-functional requirement. However, in the URN, UCMs cover functional requirements and Goal-oriented Requirements Language cover non-functional requirements yet time constraints (a non-functional requirement in this paper) are included in the UCM notation. This is perhaps an issue that could be looked at in the future.

Appendix 1

Glossary

µs	Microseconds, eg. 1µs = 1×10^{-6} seconds
Abstract Class	(in an inheritance hierarchy) A super-class which will never be explicitly instantiated – only objects of its sub-classes can be instantiated.
Action	an atomic (i.e. non-interruptible) computation within a state machine which ultimately results in a change in state of the system or the return of a value.
Action State	A state whose purpose is to execute an action and then transition to another state.
Activity	ongoing nonatomic (i.e. interruptible) computations within state machines that ultimately result in some action.
Activity State	A state that represents the execution of a computation with substructure, typically the invocation of an operation or a statement within it or the performance of a real-world procedure.
Actor	Anything that interacts with the system, eg. a user, a company department, another system, etc.
Agent	An agent is a stand-alone computer program that performs tasks on behalf of a controlling entity, but without direct, continuous supervision.
API	Application Programming Interface
ATM(s)	Automated Teller Machine(s)
BACS	Banker's Automated Clearing Service: Voca Limited (formerly BACS Limited) – provides electronic payment services (with a 3 day payment cycle) to its members. Voca Limited process Direct Debit, Direct Credit and Standing Order payments.
Bound UCM	A UCM consisting of any combination of use case paths and responsibilities which is bound to a component substrate
CHAPS	Clearing House Automated Payments System: CHAPS Clearing Company Limited – provides electronic (same day) payment services to its members
COM	Component Object Model
Component Substrate	An organisational structure of abstract components
Confirmation	(in Scotland) The official certificate stating a will to be genuine and conferring on the executors power to administer the estate, cf. Probate
CORBA	Common Object Request Broker Architecture
CCV Number	Credit Card Verification or Card Code Verification Number
Data Warehouse	A database designed to allow managers and decision makers to answer questions about their business. The data in a data warehouse is read-only and may come from a variety of sources. cf. Operational Database
DBA	Database Administrator
DBMS	Database Management System
DoB	Date of Birth
DoD	Date of Death
EPOS	Electronic Point(s) Of Sales
ER	Entity Relationship
FAMS	*FlexiPlus* Accounts Management System – a Microsoft Visual Basic.NET application
FAMSDB	*FlexiPlus* Accounts Management System Database – a Microsoft Jet 4.0 and Microsoft Access 2002 operational database
FSA	Financial Services Authority
Graph	(from Graph Theory) A diagram in which points represent nodes or vertices and segments or curves represent edges

Graph Theory	The branch of mathematics concerned with the study and application of graphs and their generalisations
GUI	Graphical User Interface
Hypergraph	A graph structure that represents causal scenarios. A hypergraph specifies all the elements (hyperedges) that make up the use case paths
Hyperedge	An element of a hypergraph. Hyperedges contains the basic use case path constructs, eg. start points, end points, responsibilities, etc.
ITU-T	International Telecommunications Union Standardization Sector
LAN(s)	Local Area Network(s)
Link	An instance of an association, which connects two objects, that a message can be sent across
LINK	Automated Teller Machine network run by the company LINK® Interchange Network Ltd
LoA	Letters of Administration – legal authority for an appointed person or persons to take over, administer and dispose of an estate when there is no executor
MAN(s)	Metropolitan Area Network(s)
Metaclass	A class whose instances are classes. Metaclasses are typically used to construct metamodels.
Metamodel	A model that defines the language for expressing a model.
MS	Microsoft
MSC(s)	Message Sequence Chart(s)
Multithreading	Sharing a single CPU between multiple tasks (or "threads") in such a way that the time required to switch between threads is minimised
Multi-agent system	a collection of agents that work in conjunction with each other
OMG	Object Management Group
OMT	Object Modelling Technique
Operational Database	A database that is set up to handle transactions and is kept up to date as of the last transaction. cf. Data Warehouse
Path	(from Graph Theory) An alternating sequence of edges and vertices in a graph where all the vertices (except possibly the endpoints) are distinct.
PIN	Personal Identification Number – used to secure a bank account
PoA	Power of Attorney – legal authority for an appointed person to act on behalf of another
Probate	(in England and Wales) The official certificate stating a will to be genuine and conferring on the executors power to administer the estate, cf. Confirmation
Read-only	Capable of being displayed, but not modified or deleted
Relation	(from mathematics) An association between, or a condition satisfied by, ordered pairs of objects, numbers, etc.
RTGS	Real-Time Gross Settlement (system)
Scenario	A sequence of actions that a system performs which achieves a notable result of value to a particular actor
Sequel	Structured Query Language
SITE	School of Information Technology and Engineering, University of Ottawa, Canada
Software Architecture	the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them[19]
SQL	Structured Query Language
Superpose	(from mathematics) To bring into coincidence
Superposition	(from mathematics) (1) The act of superposing; (2) the state of being superposed

[19] Definition from: Bass, L, et al, *Software Architecture In Practice*, Addison Wesley Professional, 2nd Edition, 2003.

SWIFT	Society for Worldwide Interbank Financial Telecommunication – provides a payment and settlement network called SWIFTNET which connects its member banks.
SWIFTNET	an intranet shared across the entire financial industry which respects highly ambitious security and standard specifications, and enables communication of mission-critical financial information and transactional data.
TIA	Telecommunications Industry Association (USA)
UCD(s)	Use Case Diagram(s)
UCM(s)	Use Case Map(s)
UML	Unified Modeling Language
Unbound UCM	An incomplete UCM consisting of any combination of use case paths and Responsibilities which is not bound to a component substrate, cf. bound UCM
URN	User Requirements Notation
Use Case	a use case is a prose description of a set of sequence of actions that a system performs that yields an observable result of value to a particular actor.
Use Case Path	A path in a hypergraph which is a visual representation of a causal scenario
View	A subset of UML or UCM elements that shows a particular aspect of a system. There are five UML views: use case view, design view, implementation view, process view and deployment view. UCMs provide a sixth view.
VB	Microsoft Visual Basic .NET
VISA	VISA – provides electronic payment services to its members
WAN(s)	Wide Area Network(s)
XML	World Wide Web Consortium's (W3C's) eXensible Markup Language

Appendix 2

Overview of HP NonStop Servers

What are they?

You probably have heard that HP NonStop servers have 24 x 7 availability, and you may have heard them described as fault tolerant. But what you may wonder, Who needs this level of availability, and why is built-in fault tolerance important when clustering capabilities seem to achieve this quality?

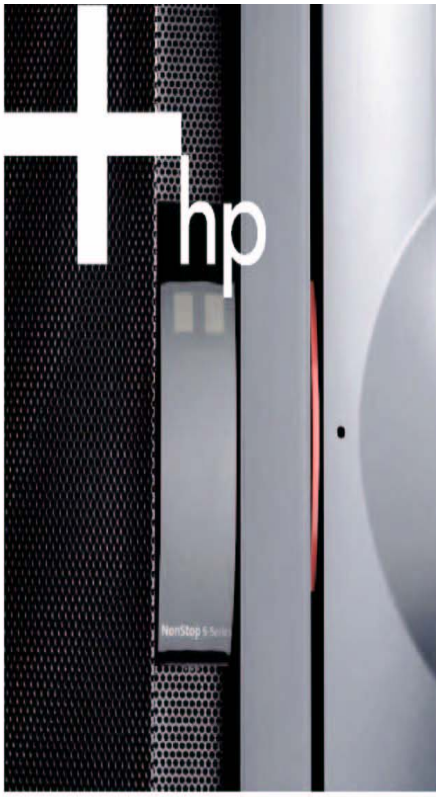
To understand the value of these servers to customers and to HP's portfolio, you should know that NonStop servers

- Are fault tolerant in both hardware and software
- Have a fully integrated stack (hardware, operating system, database, middleware)
- Provide online manageability of database, application, and hardware resources
- Utilize open standards
- Have been rated as providing the lowest total cost of ownership (TCO) among enterprise servers
- Offer simplified management using industry-standard tools in a heterogeneous environment, such as HP OpenView products

NonStop servers play an important role in HP's success by completing a competitive and powerful portfolio. In critical markets, the NonStop server is a powerful tool for customers and a weapon against high-end competition, especially IBM.

These top-rated servers deliver a better class of service than mainframes and act as the backbone of critical and complex IT environments. They are interoperable with other platforms, use open standards, and are more easily managed and more adaptive than mainframes. Plus, they are more affordable.

NonStop servers have a fully integrated stack—which means that the hardware, operating system, database, middleware, networking, and application program interfaces (APIs) are all designed to work in parallel together. An integrated stack achieves something unique: fault tolerance with virtually unlimited scalability. This quality is delivered transparently through standard interfaces to the application. As a result, the developer does not have to do anything special to take advantage of the computing environment.



Visit www.hp.com/go/nonstop

HP NonStop servers

Meeting high-end enterprise demands

invent

invent

Where do they fit in HP?

NonStop servers are powerful. They are best used for complex environments, key business-critical applications that require 24 x 7 availability, large databases for data integration and real-time computing, and real-time high-transaction environments.

The NonStop platform expands HP's portfolio. It is a better option for applications than mainframes and other fault-tolerant options and sets the bar with the highest level of availability in its class. A 2002 study by Gartner of multiple vendors of high-availability systems rated the NonStop platform on par with IBM Parallel Sysplex systems. The NonStop platform fared significantly better when measured with the NonStop SQL database against the IBM zSeries DB2.

The NonStop platform also runs an operational data store for the HP Zero Latency Enterprise (ZLE) architecture—a proven solution that integrates data and applications easily and simply across an enterprise in real time. As such, it truly spans HP's enterprise offerings. ZLE solutions leverage the full product line to reduce lag time in business-critical processes, providing numerous cost and quality benefits to stimulate growth and avert disasters.

NonStop servers also play an important role in the HP Adaptive Enterprise strategy.

- **Management:** OpenView management operation capabilities provide the opportunity for managing a complete environment.
- **Enterprise integration:** Real-time ZLE solutions are an integrated part of the HP Adaptive Enterprise strategy.
- **IT consolidation:** Data integration capabilities make NonStop servers a valuable consolidation and integration platform.
- **Business continuity:** A full portfolio of disaster-tolerant capabilities complement business-critical solutions.
- **Managed services:** NonStop solutions fit well in the managed service environment.
- **Partnerships:** The platform extends partnerships with independent software vendors and leading consultant/systems integrators.
- **Real-time enterprise:** NonStop servers are the hub in ZLE cross-platform solutions.

What we do for customers

NonStop servers are the backbone for businesses that run the world's most demanding computing environments—enterprises that remain online, all the time. NonStop technology is advantageous in the following areas:

Where high availability and scalability are required

- Companies that require 24 x 7 uptime.
- The NonStop platform is the only one that can deliver complete application availability—not just system availability.
- It offers extreme scalability to thousands of processors.
- Each additional processor provides virtually the same performance as the first, no matter how many are added.
- Examples of companies with such need: CRESTCo, Sabre.

In complex computing environments

- Companies that require simultaneous handling of complex, real-time transaction processing, standard queries, and large ad hoc queries against terabytes of storage.
- NonStop technology provides a single database image across all processors, so the database can be configured, monitored, and managed as a unified entity.
- Example: Nasdaq next-generation trading system.

With real-time enterprise computing

- 24 x 7 real-time–data environments.
- The NonStop platform is the hub of the ZLE architecture.
- NonStop systems offer a complete end-to-end solution, from data cleansing to customer relationship management (CRM) and supply chain integration.
- They support business processes with real-time decision support.
- Database technology allows for fast response time (seconds).
- NonStop technology provides very large database support with superior online manageability.
- Examples: Sprint, Banamex.

Myths

Here are some myths and facts about NonStop servers.

Myth: Proprietary

In fact, the NonStop platform

- Delivers open standards (CORBA, Java, C++, Tuxedo, BEA WebLogic Server)
- Will be moving to industry-standard 64-bit Itanium® processor
- Is interoperable with other platforms (Windows, Linux, UNIX®)
- Is easily managed and adaptive through industry-standard tools like OpenView
- Will support HP XP storage (by the end of 2004)

Myth: Installed base only

- Customers include many high-profile Fortune 500 companies (NYSE, AOL, and Nasdaq).
- New growth opportunities exist in current and emerging markets.
- Each new NonStop server customer typically doubles the size of its system within 18 months.
- NonStop server sales pull other product purchases.

Myth: Legacy

- NonStop servers lead with current standards and emerging technologies (IPv6, J2EE, XML, Web services).
- They have a leadership position in key market areas (finance, securities, telecommunications).
- They offer opportunities for new innovative solutions in current markets, such as real-time e-payments in finance.
- Emerging markets need this class of server for real-time data integration.

Myth: Expensive

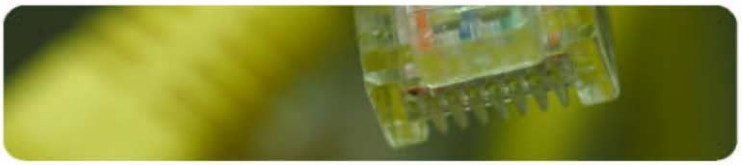
- In fact, NonStop servers offer the lowest TCO over a three-year period (Standish Group).
- TCO includes application development, deployment, ongoing management, and cost of downtime over a multiyear period.

Appendix 3

Overview of Voca Limited Connectivity Products

VOCA

BACSTEL-IP
Connectivity Products



Voca connectivity products enable organisations to submit payments and collect reports, into and from BACSTEL-IP®

Broadband Direct

This is a new BACSTEL-IP submission method available to users who want a low cost B2B broadband connection for their BACSTEL-IP submissions and report collections. It provides users with a self-install contended ADSL broadband connection into Voca's secure network. The service is aimed at users who want to benefit from moving to low cost broadband technology linked into Voca's secure private network for their financial payments. It is particularly suitable for current dial up customers such as ISDN users because the connection is 'always on' and there are no call charges. For more information visit: www.voca.co.uk/connectivity

DSL Connect

This connection method is designed to provide submitters with un-contended 256kbps Digital Subscriber Line (DSL) connectivity, from their site(s) to Voca's closed user group, Virtual Private Network (VPN). The service is designed for medium to large submitters who do not require the resiliency of the Fixed Extranet Connect service for submitting payments and/or collecting reports, but want to benefit from un-contended 256kbps connectivity into the BACS service with 4-hour fault response and Helpdesk support and progress reporting. DSL Connect uses the latest 'always on' DSL communications technology, which ensures that submitters avoid dial-up line drops and call charges. For more information visit: www.voca.co.uk/connectivity

Fixed Extranet Connect

This connection method provides customers with two fully installed fixed connections (lines and routers) from Voca's closed user group, Virtual Private Network (VPN). The service is designed for people submitting either large item volumes or high item value who require a dedicated, resilient, 'always on', managed connection, with speeds from 256kbps up to 2Mb+. This service extends Voca's network directly to the customer's site(s). This ensures customers benefit from a fully managed service, with delivery and processing guarantees, and 99.99% service availability. For more information visit: www.voca.co.uk/connectivity

Contact us

Sales Team
sales@voca.co.uk
Direct line:
0870 920 8052

www.voca.co.uk

Voca supplied BACSTEL-IP connectivity solutions

	Broadband Direct	DSL Connect	Fixed Extranet Connect
Delivery, security and availability	• Contended ADSL access into Voca's managed network • Additional security provided at network & transport levels • Fault response and fix SLA's on ADSL line with premium rate helpdesk support.	• 'Always on' un-contended DSL access into Voca managed VPN (Virtual Private Network) • Additional security provided at network & transport levels • Delivery & processing guarantee once submission is on VPN.	• Extension of Voca network into customer site(s) providing 99.99% delivery and processing guarantees • Additional security provided at Network (VPN) and Transport (SSL) levels • Voca proactively managed and monitored resilient infrastructure • Delivery & processing guarantee from Voca managed service to customer site installation
Installation	• Voca provision of ADSL capability over existing customer PSTN line with self install router to enable 'always on' submissions via Voca managed network.	• Full Voca installation of DSL circuit and Cisco 837 router to enable 'always on' submissions via Voca's managed network.	• Full Voca installation of twin BT IPClear circuits and Cisco 2691 routers to extend Voca's Virtual Private Network into customer site(s).
Connection types and speed	• ADSL connection into Voca managed VPN providing 64-256kbps upstream and 256kbps downstream connectivity.	• Un-contended DSL connection into Voca managed VPN providing 256kbps downstream and 256kbps rate adaptive upstream connectivity.	• Choice of IPClear connectivity starting at 256kbps • Flexibility to upgrade connectivity speed up to 2Mb and beyond to meet future business needs.
Submission volumes	• 256kbps, approximately 400,000+ items per hour.	• 256kbps, approximately 400,000+ items per hour.	• 256kbps, approximately 400,000+ items per hour • 512kbps, 1Mb and 2Mb Fixed Extranet Connect services available for large volumes.
Cost	Installation - none Maintenance - £19.99 per month (per Broadband Direct connection)	Installation - £2,990 Maintenance - £2,990 (per DSL Connect connection)	Installation - £22,721 Maintenance - £24,208 (Standard 256k twin connection)

Existing DialPlus users will save the current £5 monthly charge when switching to a Voca connectivity solution.

www.voca.co.uk/connectivity

Connectivity Products
Data Sheet

	Broadband Direct	DSL Connect	Fixed Extranet Connect
Network	Provision of contended ADSL link into Voca over existing customer PSTN line (PSTN line must exist at installation address prior to order being placed).	Uncontended DSL connection to BT exchange across PSTN line and then join managed VPN on Voca network.	Extension of Voca's resilient IPClear network into customers' premises.
Contention	5:1 into Voca then 25:1 across managed Voca network.	Uncontended into Voca then 10:1 across managed Voca network.	Continuously profiled to provide optimum connectivity performance.
Bandwidth	• 64-256k upstream (into Voca) • 256k downstream (from Voca).	• 256k upstream (rate adaptive) • 256k downstream.	Up to 2Mb (with higher bandwidth available on request).
Installation	Self-install router with premium rate telephone support.	Fully engineered installed router with managed line test (if existing PSTN does not exist then Voca can install PSTN line as part of DSL Connect installation).	Full installation plus project management and technical installation support.
Router	Linksys AG0241.	Cisco 837.	Cisco 2691 (rack mounted).
Managed Service	None.	Customer raised support calls managed by Voca helpdesk with Voca management of any resultant 'fix' by Voca's third party support.	Full pro-active network monitoring support management and fault fix.
Helpdesk	Premium rate telephone helpdesk and third party support.	Voca helpdesk and fault validation.	Pro-active helpdesk with full account management service.
SLA	• ADSL line (24/7) 4hr response with 40hr target fix. • Next day courier of Linksys AG0241 router once fault has been diagnosed.	• DSL line (24/7) 4hr response with 24hr target fix. • 4hr response for Cisco 837 router (Mon-Sat 8am-5pm).	• IPClear line(s) and Cisco 2691 router (24/7) 4hr response and 5hr target fix.
Multi-line resiliency	None.	Customer initiated using HSRP within standard DSL Connect router configuration.	Twin IPClear lines installed as standard.
Cost	Installation - none Maintenance - £19.99 per month (per Broadband Direct connection).	Installation - £2,990 Maintenance - £2,990 (per DSL Connect connection).	Installation - £22,721 Maintenance - £24,208 (Standard 256k twin connection).

www.voca.co.uk/connectivity

Appendix 4

BACS Payment Schemes Limited' Member Banks and Building Societies[20]

- ABBEY
- ALLIANCE AND LEICESTER
- BANK OF ENGLAND
- BANK OF SCOTLAND (HBOS)
- BARCLAYS
- CLYDESDALE
- COUTTS & CO
- THE CO-OPERATIVE BANK
- HSBC
- LLOYDS TSB
- NATIONAL WESTMINSTER BANK
- NATIONWIDE
- NORTHERN ROCK
- ROYAL BANK OF SCOTLAND

This list is correct as at August 2004.

[20] List taken from www.apacs.org.uk

Appendix 5

LINK® Interchange Network Ltd.' Member Banks, Building Societies and Independent ATM Deployers

- Abbey National plc
- Aidrie Savings Bank
- Alliance and Leicester
- Allied Irish Bank
- American Express
- Bank Machine *
- Bank of Ireland
- Bank of Scotland plc
- Barclays plc
- Bradford & Bingley plc
- Bristol & West
- Britannia Building Society
- Calypso Europe *
- Cardpoint plc *
- Cashbox *
- Chelsea Building Society
- Citibank Savings
- Co-op Bank
- Coventry Building Society
- Creation Financial Services
- Cumberland Building Society
- Derbyshire Building Society
- Diners Club (UK) Ltd.
- Dunfermline Building Society
- Egg (Prudential)
- G E Capital
- Halifax plc
- Hanco *
- Lloyds TSB plc
- Moneybox *
- National Westminster Bank plc
- Nationwide
- National Australia Group **
- National Savings & Investments
- Northern Rock
- Norwich & Peterborough
- Omnicash *
- Paypoint *
- Royal Bank of Scotland
- Sainsbury's Bank
- Scott Tod Developments *
- Securicor *
- T D Waterhouse
- Tesco Personal Finance
- Travelex
- TRM Corporation *
- Woolwich
- Yorkshire Building Society

* Independent ATM Deployers
** Includes Clydesdale Bank, Northern Bank and Yorkshire Bank

Appendix 6

Cheque and Credit Clearing Company Members[21]

- ABBEY
- ALLIANCE AND LEICESTER
- BANK OF ENGLAND
- BANK OF SCOTLAND (HBOS)
- BARCLAYS
- CLYDESDALE
- THE CO-OPERATIVE BANK
- HSBC
- LLOYDS TSB
- NATIONAL WESTMINSTER BANK
- NATIONWIDE
- ROYAL BANK OF SCOTLAND

This list is correct as at August 2004.

[21] List taken from www.apacs.org.uk

Appendix 7

Correspondence between the author and HBOS plc

		Mr. William S. Blackley
		MSc IT Student
		University of Paisley
		Paisley
		PA1 2BE
		Scotland
		Email: *wsblackley@hotmail.com*
Miss Jennifer Mosley		
LINK and Debit Card Services		
Halifax plc		Your Ref.:
PO Box 101		
Copley Data Centre		
Copley		10th March 2005
Halifax		
HX3 0TA		

Dear Miss Mosley,

How do Visa Debit Card Transactions Work?

I am currently working on my MSc in Information Technology Research Project. As part of the project, I am designing and partially implementing a software application which will allow a hypothetical clearing bank's staff to maintain its current account holder's accounts. The hypothetical bank is called *BestBank*.

The hypothetical current accounts are supposed to allow Visa debit card transactions. In order that I can set up the correct framework for designing and implementing the hypothetical bank's software application in my project, I would be extremely grateful if you could tell me if I am right in thinking that Visa debit card transactions work in the same way as Automatic Teller Machine transactions work over the LINK® cash machine (ATM) network - see enclosed word document entitled “How Automated Teller Machine transactions work”.

Can you please tell me, is my summary of how LINK® cash machine ATM transactions work correct? Do Visa debit card transactions use the LINK® cash machine (ATM) network? If not, can you please tell me how they do work?

Any advice you can give me will be of immense help to me.

I look forward to hearing from you.

Yours sincerely,

William S. Blackley
MSc IT Student (part-time)
University of Paisley.

Enc.

How Automated Teller Machine transactions work.

When a customer initiates a transaction, e.g. a withdrawal of money, through an ATM, the ATM forwards the transaction information to LINK's server. The information is then forwarded to *BestBank's* server. If the cardholder is requesting cash, LINK's server initiates an electronic funds transfer to take place from the customer's bank account to LINK's bank account. LINK's server will then send an approval code to the ATM authorizing the machine to dispense the cash. LINK's server then transfers the cardholder's funds into the merchant's[22] bank account, usually the next business day. The merchant is reimbursed for all funds dispensed by the ATM. All ATM transactions take place in real-time. See Figure 3.1 below.

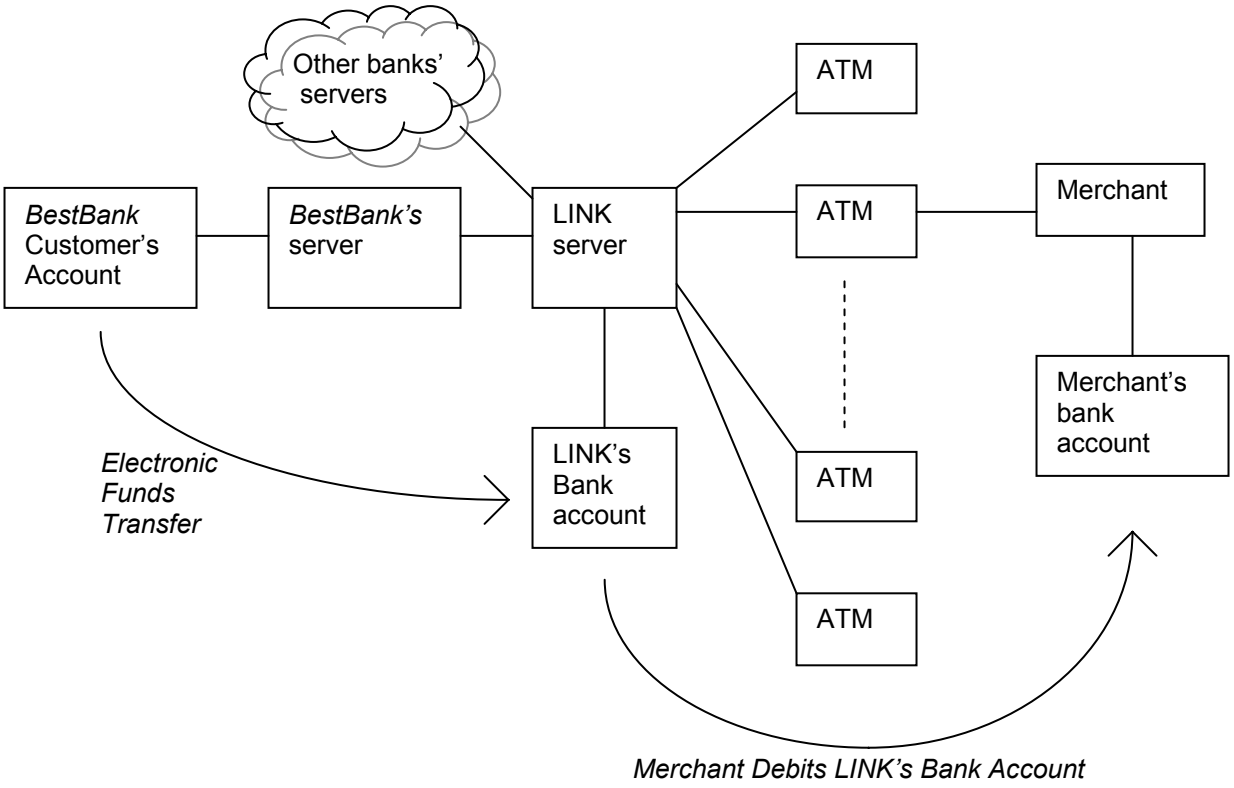


Figure 3.1: ATM cash withdrawal over LINK® cash machine (ATM) network

[22] The term *merchant* refers to anyone who hosts an ATM machine on their premises, e.g. a bank, a convenience store, a public house, etc.

TR/3/L&DCS/JM

16th March 2005

Mr Bill Blackley
MSc IT Student
University of Paisley
Paisley
PA1 2BE
Scotland

HBOS plc
LINK and Debit Card Services
P O Box 101
Halifax
West Yorkshire
HX3 0TA
Direct Line 08706060240
Fax No 01422 391274
Office Hours: Monday to Friday, 9am – 5pm

Debit Card Transactions

Dear Mr Blackley

am writing in response to your letter dated 10th March 2005.

A member of our LINK team has looked over your information on ATM transactions and they have provided me with the following information. The customer initiates a transaction at an ATM. The ATM sends a request through the LINK server to the customers bank. The bank then checks the customers account and authorises the request or declines the transaction. The customers bank then sends a response back to the ATM via the LINK server to request that it either issues the cash or declines the transaction. The ATM issues the cash and the customers account is debited with the amount by their bank immediately. The settlement of funds is done via LINK and Bank of England.

For debit card transactions I will provide a brief outline of how they work.

When you go into a shop to make a purchase, your card is swiped and the retailer goes on-line to seek authorisation. This request is sent through the Acquiring bank (the retailers bank) and then on to us. We check that your card is valid and there are funds in your account. If all the details are correct we will issue an approval message. Again, this goes back down the chain through the Acquiring bank and then on to their retailer. The approval response is received on their terminal and an authorisation code is provided. The funds will then be held to one side on your account for a number of days to allow the retailer time to process the debit. When the retailer submits the debit this goes to their Acquiring bank who then pass the debit to your account.

hope this information will be of use to you.

Yours sincerely

Jennifer Mosley
Customer Service Assistant
LINK and Debit Card Services

Appendix 8

Correspondence between the author and LINK® Interchange Network Ltd.

The following email correspondence led to a telephone conversation between the author and Martyn Gould, Card Scheme Director, LINK® Intrechange Network Ltd., in which Mr Gould fully explained the authorization procedures of ATM transactions.

From: Bill Blackley [mailto:wsblackley@hotmail.com]
Sent: 18 March 2005 18:25
To: George Donovan
Subject: Re: How do ATM Transactions work?

That is great news. I'll call him on Monday.

Once again, thank you very much indeed for your help.

Regards,

Bill.

Bill Blackley
MSc IT Student (part-time)
University of Paisley

----- Original Message -----
From: George Donovan
To: Bill Blackley
Sent: Friday, March 18, 2005 5:20 PM
Subject: RE: How do ATM Transactions work?

Bill

Our Card Scheme Manager, Martyn Gould, has offered to go through the process with you, as there are a couple of errors in your document.

Martyn can be reached on 01423 356271 (direct line).

Kind regards
George

From: Bill Blackley [mailto:wsblackley@hotmail.com]
Sent: 18 March 2005 16:18
To: George Donovan
Subject: Re: How do ATM Transactions work?

Thanks very much indeed for letting me know.

Regards,

Bill.

Bill Blackley
MSc IT Student (part-time)
University of Paisley

----- Original Message -----
From: George Donovan
To: Bill Blackley
Sent: Friday, March 18, 2005 4:12 PM
Subject: RE: How do ATM Transactions work?

Bill

I am just awaiting a response from our Card Scheme Director and will advise as soon as I receive it.

Regards

George Donovan
Product Marketing Manager
LINK Interchange Network Ltd
Tel. 01423 356284
Mob. 07974 326284
Email gdonovan@link.co.uk
Website: http://www.link.co.uk

From: Bill Blackley [mailto:wsblackley@hotmail.com]
Sent: 18 March 2005 15:08
To: #Information
Subject: Re: How do ATM Transactions work?

Dear Sirs,

Further to the email below, can you please give me an idea of when I might expect a reply?

Thanks very much for your help, in anticipation.

Regards,

Bill.

Bill Blackley
MSc IT Student (part-time)
University of Paisley

----- Original Message -----
From: Bill Blackley
To: linkinfo@link.co.uk
Sent: Friday, March 11, 2005 5:30 PM
Subject: How do ATM Transactions work?

Dear Sirs,

I am currently working on my MSc in Information Technology Research Project. As part of the project, I am designing and partially implementing a software application which will allow a hypothetical clearing bank's staff to maintain its current account holder's accounts. The hypothetical bank is called *BestBank*.

The hypothetical current accounts are supposed to allow LINK® cash machine (ATM) network transactions. In order that I can set up the correct framework for designing and implementing the hypothetical bank's software application in my project, I would be extremely grateful if you could tell me if my summary of how the LINK® cash machine (ATM) network works is correct - see attached word document.

Any advice you can give me will be of immense help.

I look forward to hearing from you.

Regards,

Bill.

Bill Blackley
MSc IT Student (part-time)
University of Paisley

This email has been scanned for all viruses by the MessageLabs SkyScan service. For more information on a proactive anti-virus service working around the clock, around the globe, visit http://www.messagelabs.com

---

LINK Information Security

Appendix 9

Case Study Validation Letters

The letter below, from the author, was sent to the following recipients:

- HSBC Bank
- Abbey National plc.
- LogicaCMG

The responses received from them follow. Chapter 3 – The Case Study was altered in light of the comments received from HSBC, Abbey and LogicaCMG.

		Mr. Bill Blackley
		MSc IT Student
		University of Paisley
		Paisley
		PA1 2BE
		Scotland
		Email: wsblackley@hotmail.com
Company Name		
Address Line 1		
Address Line 2		Your Ref.:
Address Line 3		
City		
Post Code		06 May 2005

Dear Sirs,

MSc in Information Technology Research Project

I am currently working on my MSc IT Research Project at the University of Paisley.

The project will evaluate the benefits of adding a Use Case Map view to existing Unified Modeling Language views of an object-oriented software system. This includes ascertaining if adding a Use Case Map view to existing Unified Modeling Language views helps bridge the gap between requirements and use cases and more detailed views, such as interaction diagrams (sequence diagrams and collaboration diagrams) and statechart diagrams. Also, a comparison of Use Case Maps and Use Case Diagrams will be given.

.

In order to do this, a prototype of an application for managing a hypothetical bank's customers' accounts is to be developed. The application is to be used by the bank's branch staff and is connected to the bank's database. A prototype of the database will also be developed.

The third chapter of my project, which is entitled “Chapter 3 – The Case Study” (enclosed), outlines the demonstration project that is to be developed in later chapters. In particular, section 3.2 of Chapter 3 contains the requirements specification for the application which is to manage the bank’s customers’ accounts.

I would be extremely grateful if you will tell me if the requirements specification, and the Case Study as a whole, seems accurate and realistic. In order for my research to receive some objective evaluation the requirements specification, which is the plank on which the project rests, requires external validation. It will be greatly appreciated if you would provide that external validation by commenting on the requirements specification.

Any advice you can provide me with will be of immense help.

I look forward to hearing from you.

Yours faithfully,

William S. Blackley
MSc IT Student (part-time)
University of Paisley

Enc.

The following email was received in response to my letter of 6th May 2005 to Dr Bill Kay, Principle Consultant, Financial Services, LogicaCMG. Chapter 3 – The Case Study was amended after this email was received.

From: Kay, Bill
To: Bill Blackley
Sent: Thursday, July 21, 2005 9:05 AM
Subject: RE: re your MSc project

Here are my review comments on the case study section of your project.

Overall I thought it provided a good view of an account system.

Section 3.1: it is worth stating that the FexiPlus account is only available to individual personal customers (this rules out joint accounts, business accounts and accounts for charities and societies, all of which require more complex object/data models).

Section 3.1.2: the BACSTEL interface is being replaced with BACSTEL-IP.

Section 3.1.3: all CHAPS payment requests from customers must be submitted by 16:00. Later CHAPS transfers are used for settlements between Banks, usually their Treasury departments.
3.2 Requirements.

Interest calculation: it would be worth clarifying what 'immediately' means in terms of earning interest. For a product such as FlexiPlus, I would expect it to be calculated on the balance at close of business on each working day. Also I would expect interest to be credited to the account monthly (on the same basis as overdraft charges) rather than annually.

Interest calculation, fees and charges (such as overdraft charges) would not normally be done by a Database Administrator. Rather than model these requirements, you could say that the Bank already has a separate system which handles interest calculations, fees and charges, and that this will access and update the FAMS database directly.

Payment of Bills: most banks now offer this from ATMs, phone banking, Internet banking, and by Direct Debits and Standing Orders as well as via the Branch. This would normally be done by other systems. Again, to avoid detailed modelling of the requirements, you could say that these systems will access the FAMS database directly.

Other system interfaces: balance and transaction details need to be copied to accounting systems (general ledger), reporting systems (for things like management reports, risk reports statutory reporting to Bank of England, FSA and the Inland Revenue) and anti money laundering detection. Again you could state this is handled by other systems that access your database directly and are therefore outside the scope of the FAMS development.

Bank requirements often cover a lot of ‘non-functional’ requirements such as volumes, throughput, response times, security, auditing, usability, user documentation, online help, training, testing facilities, availability, reliability, disaster recovery, hours of operation, archive, backup and restore etc. You could reasonably expect BestBank to have a Group Technology Services section which maintains and polices the standards and policies to cover all this. It would be worth alluding to these in passing, if only to state that they are dealt with elsewhere.

Customer Details records are often quite complex, as there is a need to identify all accounts belonging to a customer: for customer relationship management, risk control, security details and the like. This often requires multiple addresses (e.g. students may have different term-time and home addresses), multiple phone numbers and history records to track changes in address. Rather than get embroiled in this, I suggest you mention that the Bank already has a customer database in a separate system, and that FAMS need only hold the subset of attributes identified in your paper.

Account status. I suggest you add another status of ‘Dormant’. Banks often have idle accounts where the customer has forgotten about it, is incapacitated or has died and the executors have not contacted the Bank. Banks normally flag accounts as dormant once there have been no user transactions for a year. An account that has been dormant for many years may eventually be closed, and the balance transferred (or written off if overdrawn) to an internal bank suspense account to make the ledgers balance.

Statement printing requires a technical infrastructure: specialised network printers, branded paper and so on. Again, it would be worth mentioning this, but stating that these are already in place and therefore outside the scope of the FAMS development programme.

Good luck with your investigations into Use Case Maps. I have found that requirements modelling using techniques such as use cases and UML diagrams are effective but sadly under-used in financial services.

Regards
Bill

HSBC

Mr W S Blackley
Flat 20
Dorchester Court
2 Dorchester Place
Glasgow
G12 0BX

25 May 2005

Dear Mr Blackley

Thank you for your letter of 6 May, which has been passed onto me from our Canada Square office. I apologise for the delay in replying – it took a few days for your letter to filter down to me. I am a Systems Analyst and I lead a team that carries out analysis for the HSBC banking systems used in our branch network and in centralised service centres.

You wanted comments on whether the requirements specification and case study seem accurate and realistic.

Here are my comments:

The requirements letter contains more than requirements in that it specifies a single Windows application. It therefore places constraints on the design. You, as recipient of the letter, would probably want to question this and work with the project initiator to arrive at the true requirements – use cases would be an appropriate technique to achieve this.

I think it unlikely that your Head of Product Development would requisition a system specifically to manage a particular account type such as the Flexiplus account. The products offered by banks evolve and change too quickly for this to be viable.

The customer is at the heart of banking systems these days. Best Bank would want its customer data to be held in one place i.e. in one central database (I think your case study does imply this).

It seems unlikely that Best Bank would want its customer data to be maintained from within FAMS.

It is likely that Best Bank would be looking for a lower level of granularity in its applications, so that business functions such as 'Maintain customer data', 'Post interest', 'Pay bill', 'Print statement' could be used for different product (account) types rather than just for Flexiplus accounts.

Continued

HSBC Bank plc
41 Silver Street Head, Sheffield S1 3GG
Tel: 0114 252 8000

Here in HSBC, we use a facility called 'key services', which enables the user to move between the various business functions on the menu without having to retype the key (e.g. the customer number or the account number). Key services facilitate the lower level of granularity. Your Head of Product Development might well have requirements regarding how the new functionality should interface with existing applications.
I haven't reviewed your sections on ATM transaction procedures etc, as I thought it more important to concentrate on the realism of the requirements aspects of the paper.

In summary, the Requirements Specification resembles a project initiation letter rather than a set of requirements that you could take at face value. As recipient, your first steps might be to agree terms of reference for the project and then make use of use cases to research and agree the detailed requirements.

I hope these comments are helpful. Best wishes for the success of your project.

Yours sincerely

Catn Baldwin

Mrs C Baldwin
Senior Systems Analyst
Internal Systems, HSBC Bank plc

abbey

Shenley Wood House
Chalkdell Drive
Shenley Wood
Milton Keynes
MK5 6LA

Mr. William S. Blackley
Flat 20,
Dorchester Court
2 Dorchester Place
Glasgow
G12 0BX

31 May 2005

Dear Mr.Blackley

MSc in Information Technology Research Project

I am in receipt of your letter dated 6 May wherein you asked for comments on your Case Study. The views expressed here are mine and not necessarily the views of Abbey. I have not tried to be exhaustive either.

Requirements gathering is the most important part of the project, so the more depth and detail the better. Layout of a requirements document could include some or all of the following:

1) Version/change history and sign off authorities. Requirements change so you need to keep track of what you have done. Some requirements are replaced because they are wrong and therefore do not get implemented, but it is possible to have multiple versions of requirements all being developed concurrently for implementation in different phases.

2) Management summary.

3) Introduction - why are we doing this, stakeholders, other documents to refer to.

4) Project summary - project objectives, success criteria, scope, phasing of deliveries, environment (locations it will be used in, types of users/roles), constraints (budgets, deadlines) plus any assumptions and dependencies on other projects, hours of operation when the system is available.

If calculations are required show the algorithms, particularly key for financials, e.g. interest - is a days interest a 1/365th or 1/366th? leap years? calculations on round pounds only? or include pence? when does interest accrue from - day after deposit?, bank holiday treatment - Scotland, NI, England... ?.

These calculations will end up in the detailed Terms and Conditions of the account.. and some customers do check the interest to the penny/day. you need to be clear - examples can help , e.g. customer opening account and closing it the same day - how much interest would your bank be expecting to pay?

5) List each requirement. Recognise that requirements gathering is only one stage of a project - it needs to link in with future stages. Each separate requirement needs to be uniquely identified with an ID number. If this ID is then carried forward into detailed technical requirements, programme specs, test plans you then have traceability both ways... you can check forward that all requirements have been implemented in testing. You can also check backwards in the event of an issue to see which requirement to go back to.

   For each requirement - need a scope, inputs (triggers), outputs (screens, reports), process flow, error handling, benefits (mandatory/optional, financial. in case you need to prioritise later), acceptance criteria, how you will test it...

6) Management Information requirements are not mentioned.

7) You have focussed on the functional requirements of the system, i.e. what it does. If that is the guidance you have been given by your tutor that is fine, but perhaps state that this is the scope and therefore specifically exclude non-functional requirements. If you were to mention non-functional requirements then the sort of things you would be looking at are performance, e.g. transaction response times, volumes (daily transaction volumes, peaks, etc), security (user sign-on controls/passwords, etc), archiving, service levels to be met, capacity of the system (how many accounts, etc), disaster recovery (e.g. how soon after a failure does the system have to be back on line).

8) You do need to include relevant legislative requirements, e.g. under the Disability Discrimination Act you need to recognise both the needs of the staff using the system (colours on screens, text size), and the customers who have the products (braille statements, etc).

Within your case study.... probably don't need to describe how link, CCC, CHAPs, etc work - the rules of these schemes are well defined and can be referenced.

You don't need to refer to windows, HP, etc. The IT department will take the requirement and create a technical specification which will add in all the technology decisions about platforms and technical architectures.

General points on the counter transactions. Balancing of the tills is an important requirement; reconciling cash, cheques, etc at the end of the day. You also need to keep an audit log of each till to satisfy investigations in the event of queries later. Need to be clear about authority levels, eg. how much a day can a customer draw out in cash? eg. when would a more senior person have to get involved to release a bankers draft? Plus you'll need to link in with your general ledger.

I agree you would not make funds available until cheques cleared, but you wouldn't hold interest back until then.

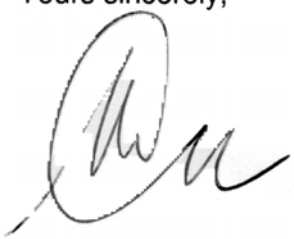
Best to avoid putting absolute values in the requirements document unless they really are absolute. Interest rates, charges, etc You need to describe the approach to interest rates (e.g. are you offering tiers for different balances?, how often might rates change?) so that the system can manage the rates, but do not need to mention "1% below base rate" - that is something for the marketing literature not the system. Marketing departments have been known to change their minds, so you need to keep volatile data out of the spec - interest rates and charges would be put into a separate area of the system with controlled access/processes in marketing to enable the details to be changed.

Need to be able to handle account adjustments - mistakes happen.

Student account. If you are offering different types of account it helps to show a table of features indicating any different treatment for the different account types. You also need an account type field in your data list. Watch out for "difficult to police" requirements. Accounts remain "student" with a free overdraft until 1 year after graduation, so how does the system “know” when graduation occurs?

Hope some of this helps.

Yours sincerely,



Mike Costello
Head of Corporate Systems
Abbey

Appendix 10

FAMS Application Source Code

CloseButton Source Code

++

Class frmAccountDetails Source Code

++

Class frmBankersDraft Source Code

++

Class frmCashDeposit Source Code

++

Class frmCashWithdrawal Source Code

++

Class frmChangePassword Source Code

++

Class frmChequeDeposit Source Code

++

Class frmCorrespondenceAddress Source Code

++

Class frmCustomerDetails Source Code

++

Class frmInterAccountTransfer Source Code

++

Class frmLogin Source Code

++

Class frmMainMenu Source Code

++

Class frmStatements Source Code

++

Class Transaction Source Code

Bibliography

[1] Amyot, D, *Introduction to the User Requirements Notation: Learning by Example.* The International Journal of Computer and Telecommunication Networking, Vol 42, Issue 3, June 2003. Elsevier North-Holland, Inc. 2003.

[2] Amyot, D, (1999) *Use Case Maps and UML for Complex Software-Driven Systems.* Unpublished. Online at http://www.usecasemaps.org/pub/uml99.pdf.

[3] Amyot, D and A Eberlein, *An Evaluation of Scenario Notations and Construction Approaches for Telecommunications Systems Development.* Telecommunication Systems Journal, 24:1, 61-94, September 2003.

[4] Amyot, D and G Mussbacher, (2001) *Bridging the Requirements/Design Gap in Dynamic Systems with Use Case Maps (UCMs).* Online at http://www.usecasemaps.org/pub/icse01.pdf

[5] Amyot, D and G Mussbacher, *On the Extension of UML with Use Case Maps Concepts.* 3rd UML Conference, York, England, 2000. Lecture Notes In Computer Science, Vol 1939 pp16-31, 2000.

[6] Amyot, D and G Mussbacher, (2002) *URN: Towards a New Standard for the Visual Description of Requirements.* SITE, University of Ottawa.

[7] *Association of Payment Clearing Services Website*. Online at http://www.apacs.org.uk/

[8] *Automatic Bankcard Services Website.* Online at http://www.atmwholesale.com/index.html

[9] BACS Service Support, (2000) *BACS Guide To Electronic Funds Processing.* Online at http://www.aptbacs.co.uk/Downloads.htm

[10] *BACS Website*. Online at http://www.bacs.co.uk/home/landingpage.php

[11] *BACSTEL-IP Website*. Online at http://www.bacstel-ip.com/home/index.php

[12] Booch, G, I Jacobson and J Rumbaugh *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[13] Booch, G, I Jacobson and J Rumbaugh *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[14] Britton, C and J Doake, *Object-Oriented Systems Development.* McGraw Hill, 2000.

[15] Buchi, M, *The B Bank: A Complete Case Study*. 2nd IEEE International Conference on Formal Engineering Methods, 9-11 December 1998, Brisbane, Australia, pp 190-200.

[16] Buhr, R J A, *Use Case Maps and UML*. Slide package, Carleton University, Canada, December 1998. Online at http://www.UseCaseMaps.org/UseCaseMaps/pub/ucm_umlSlides98.pdf

[17] Buhr R J A, *Use Case Maps as Architectural Entities for Complex Systems.* IEEE Transactions On Software Engineering, Vol 24, No 12, December 1998

[18] Buhr, R J A and R S Casselman, *Use Case Maps for Object-Oriented Systems.* Prentice Hall, 1996.

[19] *Citeseer.IST Scientific Literature Digital Library*. Online at http://citeseer.ist.psu.edu/

[20] Connolly, Thomas M and Carolyn E Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management.* 3rd Ed. Addison Wesley, 2002

[21] Deitel H M, P J Deitel and T R Nieto, *Visual Basic .NET: How to Program.* Prentice Hall, 2002

[22] *Elsevier Website*. Online at http://www.elsevier.com/wps/find/homepage.cws_home

[23] Financial Ombudsman Service (December 2004/January 2005) *Ombudsman News, Issue 42.* Online at http://www.financial-ombudsman.org.uk/publications/ombudsman-news/42/42.htm

[24] *HBOS plc Website*. Online at http://www.hbosplc.com/home/home.asp

[25] International Telecommunications Union Standardisation Sector (ITU-T)*, Recommendation No. Z.152: URN – Use Case Map Notation (UCM) – Version 3.0,* Sept 2003. Online at http://www.usecasemaps.org/urn/urn-meetings.shtml#latest

[26] *International Telecommunications Union Website*. Online at http://www.itu.int/home/

[27] *Link Interchange Network Ltd Website*. Online at http://www.link.co.uk/

[28] McCready, M, *Object-Oriented Analysis & Design Course Notes – Part 1 and Part 2.* School of Computing, University of Paisley, Scotland, 2003

[29] McMonnies, A, *Object-Oriented Programming in Visual Basic .NET* Pearson: Addison Wesley, 2004

[30] Myers, Glenford J, *The Art Of Software Testing.* John Wiley & Sons, Inc. 1979.

[31] *Object Management Group Website*. Online at http://www.omg.org/

[32] Pooley, R and P Stevens, *Using UML: Software Engineering with Objects and Components.* Addison-Wesley, 1998.

[33] *Royal Bank of Scotland Website.* Online at http://www.rbs.co.uk/

[34] *UML Use Case Diagrams: Tips and FAQ*. Online at http://www.andrew.cmu.edu/course/90-754/umlucdfaq.html

[35] *Use Case Maps Quick Reference Guide.* Online at http://www.usecasemaps.org/pub/UCMtutorial/UCMquickRef.html

[36] *Use Case Maps Website*. Online at http://www.usecasemaps.org/index.shtml

[37] Usoro, A, *The Guide to the Preparation, Conduct and Assessment of MSc Information Technology (IT) and Advanced Computer Systems Development (ACSD) Projects in the School of Information and Communication Technologies (ICT) of the University of Paisley for Students, their Supervisors, and Moderators*. Online at http://cis.paisley.ac.uk/usor-ci0/Project_guide.htm

[38] *VISA Europe Website*. Online at http://www.visa.com

[39] *Voca Limited Website*. Online at http://www.voca.co.uk/home/index.php