

KINGSTON UNIVERSITY

Faculty of Science, Engineering and Computing

# Banking System

Arturas Bulavko

January 2018

## Table of Contents

|   |    |
|---|----|
| 1 Introduction .....                              | 3  |
| 2 High Level Analysis .....                       | 4  |
| 2.1 Introduction .....                            | 4  |
| 2.2 Textual Analysis.....                         | 4  |
| 2.3 Use Case Diagram .....                        | 6  |
| 2.4 Functional Requirements.....                  | 6  |
| 2.5 Non-Functional Requirements .....             | 7  |
| 2.6 Conclusion.....                               | 8  |
| 3 Software Architecture Review .....              | 9  |
| 3.1 Introduction .....                            | 9  |
| 3.2 Client-Server Architectural Style.....        | 9  |
| 3.3 Three-Tier & N-Tier Architectural Styles..... | 10 |
| 3.4 Layered Architectural Style .....             | 11 |
| 3.5 Object-Oriented Architectural Style .....     | 12 |
| 3.6 Component-Based Architectural Style.....      | 13 |
| 3.7 Service-Oriented Architectural Style.....     | 13 |
| 3.8 Two Selected Architectures .....              | 15 |
| 3.9 Conclusion.....                               | 15 |
| 4 Architecture Comparison .....                   | 16 |
| 4.1 Introduction .....                            | 16 |
| 4.2 Software Quality Factors Analysis.....        | 16 |
| 4.3 Architecture Recommendation.....              | 20 |
| 4.4 Conclusion.....                               | 21 |
| 5 Migration Strategy .....                        | 22 |
| 5.1 Introduction .....                            | 22 |
| 5.2 Review of Service Models .....                | 22 |
| 5.3 Review of Deployment Models.....              | 24 |
| 5.4 Migration Steps.....                          | 25 |
| 5.5 Conclusion.....                               | 27 |
| 6 High Level Design .....                         | 28 |
| 6.1 Introduction .....                            | 28 |
| 6.2 UML Class Diagram .....                       | 28 |
| 6.3 UML Component Diagram .....                   | 29 |

|   |    |
|---|----|
| 6.4 UML Sequence Diagrams .....             | 30 |
| 6.5 UML Activity Diagrams.....              | 31 |
| 6.6 Database Design.....                    | 33 |
| 6.7 Conclusion.....                         | 34 |
| 7 Review of Technologies .....              | 35 |
| 7.1 Introduction .....                      | 35 |
| 7.2 Technology Review .....                 | 35 |
| 7.3 Chosen Technology Justification.....    | 36 |
| 7.4 Conclusion.....                         | 36 |
| 8 Implementation .....                      | 37 |
| 8.1 Introduction .....                      | 37 |
| 8.2 Application Functionality .....         | 37 |
| 8.3 Challenges .....                        | 41 |
| 8.4 Conclusion.....                         | 41 |
| 9 Conclusion.....                           | 42 |
| References .....                            | 43 |
| Appendix A – Component Classes.....         | 45 |
| Appendix B – Service Examples .....         | 46 |
| Appendix C – Database Table Structure ..... | 47 |

## 1 Introduction

The ABC Banking Group focuses on delivering financial services to their clients. The central business activity is the transaction processing system which handles the money flow between the accounts. The bank also offers several web facilities aiding customers in account management. The company currently adopts a LAN-based system reachable over the web via a legacy software. Sommerville (2010, p.245) describes legacy software as outdated technology, lacking structure and documentation but is critical to the business process. Additionally, Crotty and Horrocks (2017) indicate that maintaining a legacy system is not cost effective. Therefore, to get the best return on investment, keep the business competitive and enable future expansions, the ABC Banking Group requires a new cloud service-oriented architecture to reflect on the recent advances in technology.

The report will begin by analysing the requirements requested by the client. Such strategy will help understand the problem and aid in avoiding specification ambiguity as well as give criteria enabling to judge the system towards the project completion. Having captured the requirements, the report will focus on thoroughly examining a range of available software architectural styles to explore the best usages in addition to role investigation of each style. The report will proceed onto selecting the two most suitable architectures for this project and comparing them in terms of software quality factors. This strategy will help evaluate both architectures with the factors that are most suitable based on the provided problem. The comprehensive comparison will justify the selected architecture and assist in proposing the migration strategy from a LAN-based system to a cloud environment. The cloud service and deployment models will also be reviewed and the appropriate solutions will be selected. The report will propose an architecture design suitable for the client by means of the UML diagrams. This report will continue the design phase by introducing several sequence and activity diagrams, as well as an Entity-Relationship (ER) diagram for the database. The report will then proceed onto reviewing the technology which could be used to implement the banking system. An appropriate technology will be selected and justified to ensure all requirements identified earlier are correctly implemented. Lastly, the report will briefly discuss the implementation phase using the selected technology by presenting a number of screenshots and discussing the challenges which arose through the phase.

## 2 High Level Analysis

### 2.1 Introduction

The analysis section will focus on investigating the requirements using the Object-Oriented tools, commonly known as OOA (Object-Oriented Analysis). This method concentrates on creating a model of the system, disregarding the implementation constraints. The emphasis of OOA is founding the objects, establishing their attributes, describing behaviour and the interaction process. Furthermore, a list of requirements will be created which will aid in system evaluation primarily during the testing phase. This will also eliminate requirement ambiguity at an early stage of the project.

### 2.2 Textual Analysis

The textual analysis is ideal in situations when the problem description is available at the start of the project. Fundamentally, the analysis splits the provided text into verbs and nouns. The verbs generally correspond to the methods of the application, whereas the nouns represent the objects and their attributes. OOA makes a clear distinction between objects and methods, however, certain nouns can be unreasonable and thus must be disregarded. Similarly, the verbs can be repeated and synonyms can be used to represent the same verb in a lengthy textual description, implying that whilst the analyst treats both words as same, they can mean two completely different ideas within the business logic. To overcome these issues, it is necessary to understand the business logic thoroughly, avoiding ambiguity which will help deliver the best solution to the problem.

The textual analysis demonstrated below was elicited from the brief. All nouns were highlighted in blue and all verbs were highlighted in red in order to emerge them from the overall description.

*"In order to remain competitive and be able to expand its business ABC Banking Group must update its services to reflect the recent advances in information and communication technology. This will require the design and implementation of an adaptable technology migration strategy. Currently, ABC Banking Group system is a LAN based, able to be reached over the web using legacy software. Thus, the Group needs a migration strategy from a LAN based system to Cloud based system, however such a migration requires the consideration not only of the underlying Cloud service oriented architecture, and its benefits, but also should reflect the main business activities of the Group.*

*At the core of the Group's business activities is its transaction processing system. The system is used to define accounts and transactions. Accounts refer to things like customers' bank accounts, while transactions are things like deposits and withdrawals which are essentially time-stamped records. Each account keeps track of the transactions that affect it. It also has a set of attributes such as customer's name, address, balance, overdraft, running totals (of deposits and withdrawals) computed from the transactions etc.*

*Once an account is set up, it is used by creating transactions and by querying the attributes of the account. Transactions can come from other systems, like direct debits, or from different branches and they can be created by program control or can be created by a user filling out an input screen. Customers can access their account and conduct transactions using their desktops, mobile phones etc.*

*Your task is to design new service based architecture of the system. It is up to you how to go along the task. However, you have to take into account the distributed nature of the problem and the possibility*

of *accessing* account details, on the *server*, using different *clients* and different *graphical user interfaces*. These *interfaces* are *programmed* so that they *communicate* with the *server*.

You define how an *account* *handles* *transactions* that are *posted* to it, one way of *handling* *transactions*, is by *putting* them in a *list* in order of their date. *Queries* can be from a simple *interface*, from *reports* such as *bank statements* or from *programs* that are *creating* *transactions*. All *interactions* with the *system* are achieved by creating *transactions* and *querying* *attributes*.

The *system* should be able to *perform* a number of *operations* including *creating* *account* for every *customer*, *holding* the *customer's* *name* and *address*, *allocating* a numeric *code* (account number) for every *customer*, *balance*, *cost* for *overdrafts*, *returning* the *statements* etc. The *system* also should be able to *add*, *delete* *customers* and *work out* the total number of *customers*.” (Khaddaj, 2017)

Having identified the verbs and nouns it is beneficial to collect them into one place. This strategy eliminates all the duplicate words and removes meaningless terms. *Table 1* shows the necessary classes, their attributes, methods to deliver business objectives and interactions between the classes.

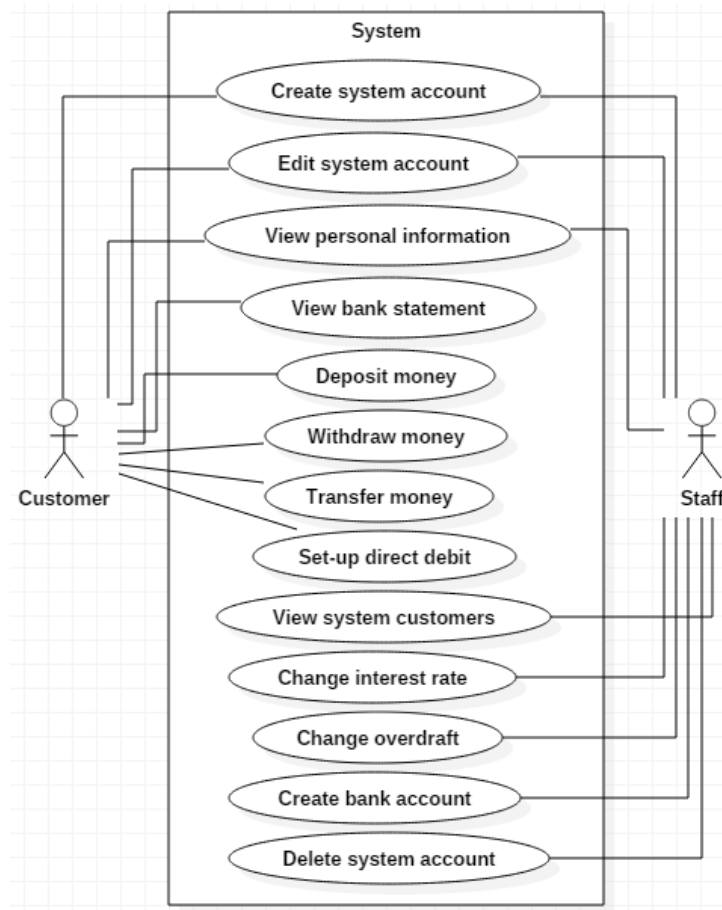
|  |
|--|
| CLASSES  |
| Menu, Account, SavingsAccount, CurrentAccount, Transaction, DirectDebit, ExternalTransaction, Customer, Individual, Corporate, ABCBankingGroup.  |
| ATTRIBUTES (all private)   |
| <b>Account</b> – runningTotals, openDate   <b>SavingsAccount</b> – interestRate   <b>CurrentAccount</b> – overdraftLimit   <b>Menu</b> – List <Transaction> transactionList, List <Account> customerAccountList, List <Customer> customerList   <b>Customer</b> – accountNumber, name, address, contactNumber, email, password   <b>Individual</b> – surname, gender, dateOfBirth   <b>Corporate</b> – companyType   <b>Transaction</b> – amount, timeStamp, sender, receiver   <b>DirectDebit</b> – paymentDate   <b>ExternalTransaction</b> – branchName, branchAddress, branchPostCode, branchCode. |
| METHODS (getters and setters inclusive)  |
| <b>Account</b> – showCustomerTransactions   <b>Menu</b> – inisitateTransaction, addCustomer, deleteCustomer, openAccount, closeAccount, editCustomerDetails, login, logout   <b>Transaction</b> – depositMoney, withdrawMoney   <b>Customer</b> – totalNumberOfCustomers   <b>Individual</b> – workOutAge.   |
| INHERITANCE  |
| <b>Transaction</b> – DirectDebit, ExternalTransaction   <b>Customer</b> – Individual, Corporate   <b>Account</b> – SavingsAccount, CurrentAccount.   |
| AGGREGATION  |
| Menu – Account, Transaction, Customer   ABCBankingGroup – Account.   |
| ASSOCIATION  |
| Account – Transaction   Menu – ABCBankingGroup.  |

**Table 1 – OO format of Textual Analysis**

*Table 1* demonstrates the final step of the textual analysis procedure. This stage mainly focuses on translating text into an unfinished Object-Oriented architecture. The term unfinished implies that several additional classes and methods will be added during the Design phase to accommodate the GUI (Graphical user Interface), storage and security aspects. At this stage, the textual analysis is considered to be done as the business logic was clearly captured within the scope and it is safe to proceed onto defining the system requirements.

## 2.3 Use Case Diagram

A use case diagram is regularly used to identify and graphically illustrate the goals of each user. It focuses on each system user, and explores the interaction steps with the system. The diagram is created using StarUML which is a software modelling tool widely used by the system analysts.



**Figure 1 – Use Case diagram for the System**

Figure 1 illustrates a use case diagram created for this project. It was discovered that the system will have two user groups, the customer and the staff. The customer will primarily use the system to manage their bank account and complete transactions. The staff will be an admin to the system, supervising and providing guidance to the customers. Both user types have specific functionality in the system, allowing them to complete their task as part of the user group. However, some functionality overlaps to both user types, implying that both users can access specific service within the system. This can be achieved via user privileges and sessions. Having identified all the user goals, it is necessary to capture the requirements in the next phase.

## 2.4 Functional Requirements

Functional requirements are used to represent the system behavior and the users' goals. In this project, the requirements are elicited from a provided textual description as well as use case diagram. In order to focus on the most important functionalities of the system, the requirements are prioritised using MoSCoW which is a part of DSDM Agile methodology. Such prioritization divides the requirements into four types of priorities, which are; Must, Should, Could and Won't have.

| ID | REQUIREMENT               | PRIORITISATION | USER GROUP      |
|----|---------------------------|----------------|-----------------|
| 1  | Create system account     | MUST           | Customer, Staff |
| 2  | Edit system account       | SHOULD         | Customer, Staff |
| 3  | View personal information | COULD          | Customer, Staff |
| 4  | Recover system account    | WON'T          | Customer, Staff |
| 5  | View bank statement       | MUST           | Customer        |
| 6  | Deposit money             | MUST           | Customer        |
| 7  | Withdraw money            | MUST           | Customer        |
| 8  | Transfer money            | MUST           | Customer        |
| 9  | Set-up direct debit       | MUST           | Customer        |
| 10 | View system customers     | SHOULD         | Staff           |
| 11 | Change interest rate      | MUST           | Staff           |
| 12 | Change overdraft          | MUST           | Staff           |
| 13 | Create bank account       | MUST           | Staff           |
| 14 | Delete system account     | COULD          | Staff           |

**Table 2 – List of Functional Requirements**

Table 2 shows several requirements which primarily belong to the customer user group as the project client mostly offers financial services to their clients. It was also decided to add staff user group who will deal with the administrative work. As seen from the table, all critical requirements have 'Must have' prioritization, meaning that they are crucial to the business logic and must be implemented regardless of the constraints. Some of the requirements are shared by the two user groups, indicating that both parties have access to a specific feature within the system. The requirement with ID 4 was classed as 'won't have' due to time constraint and will not be implemented. However, this shows that all system functionality was truly considered and perhaps the requirement will be implemented at a later revision of the system.

## 2.5 Non-Functional Requirements

Non-functional requirements typically describe the attributes of the system and are primary used to evaluate the quality of it. They are split into several areas and must provide a clear criterion against which the requirement can be measured. The requirements are elicited from the provided textual description which specifies the desired qualities of the system by the client.

| AREA         | REQUIREMENT   |
|--------------|---|
| Security     | 1. The user shall be authenticated before granting access.<br>2. All sensitive data held in database shall be encrypted.<br>3. Data stored within the database shall follow Data Protection Act 1998. |
| Efficiency   | 4. Latency shall be under 5 seconds across all internet connections.<br>5. The system shall be able to run smoothly with 1,000 concurrent users.  |
| Portability  | 6. The system shall be accessible from desktops and mobile phones.  |
| Availability | 7. The system shall be available 99.8% of the time during the year.<br>8. The system shall be restored within 2 hours after a crash.  |
| Reliability  | 9. The system shall be able to process 10,000 requests within 12 hours.<br>10. The copy of the database shall be kept on a separate server.<br>11. The system shall have a ROCOF of 0.00001.          |
| Usability    | 12. The system shall use consistent layout on all pages.<br>13. The system shall be accessible by people with disabilities.   |
| Scalability  | 14. New system functionality shall be added within current architecture/code.   |

**Table 3 – List of Non-Functional Requirements**



*Table 3* presents a list of non-functional requirements for this project. As the transactions are the core business activity, it is expected to have a high reliability, efficiency and availability of the system in order to provide constant services to the clients. Additionally, the banking group wishes to expand in the future, implying that the system must adopt scalability and reusability attributes to ease the system expansion and management. Lastly, the portability attribute is described by the need to make the system accessible from a range of devices available on the market.

## 2.6 Conclusion

Throughout this chapter, a textual analysis was carried and as a result, main characteristics of the system have been captured in an Object-Oriented fashion. Additionally, use case diagrams were created to illustrate each system requirement graphically. A list of functional requirements has been created along with the MoSCoW prioritisation. Additionally, the desired qualities of the system have been captured as Non-Functional requirements and will be used to assess the system towards the end of the project.

In order to improve the analysis phase, it would be beneficial to conduct several interviews with the ABC Banking Group to eliminate potential ambiguity, such as; can the account belong to one or many customers. However, as this phase is complete, it is necessary to research various architectural styles for this project. The next chapter will compare the advantages and disadvantages of each architectural style, selecting the most suitable two for this project.

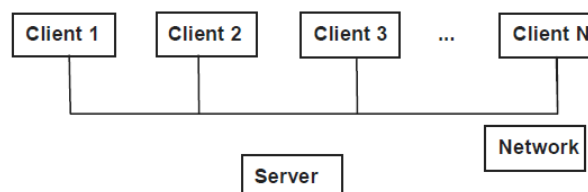
## 3 Software Architecture Review

### 3.1 Introduction

This chapter will examine numerous software architectural styles which can be utilised in the project. It will closely analyse the benefits and disadvantages of each style, providing through discussion of key elements in relation to the project. Towards the end of this chapter, two appropriate styles will be nominated along with a full decision justification.

### 3.2 Client-Server Architectural Style

The client-server architectural style is part of the distributed systems. It adopts a request-response cycle whereby the user interacts with a remote server via the client. This is also known as inter-process communication. The architecture shows a configuration between the client, communication middleware and the server which interact to form a fully functioning system. *Figure 2* illustrates an example of the client-server architectural style. The client can be software (web-browser, phone application) or hardware (phone, tablet) enabling the user making requests to the server. The server is a software which provides specific service as a response to the client's request. The server is able to serve numerous clients simultaneously, but each client does not know about the existence of another client. The communication middleware is any process that enables the client and the server communication, frequently associated with a network. A typical communication between the client and the server uses RPC (Remote Procedure Calls) or SQL (Sequential Query Language) statements which are lightweight and thus reduce the network traffic (Yadava and Singh, 2009, p.41).



**Figure 2 – Client-Server Architectural Style (Yadava and Singh, 2009, p.43)**

In order to successfully develop a client-server architecture, the following principles must be fulfilled: (Yadava and Singh, 2009, p.45)

- ✓ All components must be able to run on a multitude of hardware platforms.
- ✓ All components must support several operating systems and network protocols.
- ✓ All services offered within the network must be open and visible to the client.
- ✓ The processes must be evenly distributed between the client and the server.
- ✓ All principles must comply with the defined standards.

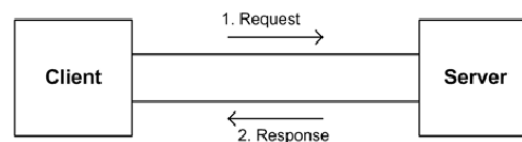
It is worth noting that the client and the server can run on the same machine and interact locally. Additionally, the client can access several servers for a specific service implying that it is not restricted to one server in the client-server architecture. The servers are usually built from two parts; a master program which accepts new requests, and slaves that handle the requests concurrently. This approach reduces the workload on the server, making it efficient in handling multiple requests.

The client-server architectural style is often used to provide access to a centralised database for the clients. An advantage of the client-server architectural style is absence of the need to transfer files. In

turn, the server responds via a query which has a positive effect on the network traffic. Other benefits include distribution of services, centralised storage and easy scalability. This allows introducing new clients and servers, providing new system features or altering the workload. It is also relatively simple to change the server without affecting the system as a whole, mainly due to independence. However, being dependant on the network, the client-server style may have varying performance and be subject to attacks. Additionally, network failure can cause disruptions to the service.

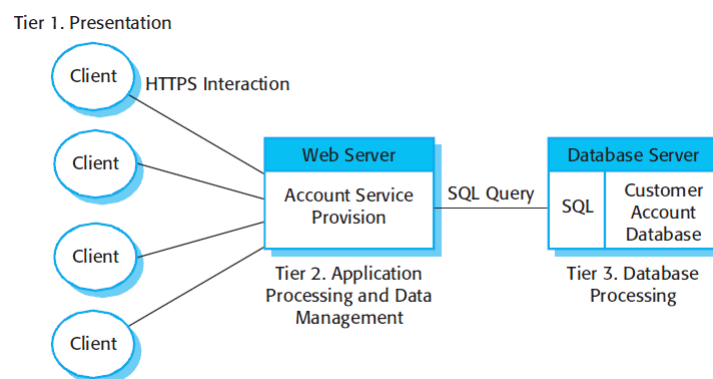
### 3.3 Three-Tier & N-Tier Architectural Styles

The three-tier and n-tier are a part of the client-server architectural style whereby the GUI, business logic and storage functions are separated. In fact, the client-server architecture is based upon the trivial two-tier architecture, shown in *Figure 3*. In two-tier paradigm, it is relatively easy to establish secure connection by adopting encryption between the client and the server. However, the two-tier architectural style only performs well with a small number of clients which concurrently access the server, suggesting that scalability is a major issue.



**Figure 3 – Two-Tier Architectural Style (Vogel et al., 2011, p.196)**

To overcome the scalability issue of the two-tier, the three-tier architectural style introduces a new middleware tier. *Figure 4* demonstrates an example of a three-tier architectural style for an internet banking system. The client (tier one) is a browser through which the user interacts with the system by sending requests. The web server (tier two) processes the requests and manages the data, allowing the user to deposit or transfer money. The database server (tier three) holds all the necessary records, allowing the web server to query specific information. By introducing an intermediate tier, the exchange process can be optimised by using low-level protocols which improve the performance of the system. However, due to more points of interaction in a communication between the client and the server when compared to two-tier, the performance worsens. Three-tier allows to make the application modularised, enabling easy maintenance post deployment. This also has a positive effect on scalability as the new components can be added effortlessly due to distributed deployment of the functions. It is also relatively easy making changes to the system because all tiers are independent and rely on interfaces for communication.

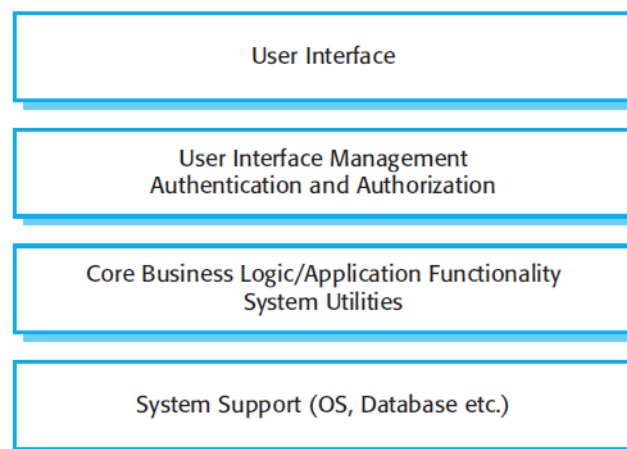


**Figure 4 – Three-Tier Architectural Style (Sommerville, 2010, p.494)**

The N-tier architectural style is also known as multitier architecture. It is a part of client-server style which separates the tasks across different processors. The three-tier style can be expanded to add servers to process specific functionality which will ultimately turn it into n-tier architecture. The n-tier styles are commonly utilised to achieve high dependability requirements as well as being able to process large number of requests simultaneously. Due to complexity of n-tier, it becomes harder to maintain the system. Sommerville (2010, p.495) states that the n-tier architecture is typically used in unstable environments and where the data is integrated from several clients. This is true because new tiers can be rapidly added to accommodate the instability, avoiding major changes to the entire system. Furthermore, by integrating all data, the database is likely to have correct and conversant information, conforming to Data Protection Act 1998.

### 3.4 Layered Architectural Style

The layered architectural style groups related functionality and stacks it into vertical layers. As demonstrated in *Figure 5*, each layer is responsible for a specific function of the system. The bottom layer provides support to the database and the operating system. The layer above is concerned with the business logic by processing the information. The next layer formats the data, providing user authentication and authorization. The top layer presents the data to the user in a human-readable format using a GUI (Graphical user Interface). The amount of layers depends on the system and the number can vary depending on the complexity of the software and the user requirements. Typically, the layer above acts a service provider and the layer below acts as a service consumer. The components inside each layer can interact freely, however, to allow a loosely coupled communication between each layer, an interface must be used.

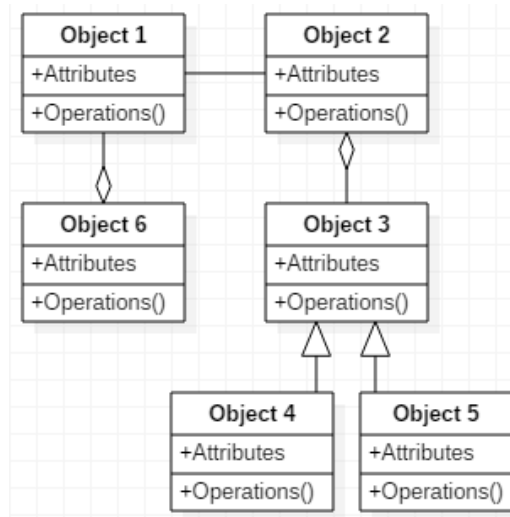


**Figure 5 – Layered Architectural Style (Sommerville, 2010, p.158)**

The primary advantage of the layered approach is easy maintenance because due to localisation, each layer can be replaced or altered without having to rewrite code for the entire system. This is achieved through interfaces. However, Sommerville (2010, p.158) debates that separating each layer correctly is problematic, implying that the top-level layer may sometimes communicate directly with the bottom-level layer rather than hierarchically. Additionally, due to having several layers, the application can have low performance as processing each layer can be time consuming. The layered approach is commonly used in large organisations where a specific team develops one layer of the system. It is then necessary to merge all layers into sole application revealing problems with the interfaces. This can be seen as another disadvantage as the merging process can be costly and time consuming.

### 3.5 Object-Oriented Architectural Style

The object-oriented (OO) architectural style is based on object decomposition, used to “describe system as a collection of classes” (Oussalah, 2014, p.8). It focuses on decomposing complex systems into interacting objects which represent data structures in the memory. *Figure 6* shows an example of OO architectural style consisting of objects, their attributes and methods for managing the data.



**Figure 6 – Object-Oriented Architectural Style**

The relationships are fundamental to OO paradigm which are used to show the interaction process between the objects. Vogel et al. (2011, pp.144-145) outline the following relationships:

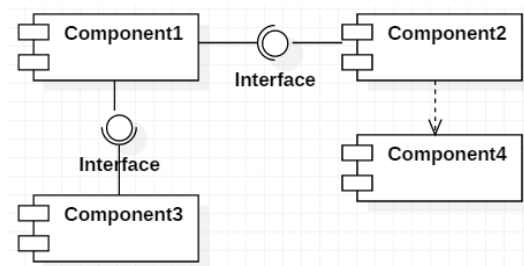
- ✓ Association – Current object uses an object which is an instance of current or another class.
- ✓ Aggregation – Current object is part of another object.
- ✓ Inheritance – The child class acquires all the attributes and methods of the parent class.
- ✓ Interface & Abstract Class – A class can use an interface to show all offered methods, but not their implementation.

Another important aspect of OO is polymorphism, allowing the object, variable and operation to be in multiple forms. It enables to declare several implementations of the same interface, making OO paradigm extensible and scalable. There are two types of polymorphism; compile and runtime. In compile time polymorphism (static binding), the compiler resolves the call, enabling prompt execution. In runtime polymorphism (dynamic binding), the call is resolved during the runtime by JVM, slowing down the execution but introducing more flexibility.

The OO architectural style offers several advantages, such as modularisation due to abstractions. Furthermore, encapsulation allows creating high-level objects and achieve better information hiding (Vogel et al., 2011, p.147). The OO paradigm is easy to maintain, highly modularised and extensible which makes it an ideal candidate for the current project. Additionally, the produced code is easily testable, ensuring integrations and system changes do not introduce inconsistencies and defects. The disadvantage of OO architectural style is the need to fully understand the problem in order to develop an elegant solution conforming to all requirements. Vogel et al. (2011, p.147) tells that OO does not fully support reusability due to classes being too specific and less abstract. This is a major disadvantage of the object-oriented approach because more time is required to create systems.

### 3.6 Component-Based Architectural Style

The component-based architectural style was created to overcome the OO limitation in which the objects cannot be widely reused due to them being too detailed and specific. The idea of component is to be a reusable, self-contained module which is independent of the system as a whole. Szyperski et al. (2002, p.3) describe components as unit of composition, suggesting that the components are designed to collaborate with each other to form new system. A component is a loosely coupled entity defined by the interface which provides a specific functionality to the system. Components are connected through the interfaces in which a clear separation between the implementation and the interface exists. Component-based architecture is established by reusing existing components or developing new software components in order to build complex systems. As illustrated in *Figure 7*, components can be treated as building blocks which are put together in order to form a complex system. An interface can be seen as a bridge which enables communication between the components.



**Figure 7 – Component-based architectural style**

A single component is commonly deployed using OO paradigm, however, procedural model can also be used. A component can be used without making any changes to its implementation by supplying the necessary parameters. Jyotsna (2014) states that out of many component infrastructure technologies, only three have become standardised, which are; CORBA, COM/DCOM and EJB.

The primary advantage of components is their reuse, allowing development of complex systems much faster, as oppose to writing code from the beginning (Sommerville, 2010, p.453). It is also easy to change the component's implementation without affecting the system due to modularisation. Vogel et al. (2011, p.150) point out that the component-based architecture separates concerns much better than the OO paradigm, implying that the technical concerns can be reused in different systems due to encapsulation. Additionally, each component can be adjusted to provide more functionality. All components are standardised and documented to allow potential users to fully assess the component ensuring it meets their need before using it.

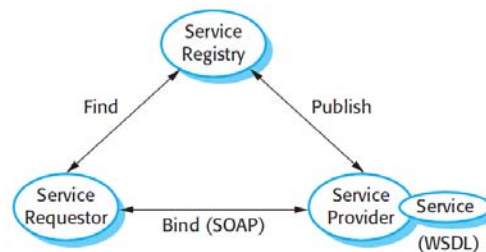
However, the disadvantage of component-based approach is maintenance cost (Jyotsna, 2014). Since the component is reused in a number of different systems it is necessary that it conforms to several requirements concurrently. Similarly, the ambiguous requirements make it hard to develop components which can be reused by other systems. Lastly, to enable reuse, each component must be independent and loosely coupled which is costly and time consuming to achieve.

### 3.7 Service-Oriented Architectural Style

Service-oriented architectural (SOA) style is a way of deploying distributed systems using independent components. It is a collection of components which communicate with each other in order to provide functionality to a complex application. SOA can be seen as a solution enabling software

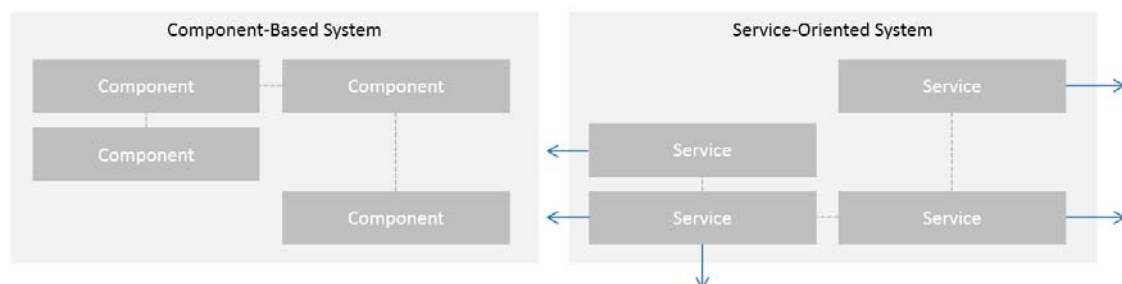
communication using XML-based protocols, such as SOAP (Simple Object Access Protocol) and WSDL (Web Service Definition Language) (Sommerville, 2010, p.509). SOAP is used to support structural information exchange between the services. WSDL on the other hand, is a standard for defining an interface which describes the functionality of a service. Moreover, REST (Representational State Transfer) is a simplified architectural style of SOAP and WSDL, allowing to implement web services.

A service is software and hardware independent, stand-alone, scalable, reusable, self-contained, distributed, loosely coupled and standardised solution delivering a particular functionality (Vogel et al., 2011, p.206). A service is made-up of service implementation and service interface, and can consist of other services. *Figure 8* demonstrates the conception of SOA. The service providers develop a service along with full documentation and interface specification using WSDL. The service is published in the service registry to make it discoverable. The service requester locates the service specification on the service registry and finds the service provider. Once the provider is found, the communication between the application and the service can be established, usually using SOAP.



**Figure 8 – Service-oriented architectural style (Sommerville, 2010, p.510)**

SOA and component-based architecture may sound familiar due to both supporting abstraction in order to achieve loose coupling and both providing support for automated testing. Service-oriented architecture can be seen as an improvement to components. *Figure 9* demonstrates both architectural styles which outlines a clear difference between the two. The component-based architecture is concerned with modelling and building the complex system by dividing it into components. SOA organises services and defines various standards for communication between them, independently of the software and hardware requirements. The individual services used in the current system can be reused by other systems in order to provide new functionality.



**Figure 9 – Component-based vs Service-oriented systems**

A service is usually a black-box to the service requestors, suggesting that the internal processing is hidden and only the output is displayed. This can be advantageous because loopholes introducing malware are much harder to locate. Another benefit of SOA is independence from products and language/technology ensuring the service can be used by a range of developers without having to

worry about the integration. Similarly, companies can rapidly adapt to the market by reusing, changing or updating certain services instead of rewriting an entire system. Services follow open standards, allowing them to be easily located and reused.

SOA has several drawbacks, such as high development costs. Additionally, a number of security issues arise in a loosely coupled environment. The services communicate asynchronously, suggesting that SOA is best to avoid in real-time systems. By using services from service providers, the system becomes reliant on other people to maintain and provide a fully dependable service. With paid services, it may be costly to fully test a service prior to integration. Similarly, the service can be discontinued by the vendor, suggesting that a new component must be either found or developed. Though, this is not an issue for large companies where components are developed and tested internally by different teams.

### 3.8 Two Selected Architectures

Having reviewed each architectural style it became apparent that the two most suitable styles for this project are Service-Oriented and Object-Oriented. These two styles match majority of the client requirements as well as being widely-used approaches today, indicating that the deployed system will be futureproof. Being so popular presently, the maintenance of the system will be carried-out easily by the most software engineers.

The client-server architectural style was disregarded due to coupling, the client and the server are tightly coupled, meaning that scalability and reusability are at a disadvantage. The OO and SOA are loosely coupled which fully matches the user requirements, since the client wanted to expand their services in the future. Besides, the client-server style performs poorly with a large amount of users (Vogel et al., 2011, p.196). This is a major issue for a banking system as it is expected that large volume of people will use the system concurrently.

Three-Tier architecture was neglected due to poor performance when compared to OO and SOA. The N-Tier architectures was also abandoned due to complex maintenance. Both architectural styles are harder to implement than SOA and OO, suggesting that the project development would last much longer. The layered architectural style was abandoned due to weak performance. As mentioned earlier, the layered approach requires more time to process data when compared to SOA and OO. The component-based architectural style will be taken forward and implemented as a SOA.

The OO architectural style was chosen because it is easily scalable due to polymorphism, reusable due to inheritance and fully testable using most available techniques. Similarly, SOA is scalable, reusable, loosely coupled and independent making it an ideal competitor to OO. Both architectural styles bring more advantages to the projects than disadvantages, making them ideal solutions to the problem.

### 3.9 Conclusion

This section has focused on reviewing six presently used architectural styles. A description of each paradigm was provided, along with benefits and drawbacks of their application. Due to time constraints, it was not possible to explore more architectural styles such as; repository, pipe & filter and peer to peer (p2p). Towards the end, two architectural styles were selected as potential solutions to the problems defined in this project. The two selected styles will be compared in the next section.



## 4 Architecture Comparison

### 4.1 Introduction

This section will produce a comprehensive comparison between the earlier selected SOA and OO architectural styles. Several software quality factors which are critical to this project will be used to assess both architectural styles in order to select an appropriate one. The most suitable style will be adopted throughout the development lifecycle to deliver a solution to the client.

### 4.2 Software Quality Factors Analysis

Software quality assesses the quality of the design of the system as well as the quality of conformance to that design. An architectural design has a direct impact on the Quality of Service (QoS), for example; coupling affects maintainability and reusability. This suggests that a correctly selected architecture can have a positive impact on QoS during the early stages of the development lifecycle. Zhu (2005, p.33) states that errors made during the design of the architecture can be costly and maybe impossible to rectify at a later stage.

There are countless software quality factors available presently. These factors are reflected in the ISO (International Organisation for Standardisation) and IEEE (Institute of Electrical Engineers) standards which group and prioritise them in terms of their effectiveness towards QoS. Schulmeyer (2007, pp.67-68) hints that ISO/IEC 9126 series is best applicable in measuring software quality due to resting on three components; internal quality, external quality and quality in use. This is achieved by relying on six quality factors, namely: functionality, reliability, usability, efficiency, maintainability and portability which are further divided into twenty-seven sub-attributes allowing to accurately measure the quality (Siau et al., 2011, p.116). These taxonomic models are great for defining quality but are lacking in measuring and predicting it. Furthermore, they are standardised for all projects with emphasis on specific requirements, however, in order to develop a quality software, it is necessary to assess the problem and define software quality factors specific to the project (Zhu, 2005, pp.39-43).

Having familiarised with the software quality factors offered by the above standards and fully understood the client requirements, numerous quality factors have been devised for this project. These factors will be used to assess the two selected architectural styles and ultimately select the best one for this project. The chosen software quality factors along with the reasoning are outlined below.

- ✓ **Correctness** – The group will be mainly dealing with customers; therefore, it is vital to implement user requirements correctly. Otherwise the system will not be used.
- ✓ **Maintainability** – The system is likely to be maintained by the ABC Banking Group, suggesting that the system must accommodate future changes by other developers/programmers.
- ✓ **Efficiency** – The organisation's key business activity is transaction management which suggests that the system throughput must be high and use as little resources as possible.
- ✓ **Scalability** – ABC Banking Group wishes to expand in the future, implying that the system must be easily scalable to add new features without having to change the entire architecture.
- ✓ **Reliability** – Since it is a banking system, each request must result in a correct output from the system. Additionally, the system must automatically withstand and recover from failures, informing the banking group of the occurred faults.
- ✓ **Reusability** – The banking group wishes to remain competitive in the market. By reusing existing components, new services will be developed quicker, rivalling the competitors.

- ✓ **Portability** – By expanding the baking group's services in the future, the current system should easily adapt to new technologies and conform to the latest standards. This will also ensure that the current system is easily accessible from a range of devices, as the client requested.

The software quality factors have been chosen and fully justified. It is now vital to elaborate and state what each factor means in order to remove ambiguity and correctly compare the two styles.

#### 4.2.1 Correctness

Correctness examines whether a user requirement was implemented correctly. This is a critical to the current system because if the user requirements are implemented incorrectly, the system will not be used. It can be easy writing a function which will complete something without faults, but writing a function which will perform what the user desires can be problematic. This is due to ambiguity during analysis phase and lack of problem understanding. An architectural style has a direct impact on correctness because various methods can be used to validate and verify the product.

Both architectural styles adopt techniques allowing to validate and verify the software through a range of testing procedures. OO supports automated testing tools, such as unit testing which is used to ensure that the function has been implemented correctly. Similarly, SOA also supports automated testing, however, due to the nature of the architecture, it is harder to implement than in OO. Dustdar and Haslinger (2004) argue that there is lack of automated testing for distributed systems which would enhance the correctness of the systems developed using SOA. Unit testing in SOA is a much lengthier process due to the need to test both, the service component and the individual services in isolation (Chatterjee, 2008). Integration testing can be more straightforward in OO as the system does not need to be simulated. This is an issue in SOA because service proxies must be established in order to perform integration testing to check whether a use case was fulfilled, (Chatterjee, 2008).

Service in SOA is typically a black-box to the consumer, suggesting that the code cannot be tested via inspections and walk-throughs. Such approach can negatively impact on the performance of the code. OO architectural style on the other hand, is typically white-box, allowing other members of the company to inspect and review the code, ensuring it is fully optimised. The lack of broad aspect of testing tools in SOA means that the software may lack correctness. The use of inheritance in OO style may reduce the amount of testing due to reusing previously tested software. SOA is largely similar since the service reuse is one of the central advantages of the architectural style. However, by reusing services from third-parties it is necessary to test both; the service consumers and service providers to ensure that the service is delivered correctly and as specified. This is time-consuming and costly as a result. Furthermore, these types of tests typically test for absence of faults and not whether the function correctly implements the user's goal.

OO architectural style intensely supports the development process to ensure requirements are correctly captured, converted into an appropriate design and implemented appropriately. SOA also supports the development process, however, it is less user centric and more emphasis is made on the service itself. The service is made abstract which is great for use, but can negatively impact the correctness of the function.

As a result of this comparison, it is clear that OO architectural style achieves correctness to a higher degree by adopting a range of techniques. SOA is lacking in some areas due to being relatively new technology, resulting in absence of necessary tools to validate and verify the developed system.

#### 4.2.2 Maintainability

Maintainability is concerned with the process of maintaining the system. A good software system should be simplistic and well-documented, allowing external engineers to manage it post deployment. Zhu (2005, p. 36) describes two types of maintainability operations to be modification for correction and modification for environment change adaptation. The first type examines the complexity of resolving the system bugs. Similarly, the second type inspects the difficulty in making changes to the system by introducing new functionality. Both types of maintainability are important factors in the current project because faults are inevitable on release and the ABC Banking Group wishes to expand the system in the future by adding more functionality.

Sommerville (2010, p. 109) states that in order to achieve the best maintainability the system must have high cohesion and loose coupling. Coupling looks at dependencies between the components in an architecture, whereas cohesion looks at the dependencies within the components. Cohesion measures the degree to which a component is self-contained (Sommerville, 2010, p.123). OO architectural style and SOA are both modularised and are loosely coupled. This allows to instantly locate the necessary component and make changes to it. The use of encapsulation in OO enables the developers changing the implementation without affecting the interface because they are separated. SOA operates in a similar fashion, whereby the interface is separate from the implementation. Due to service granularity in SOA, developers can also change the implementation effortlessly. Both architectural styles aim to develop simplistic and well-documented systems which are maintainable. This results in both architectural styles having similar maintainability properties.

#### 4.2.3 Efficiency

Efficiency characterises the responsiveness of the system. It looks at the time required to process a sequence of events, which are typically split into three parts (Zhu, 2005, pp.33-34). First, the time taken for the communication between the components to be established. Second, the processing time of the component execution. This takes into account the parallel execution time spent on mutual exclusion and synchronisation. Third, the time needed to perform the business logic. It looks into the data structures and algorithms used in order to provide functionality. All three parts are influenced by the architectural style.

The component communication differs in both architectural styles. SOA is reliant on the network since the data is passed using well-defined formats. Services in SOA communicate using the service contract which is based on an agreement (Koskela, 2007). This demonstrates that component communication is highly reliant on the network speed. OO communicates by invoking methods, dependency injection or through an adaptor. Sadly, there are currently no comparison tests to estimate which method is more efficient. However, efficiency can be looked at from a different perspective.

Since services are self-contained, hardware and software independent, it is possible to use procedural programming in components that are processing larger loads. Procedural programming follows hardware architecture, making it significantly efficient in processing data when compared to OO. This significant advantage of SOA has a huge benefit towards efficiency.

#### 4.2.4 Scalability

Scalability is partitioned into two areas which are influenced by the architectural style. The first area checks whether the system can handle an increased load without affecting the overall performance. The second area is concerned with enlarging the system to add further functionality.

Shim et al. (2008) state that SOA was introduced to accommodate prompt requirement changes. This is achieved by creating reusable, self-contained, independent and loosely coupled services which provide a specific function to the system. As a result of this, components can be easily reused to develop new system functionality easier. Vogel et al. (2011, p.147) debates that reusability in OO architectures is less likely to be achieved due to the objects being too specific. Furthermore, Voelz and Goeb (2010) debate that the level of abstraction in SOA is much higher than it is in OO, allowing to easily reconfigure services without having to change major parts of the system. Since there are service vendors, it is much faster to scale the system by utilising their services. Generally, the underlying principles of SOA make it more scalable when compared to OO which is more dependent on classes.

#### 4.2.5 Reliability

Reliability specifies the probability of a system failure in a given period of time or vice-versa. These failures may occur as a result of poor architecture selection. Zhu (2005, p.35) argues that a good architecture decomposes complex problems into tiny components which are more testable and are easier to understand. This in turn decreases the chance of component failure. Sommerville (2010, p.323) suggests POFOD (Probability Of Failure On Demand) and ROCOF (Rate Of Occurrences Of Failures) metrics to measure the reliability. POFOD is commonly used in systems where a failure can cause serious problem. ROCOF is best used in systems with regular demands for a service, which makes it ideal for this project. However, Zhu (2005, p.35) also suggests that reliability can be measured as “mean time between failures”. This suggests that an architecture can be measured in terms of repair time required to eliminate the fault.

Since reliability is dependent on time spent locating the fault, it is logical that the architectural style with better maintenance will be more reliable. However, having compared OO and SOA, the maintenance was considered equivalent. Both architectural styles decompose the problems into tiny components which are then implemented. By looking at reliability from the metrics point of view, it becomes apparent that estimating POFOD and ROCOF in OO paradigm is much easier. This is because the amount of automated tools available for OO is significantly higher than in SOA. Various tests can be run in OO to assess the reliability of the function. This process would have to be done manually in SOA, negatively impacting the time. Furthermore, the reliability in SOA could be affected by the network connection. The service provider can potentially offer a very reliable service, but due to communication issues the service can be unreliable. As a result of this, OO is considered to have much better reliability when compared with SOA.

#### 4.2.6 Reusability

Reusability describes the ease to reuse software components in other systems. An architectural style plays an important role in reusability as it describes system decomposition and relationships (Zhu, 2005, pp.36-37). By making each module independent, complex systems can be developed in a short period of time by reusing earlier developed components.

One of the primary advantages of SOA is reusability. Services can be widely reused as they are not tied to the project (Koskela et al., 2007). They are more abstract than the classes in OO paradigm. OO can “only partly supports reusability” (Vogel et al., 2011, p.147) because the classes are too detailed and specific. The encapsulation used in OO allows code reuse by grouping objects into classes, enabling inheritance (Koskela et al., 2007). While this is introducing reuse, the SOA style is more generalised and proposes more reusable functionality. Both architectural styles are considered to have loose coupling, enabling reuse to a certain degree. SOA offers greater reusability than OO, allowing systems adopting SOA quickly adapt to the market and environment changes.

#### 4.2.7 Portability

Portability examines the ease of transporting system from one platform to another. According to Zhu (2015, pp.35-36) a good architecture groups together the facilities dependant on environment in order to allow easy migration to a new platform. This strategy allows to rewrite specific components to match them to a new platform, rather than rewriting an entire system. Zhu (2015, p.36) also states that the best portability is achieved by making entire system independent from the environment.

SOA is based on widely recognised standards which ensures it is highly portable (Voelz and Goeb, 2010). By conforming to the standards, SOA system is less likely to require major changes to the code. Sommerville (2010, p.509) states that SOA are platform-independent meaning that systems adopting SOA can run in many environments. Similarly, Voelz and Goeb (2010) reveal that SOA can quickly and easily adapt to changes in the environment due to platform independence. In contrast, OO portability depends on the programming language, making it less portable than SOA. Furthermore, OO depends on the hardware architecture, implying that it cannot adapt to the changes in the environment rapidly. It is clear that SOA is much more portable in comparison with OO.

### 4.3 Architecture Recommendation

As a result of the above analysis, a comparison matrix is made to conclude each architectural style. A binary format is used to estimate the appropriate architectural style for this project. 1 is given to the architectural style where it outperforms another style, which will get 0. If both architectural styles are equally great in achieving a certain software quality factor, they will both get 1.

| Factor          | OBJECT-ORIENTED | SERVICE-ORIENTED |
|-----------------|-----------------|------------------|
| Correctness     | 1               | 0                |
| Maintainability | 1               | 1                |
| Efficiency      | 0               | 1                |
| Scalability     | 0               | 1                |
| Reliability     | 1               | 0                |
| Reusability     | 0               | 1                |
| Portability     | 0               | 1                |
| <b>TOTAL</b>    | <b>3</b>        | <b>5</b>         |

**Table 4 – Software Quality Factor Comparison Matrix**

As demonstrated in *Table 4*, the service-oriented architectural style has scored two more points than the object-oriented architectural style. After a close analysis of the two styles, the service-oriented architectural style was selected for this project. It was chosen because it allows to use countless technologies within the scope. The architecture allows to use hardware and software independent services in order to increase efficiency. The portability and reusability of SOA are much better in

comparison with OO architectural style, which was proven in section 4.2. A major limitation of SOA is correctness, suggesting that the system design must be correct and possibly signed-off by the client to ensure all ambiguities are correctly understood and requirements have been correctly captured.

#### **4.4 Conclusion**

This section focused on comparing object-oriented and service-oriented architectural styles in relation to the quality factors of this project. The quality factors were elicited from the brief. As a result of deep comparison, it was revealed that SOA is more applicable for this project than OO.

It would however be beneficial to run real tests in order to compare the results, as oppose to comparing the two styles in terms of their features. The lack of research which compared the two styles directly in terms of software quality attributes made it hard to state real facts rather than reasonable assumptions based on research. As this chapter is done and the appropriate architectural style is selected, a migration strategy must be developed in the next chapter.

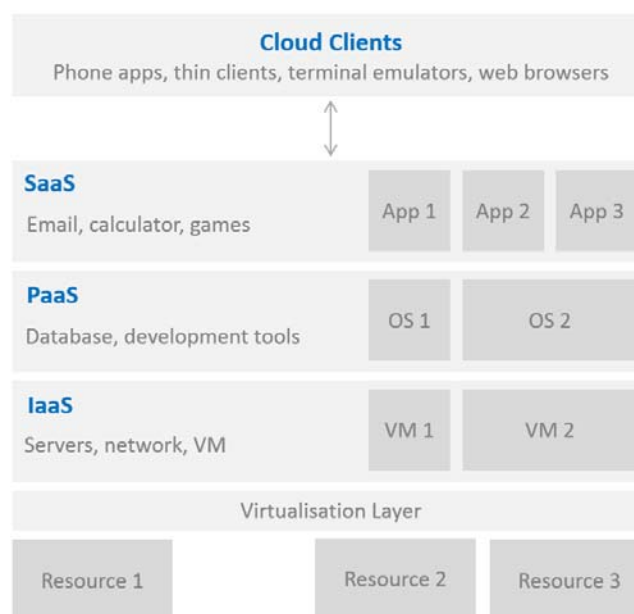
## 5 Migration Strategy

### 5.1 Introduction

This chapter will review cloud computing by examining various service and deployment models on offer. An appropriate model and deployment model will be selecting, allowing to map SOA onto it. The chapter will conclude by offering migrations steps from a LAN-based system to the Cloud.

### 5.2 Review of Service Models

Cloud computing adopts the principles of SOA by partitioning the problem into a number of services. It provides on-demand resources as services which are accessible over the network. Cloud services are highly scalable and are easily accessible. Services are typically split into three models, shown in *Figure 10*. This allows to correctly match the demands of the customers.



**Figure 10 – Layers in the Cloud Services (Adapted from Khaddaj, 2012)**

The first layer is Software as a Service (SaaS), followed by Platform as a Service (PaaS) and lastly, Infrastructure as a Service (IaaS). Each layer will be analysed next.

#### 5.2.1 Software as a Service (SaaS)

SaaS allows on-demand, pay-per-use usage of platform-independent applications available for multiple end-users. Essentially, the vendor hosts a system on the cloud which is leased to the customer. Zaigham (2013, p.50) states that the developers have access to the APIs. Though, this service is mostly applicable to end-users with no programming skills who wish to perform a specific task. All of the resources are managed by the service provider, implying that the service provider has access to all the data. SaaS model is universally accessible from all platforms, requiring minimum planning. The software is supported by the service provider, ensuring it is dependable. Due to absence of licensing costs and the software is not purchased, this model is less costly (Zaigham, 2013, p.50). However, the integration with other systems and software is practically impossible due to limited portability. Moreover, the performance is affected by the network, suggesting that issues may arise for customers wishing to use SaaS through slow network connections.



### 5.2.2 Platform as a Service (PaaS)

PaaS essentially provides platform and tools, allowing customers to create systems without having to install the software and worry about the underlying infrastructure (Zaigham, 2013, p.49). This service offers operating system, database, web-server and various APIs. The client is responsible for managing the application resources and the data, and the vendor handles all of the other resources. Such model is mostly applicable to developers wishing to use resources which are automatically scaled by the service provider to accommodate the system loads.

The main advantage of PaaS model is rapid development of scalable systems without the need to think about the infrastructure and software licensing implications. PaaS allows customers to run their own software on the platform, which overcomes the limitation of PaaS. Security is one of the primary disadvantages of PaaS because the customer does not have control over data processing and virtual machine. Hill et al. (2013, p.14) describes lack of portability resulting in vendor lock-in as another drawback of PaaS. Due to reliance on the offered tools by the service provider, the application is less likely to be portable to another provider for concurrent running as the tools used might not be supplied by another provider. As a result of this, the customer must remain with the current vendor.

### 5.2.3 Infrastructure as a Service (IaaS)

IaaS offers a web-based access to the computing architecture, such as storage and computing power to the customers. Being virtualised, it allows multiple users accessing the resources. The vendors provide physical hardware and are responsible for managing data storage, virtualisation, servers and networking resources. The back-end is owned and managed by the service providers. The cost of the model depends on the usage, rather than fixed monthly/yearly payments and allow scaling the services to meet the demands of the business. This reduces the overall company expenditure.

IaaS model offers enhanced scalability due to dynamic workloads and increased flexibility, enabling prompt adaptation to the market. Zaigham (2013, p.49) notes that IaaS improves “mobility, stability, agility, availability and elasticity” of the business, making this service appropriate for this project. Hill et al. (2013, p.14) argue that IaaS is perfect for companies wishing to control the entire platform and the software stack, allowing them to request more infrastructure promptly. However, the client is responsible for managing the virtual machine, this is not an issue in SaaS and PaaS. Additionally, network and service delays are possible due to reliance on the network connection.

### 5.2.4 Service Model Selection

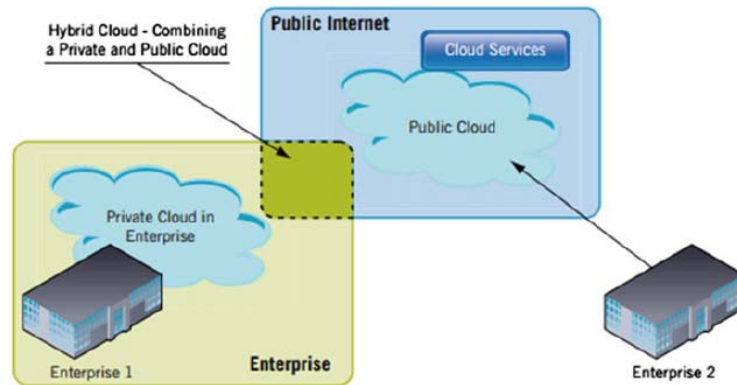
The most suitable service for this project would be IaaS as it offers full flexibility to the developers. Furthermore, since the vendor provides hardware and is responsible for managing main resources, this model will enable the ABC Banking Group to focus on their business case, rather than organise fundamental things. IaaS is also great due to the payment model because it scales with resource usage which can always be changed to accommodate the needs. However, since security is an issue in SaaS, the newly developed system must implement security issues to limit the threats.

PaaS is not very suitable for this project as it is highly reliant on the service provider, offering limited flexibility. It is also possible to lock-in with one vendor for the lifetime of the system due to portability issues. The SaaS model was disregarded because an appropriate system will be developed for this project, ensuring it conforms to the client requirements. As such, there is no need to use generic software and slowly adapt it towards the client requirements.



### 5.3 Review of Deployment Models

The cloud deployment models are used to represent the different environments of the cloud. The models are highly reliant on the client requirements. Each type of the deployment model offers certain services with varying management and security features.



**Figure 11 – Cloud Deployment Models (Zaigham, 2013, p.50)**

Figure 11 demonstrates the cloud models which are; public, community, private and hybrid. Each of the deployment models will be discussed next.

#### 5.3.1 Private Cloud

A private cloud is hosted and managed by a private organisation or a third party (Hill et al., 2013, p.11). The infrastructure is organised to accommodate the business requirements of a specific company. Due to being private, only permitted people are allowed to access the data and applications (Zaigham, 2013, p.51). As a result, private clouds are very secure and allow full control of the system. Consequently, maintenance cost of private cloud is relatively expensive. Private cloud offers flexibility and greater control as the infrastructure can be adapted to match the system requirements.

#### 5.3.2 Public Cloud

Public cloud is a shared infrastructure made available to the consumers by the vendors. It offers computing and networking resources which can be provisioned on demand. The back-end is owned and managed by the provider (Hill et al., 2013, p.11). This model allows the users to develop and deploy systems using little financial investments (Zaigham, 2013, p.51). Public model is relatively cheap to use, supporting all service models. Generally, the vendor provides reliable and dependable services which are location independent. Due to publicity, the security of this model suffers. Besides, the model is not fully flexible, implying that it cannot achieve specific QoS requirements. Public cloud has limited privacy, meaning the consumer does not know how and where the data is stored (Goyal, 2014). This means that people can get unauthorised access to the data.

#### 5.3.3 Community Cloud

Community cloud provide resources which can be accessed by other consumers. This is also known as “multi-tenant environment” (Zaigham, 2013, p.51). The community cloud is often owned and managed by a member of the association. This model is mainly applicable to competing companies, but due to shared resources it is not wise. As a result, community cloud is commonly used by non-profit organisations. The community cloud is considered to have weak security. It also costs more than the public cloud (Goyal, 2014).

### 5.3.4 Hybrid Cloud

A hybrid cloud arises by composing two or more cloud models together, frequently the public and the private. By partitioning services onto different cloud models it is possible to overcome various limitations of a particular model to achieve specific system characteristic. Zaigham (2013, p.51) states that the sensitive data is often stored internally with a public backup to achieve high reliability. As a result of this, the pros and cons are the same as individual model. It is also not clear how the encryption keys are managed when private and public clouds are merged (Goyal, 2014).

### 5.3.5 Deployment Model Selection

In order to select a suitable deployment model for this project it is necessary to compare how each model implements the quality requirements demanded by the client. The community cloud model is disregarded due to poor security and sharing of resources with competitors. *Table 5* demonstrates the assessment undertaken to compare public and private clouds. The hybrid cloud was not compared because the two can be combined to become hybrid model. The quality attributes were captured from non-functional requirements and section 4.2 which looked at software quality factors.

| Quality Requirements | PUBLIC CLOUD               | PRIVATE CLOUD                                      |
|----------------------|----------------------------|--|
| Security             |                            | Limited access to the system                       |
| Efficiency           |                            | Can match specific hardware                        |
| Portability          | Independent resources      |  |
| Availability         | Guaranteed by vendor       |  |
| Reliability          | Guaranteed by vendor       |  |
| Usability            | Unaffected                 | Unaffected   |
| Scalability          | Scalable resources         | Scalable resources                                 |
| Correctness          |                            | More adaptable to the business                     |
| Maintainability      | Maintained by the provider |  |
| Reusability          |                            | More components related to the business activities |
| TOTAL                | 6                          | 6  |

**Table 5 – QoS and Non-Functional Requirement Comparison Matrix**

From *Table 5* it is clear that both, the private and the public cloud models implement certain quality requirements better than the other. Since both models have scored an equal number, it is decided to use Hybrid model for this project. This will allow to use both cloud models simultaneously to balance the system out in terms of quality requirements. The use of hybrid model will enable the system to achieve majority of the requested quality features. Service Level Agreement (SLA) can be utilised to ensure the public cloud provider always delivers quality services to the consumer. According to Goyal (2014) the hybrid cloud reduces capital expenses if private model is used in conjunction with public model. This will help save money as developing SOA architecture is costly, compared to the other architectural styles.

## 5.4 Migration Steps

The migration process is a hard and long process which relocates a system. Therefore, it must be well-planned and structured process to ensure as little inconsistencies arise as possible. Chihi et al. (2016) note that for an effective migration, a team must be formed which will manage and monitor the project, providing all documentation. This helps deploy a more consistent system as same people will undertake the migration process from start to end. There are a number of different companies

offering migration services to a cloud, however, this report will demonstrate a more general approach. Zaigham (2013, pp.254-259) proposes the following steps in order to migrate successfully to a cloud:

1. Evaluate – analyse the business case and identifying the requirements.
2. Select – chose an applicable service and deployment model for the architecture.
3. Migrate – perform the migration, ensuring new faults are not introduced by running tests.
4. Optimise – fine-tune the system, ensuring it can meet the demand when needed.
5. Operate – use a range of available tools to monitor the performance of the system.

This report has fully covered the evaluation phase. All user goals have been captured and a number of functional and non-functional requirements have been developed. The functional requirements will be used to develop all necessary system functionality, whereas, the non-functional requirements will dictate the quality attributes of the system. The non-functional requirements will be used to assess QoS in order to select the most suitable deployment and service model for this project.

The selection step was also covered in this report. By reviewing a range of architectural styles in terms of their benefits in relation to the non-functional requirements, SOA was chosen as an ideal candidate. Furthermore, by assessing the QoS, IaaS service and hybrid deployment models were chosen for this project. SOA and cloud complement each other, allowing them to work together efficiently and map effortlessly. This allows to map SOA directly onto the cloud as the hardware can reflect the architecture in order to deliver services. They are both based on services and distributed software. With an exception that cloud is virtualised. SOA and cloud are extremely scalable and deliver peak performance when compared to alternatives. Due to service distribution, the reusability of services and resources allows both to be highly maintainable.

Having done the evaluation and selection stages, a migration can be initiated. Zaigham (2013, p.258) notes that prior to migration, an application profiling must be carried-out. It is used to assess the current system in terms of usage patterns and data flow for a period of two weeks. Such statistics enable will aid in measuring the efficiency of the new system and help create tests to ensure system correctness. This step must be done on the current LAN-based system to obtain usage statistics. Once the application profiling is done, a migration process can begin onto IaaS hybrid deployment model. Hill et al. (2013, p.105) note that IaaS model allows easy migration of deployed systems onto the cloud hardware, minimising the changes required to the system post migration. The migration is followed by running several tests to ensure the system was correctly migrated and no faults were introduced.

As part of the migration, the client must consider a number of resources which are not provided by the vendor. These are; operating system, programming languages, web servers and application in the IaaS model (Hill et al., 2013, p.105). Since the physical hardware, such as; CPU, networking, storage and memory size is determined and provided by the vendor there is no need to estimate these resources. Furthermore, the provided resources are easily scalable to accommodate different loads. The vendor must also provide APIs to support database and networking infrastructure for this project. Currently, the cloud service providers offer various packages to their clients which differ in resource quantity. By knowing exactly which resources and their quantity are needed for the system, it is possible to find a best deal to reduce the cost. However, Antonopoulos and Gillam (2010, p.364) suggest that inexpensive resources can negatively impact on the system due to low liability of the vendor during failures. As an entry point into cloud, the current project can utilise the medium pricing package of most service providers to ensure there are enough resources for the system being

developed. Such strategy will allow to deploy the system and based on optimisation step, scale the resources. It is also relatively inexpensive when compared to other packaged available on the market, but provides all the necessary tools and services for this project.

There is no need to wrap the current legacy system because a brand new solution will be developed using SOA as part of this project. However, if that was needed, Zhang et al. (2010) propose three strategies which could be adapted. The first is black-box approach which checks the legacy interfaces and maps them to SOA. The second is business logic approach which transforms the system based on critical functions. Lastly, the grey-box approach combines both previous approaches whereby the interfaces are combined with the business logic. Due to higher scalability, the last approach is commonly used in wrapping legacy systems.

The optimisation step will ensure that SOA runs efficiently on a cloud. It will guarantee that the system can “grow and shrink automatically” (Zaigham, 2013, p.258) to accommodate the system load. The operation phase will enable to monitor the deployed system using various tools to ensure it functions as intended and delivers dependable service.

Due to complexity of the migration, Zaigham (2013, pp.259-260) proposes the following policies to ensure that the process is done correctly:

- ✓ Planning policy – Specifies the migration order and connection process between the systems.
- ✓ Candidate qualification policy – Qualitative and quantitative criteria to assess whether the server matches the required cloud environment.
- ✓ Sizing policy – Mapping existing workloads onto cloud resources to keep same performance.
- ✓ Placement policy – Defines the component deployment.
- ✓ Exception policy – Actions required to be taken for components not suitable for the cloud.

These policies would be highly beneficial in this project because they reduce the risks of encountering costly errors during migration. To conclude, a step-by-step migration guide was created for this project. The first two phases are covered in this report, ensuring the problem is fully analysed and correct requirements are captured.

## 5.5 Conclusion

This chapter has successfully reviewed and selected the cloud service and deployment models. The earlier defined QoS and non-functional requirements have helped in assessing the models. This has allowed to select appropriate models for this project in order to fully match the requirements. Lastly, the migration steps were proposed to aid in system migration to the cloud.

## 6 High Level Design

### 6.1 Introduction

This phase will focus on designing a service-oriented architecture for this project. The analysis done earlier will be mapped onto UML (Unified Modelling Language) diagrams to represent the architecture graphically. Furthermore, the created component diagram will illustrate how the system will deliver all necessary services from a range of distributed components.

### 6.2 UML Class Diagram

The class diagram shows a static structure of the classes, their methods and attributes, along with the relationships between each class. Such diagram aids in converting requirements into a data model which can be used to develop a system conforming to the client specification.

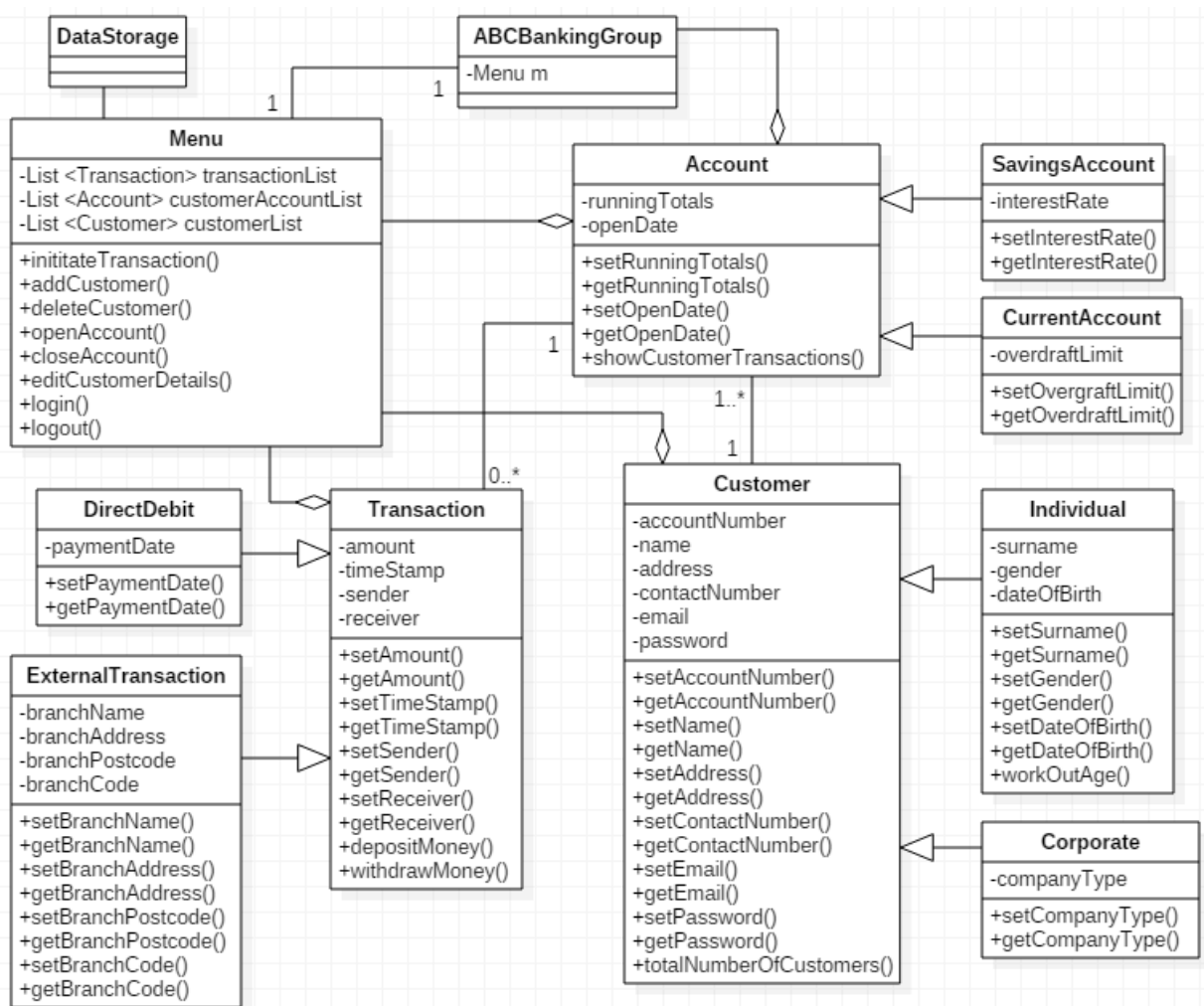


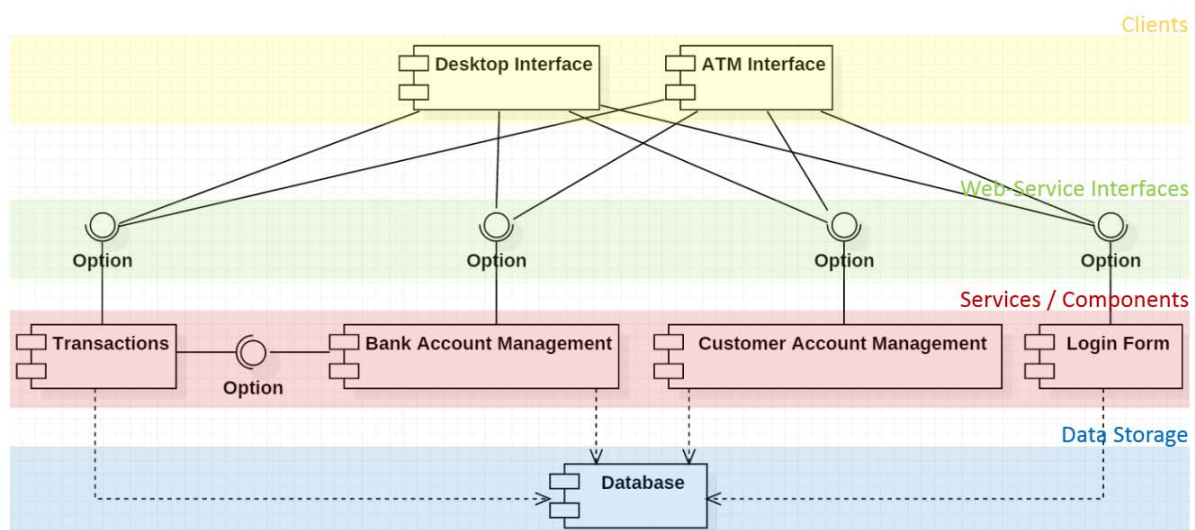
Figure 12 - Project Class Diagram

Figure 12 demonstrates a class diagram which was created after analysis phase for this project. It shows all necessary classes, attributes and methods required to fulfill the client's needs. A new class was added after the textual analysis in order to elegantly adopt a service-oriented approach. The class is 'DataStorage' which acts as a database holding all related data to the business.

The class diagram in *Figure 12* shows a number of lines specific to UML which fundamentally display the relationship between each class. The straight line demonstrates an association, the line with an arrow shows inheritance and a line with a diamond illustrates aggregation. Since all of the attributes are private, it is necessary to add getter and setter methods to enable access to the attribute from another class. Majority of the methods were captured during the analysis, however, the Menu class has new method additions which will primarily enable the user to operate the system. These methods are believed to be displayed on a GUI as various options to the system users. Once the class diagram is finalized, a component diagram must be made next to illustrate the distributed deployment.

### 6.3 UML Component Diagram

A component diagram is used to show the complex system distribution and relationships between each module. It is used in Component-Based Development (CBD) to demonstrate systems with SOA. The components are split into logical and physical classes. The logical component is independent of the physical and represents what each component does, whereas the physical component represents software and hardware technology, describing how the component does its job. The components communicate by providing or requiring an interface in order to deliver a service.



**Figure 13 - Project Component Diagram**

The structure of the system for this project is illustrated in *Figure 13*. It shows how certain functionality was grouped together to form components and distributed in order to provide services in a service-oriented paradigm. By adopting this modularised approach, the system will adhere to all non-functional requirements identified earlier. Each component will operate independently of the other modules, making the system scalable and modifiable/adaptable since a specific module can be changed effortlessly without having to re-write all other system features. It was decided to group Customer and Account classes into one component, Customer Account management. This was done to keep the system simplistic. The two processing components will have a direct relationship with the database to store, retrieve and update records for each customer.

## 6.4 UML Sequence Diagrams

Sequence diagram graphically demonstrates the order of object interaction in order to deliver a specific function. It helps the developers understand the flow of critical activities and enables to test the system against all scenarios. The sequence diagrams are typically created using UML (Unified Modeling Language) which allows all stakeholders to understand the system activities prior to the implementation, introducing adjustments if necessary at an early stage. A number of sequence diagrams were created for this project, which will be presented next.

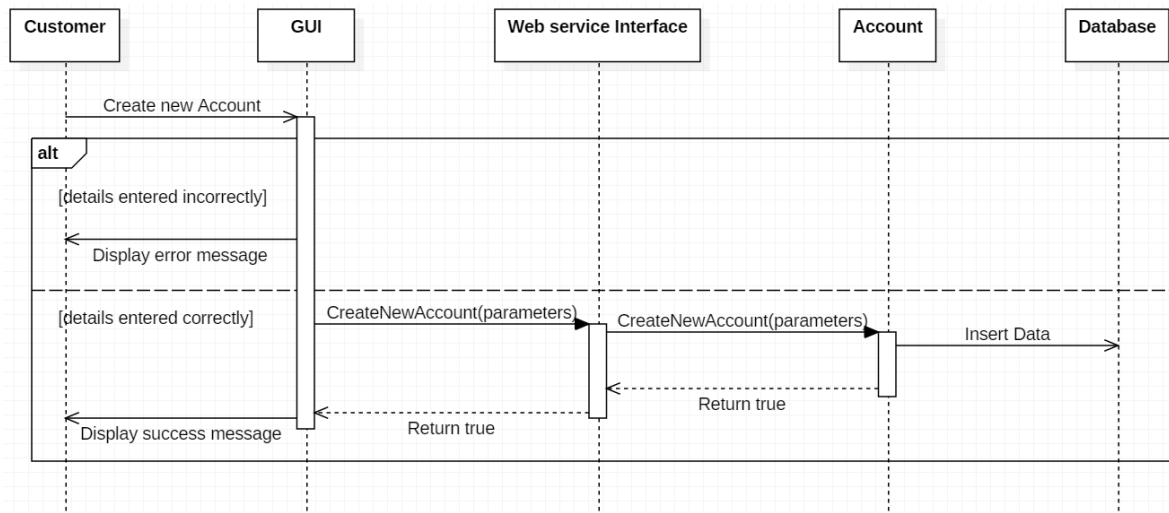


Figure 14 – Open Bank Account Sequence Diagram

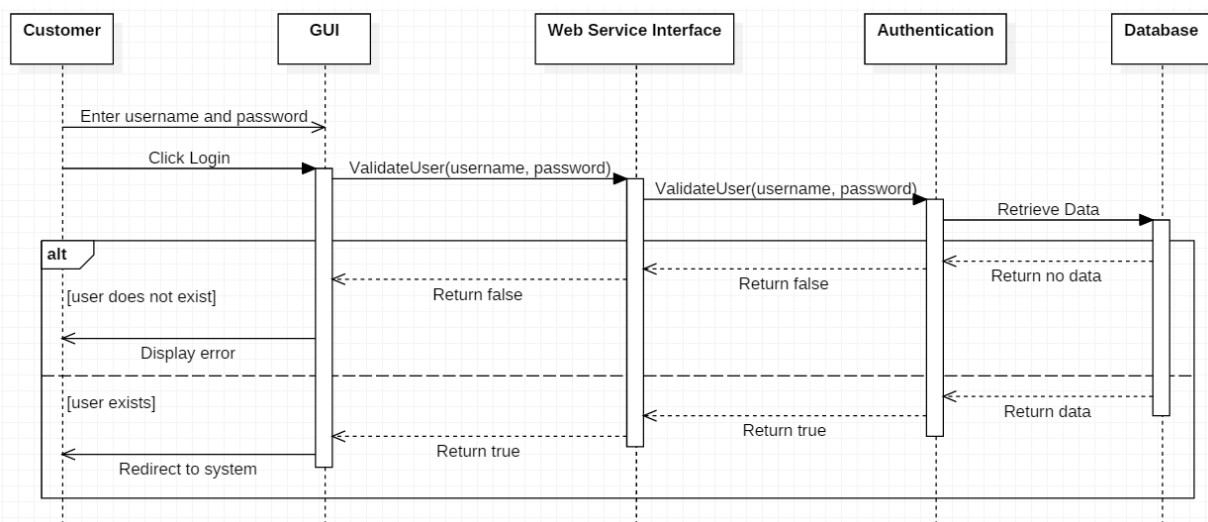
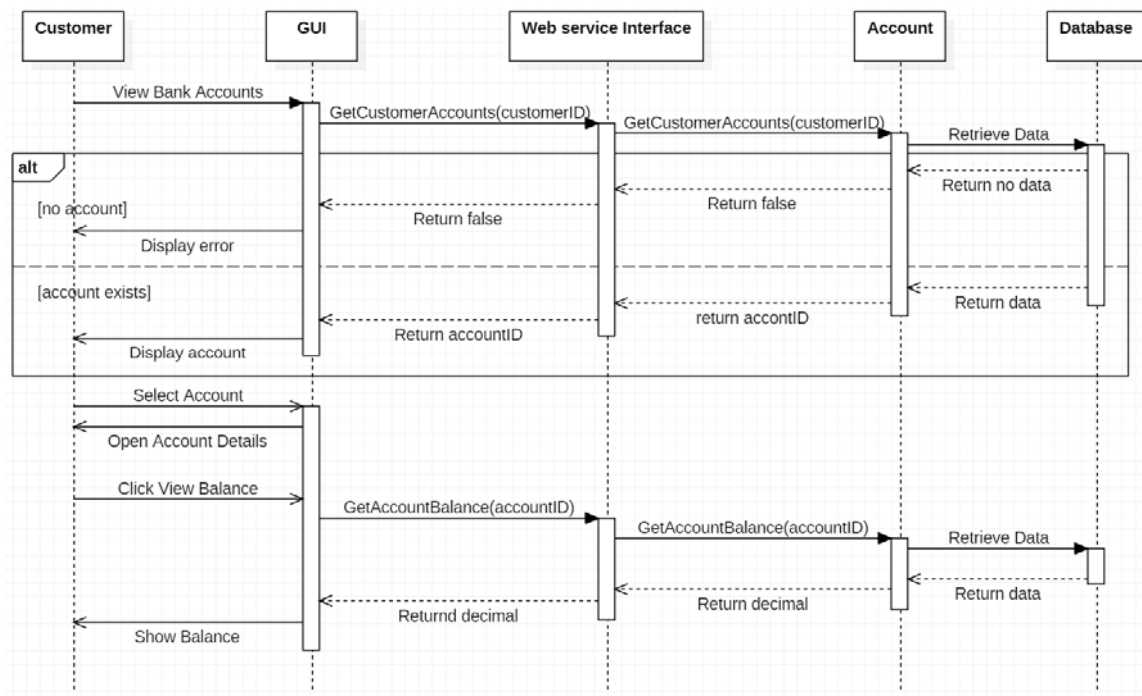


Figure 15 – Login to the System Sequence Diagram



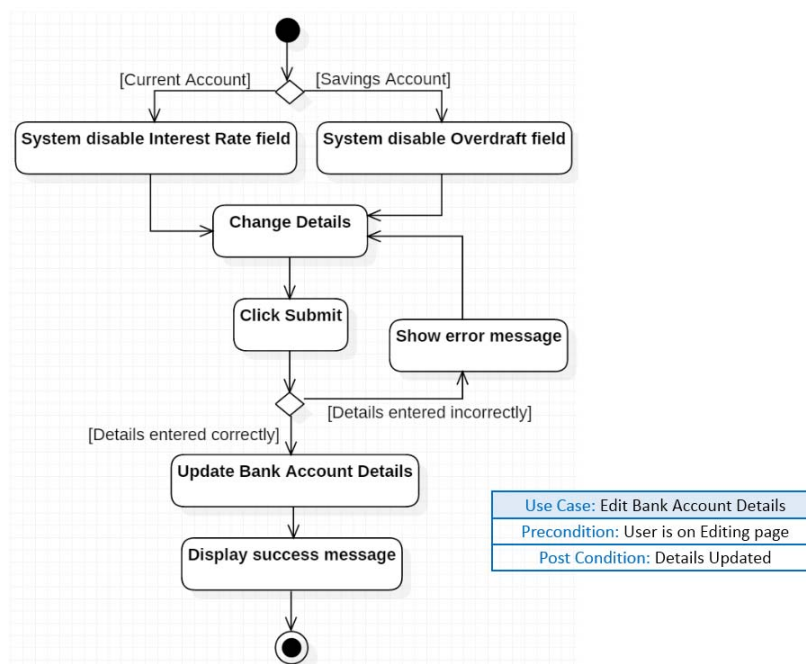


**Figure 16 – View Account Balance Sequence Diagram**

Figures 14 – 16 demonstrate the sequence diagrams created for this project. Each diagram shows the interaction steps between each component, allowing the user performing an operation.

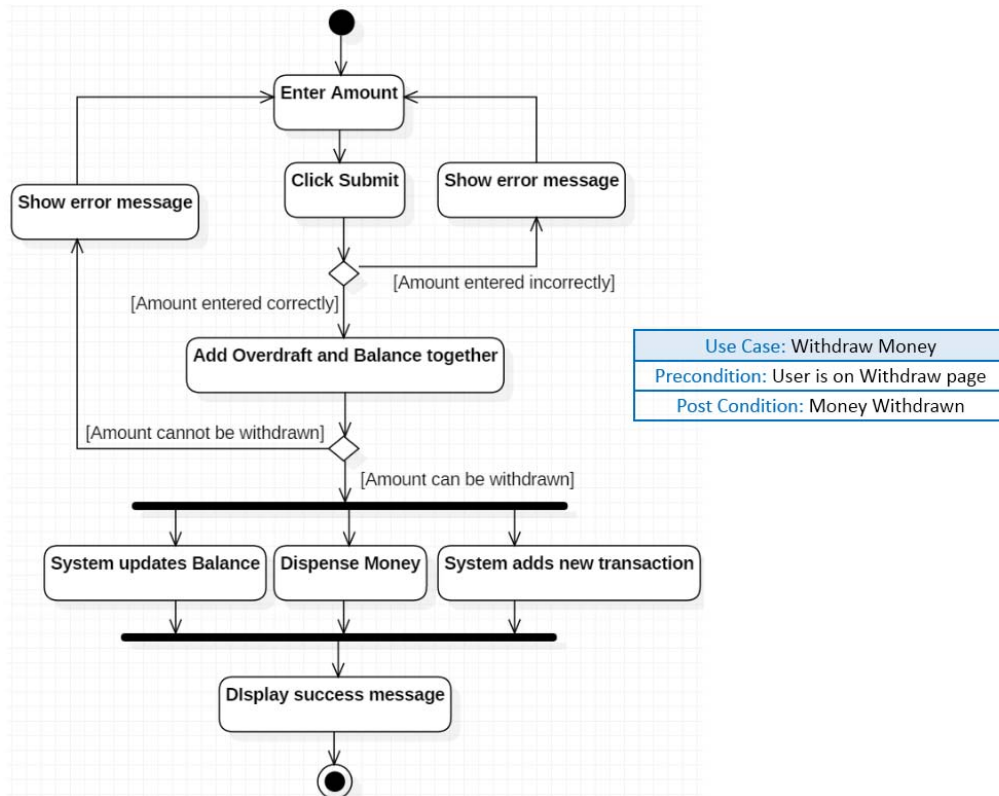
## 6.5 UML Activity Diagrams

Activity diagrams represent the activity workflow, showing process decisions and alternate paths. Such diagrams aid in writing the code because all actions within a function have a logical graphical algorithm which can be easily translated into code. Activity diagrams are also modeled using UML.

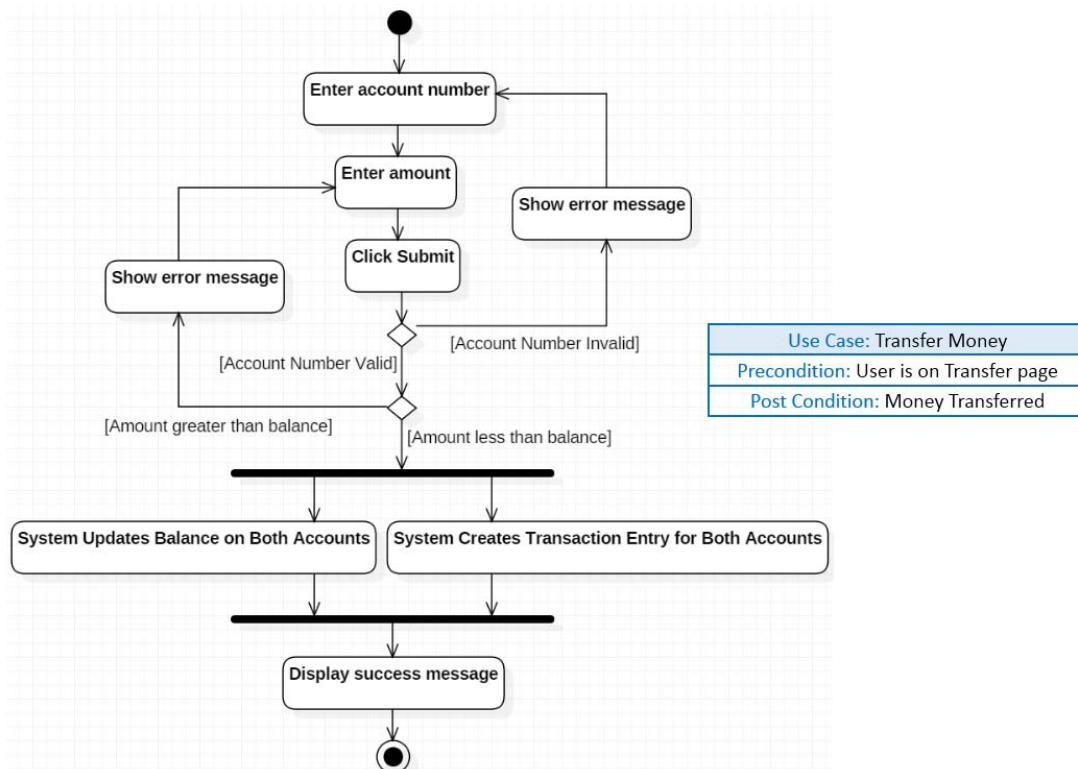


**Figure 17 – Edit Bank Account Details Activity Diagram**





**Figure 18 – Withdraw Money Activity Diagram**



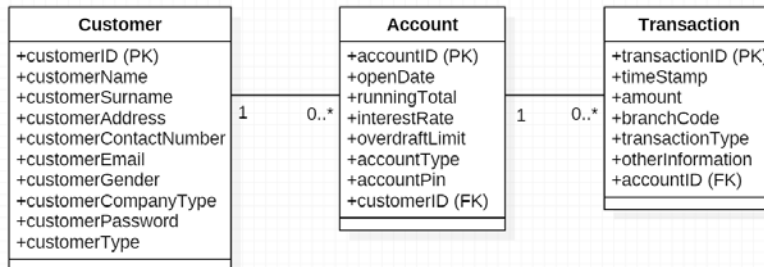
**Figure 19 – Transfer Money Activity Diagram**

Figures 17 – 19 illustrate the activity diagrams created for this project. Each diagram shows the flow of the critical system activities.

## 6.6 Database Design

### 6.6.1 Entity Relationship Diagram

The database will be used to store all necessary data in a centralised place. An Entity Relationship (ER) diagram is used to plan the database design by showing entities and their relationships. An efficient database aids in query optimization, reducing the latency of each user action.



**Figure 20 – ER Diagram**

Figure 20 shows a proposed database design in 3<sup>rd</sup> Normal Form which can be utilised in this project. Each table will be used to store necessary information about an entity and have a relationship with another table. The Customer may have many Bank Accounts, but an Account must only belong to one Customer. Similarly, the Account may have many Transactions, but a transaction must only belong to one Account. Such modelling fully matches the requirements identified earlier.

### 6.6.2 Data Dictionary

Data Dictionary compliments ER diagram by outlining the structure of each database field.

| Relation Name | Attribute Name        | Data Type | Length | PK/FK | Allow Null | Default   |
|---------------|-----------------------|-----------|--------|-------|------------|-----------|
| Customer      | customerID            | INT       |        | PK    |            |           |
|               | customerName          | VARCHAR   | 30     |       |            |           |
|               | customerSurname       | VARCHAR   | 30     |       | Yes        |           |
|               | customerAddress       | VARCHAR   | 30     |       |            |           |
|               | customerContactNumber | VARCHAR   | 11     |       |            |           |
|               | customerEmail         | VARCHAR   | 40     |       |            |           |
|               | customerGender        | VARCHAR   | 6      |       | Yes        |           |
|               | customerCompanyType   | VARCHAR   | 10     |       | Yes        |           |
|               | customerPassword      | VARCHAR   | 20     |       |            |           |
|               | customerType          | VARCHAR   | 1      |       |            |           |
| Account       | accountID             | INT       |        | PK    |            |           |
|               | openDate              | DATETIME  |        |       |            | getDate() |
|               | runningTotal          | DECIMAL   | 15,2   |       |            |           |
|               | interestRate          | DECIMAL   | 15,2   |       | Yes        |           |
|               | overdraftLimit        | DECIMAL   | 15,2   |       | Yes        |           |
|               | accountType           | VARCHAR   | 1      |       |            |           |
|               | accountPin            | VARCHAR   | 4      |       |            |           |
|               | customerID            | INT       |        | FK    |            |           |
| Transaction   | transactionID         | INT       |        | PK    |            | getDate() |
|               | timestamp             | DATETIME  |        |       |            |           |
|               | amount                | DECIMAL   | 15,2   |       |            |           |
|               | branchCode            | VARCHAR   | 20     |       | Yes        |           |
|               | transactionType       | VARCHAR   | 2      |       |            |           |
|               | otherInformation      | VARCHAR   | 20     |       | Yes        |           |
|               | accountID             | INT       |        | FK    |            |           |

**Table 6 – Data Dictionary**

*Table 6* presents a data dictionary for the system's database. The *customerContactNumber* field will be stored as a varchar because int data type removes the 0 at the front of the phone number. As a result, numbers starting 0788 would be stored as 788 in the database, which is incorrect. It was decided to store all money figures as decimal(15,2) in order to preserve the accuracy. This means that the field is capable of storing 15 scales with the precision set as 2. Float type is not applicable to money storage because it performs rounding of the figure, whereas decimal keeps the number accurate. The timestamps will use default inbuilt *getDate()* function which automatically collects the date and time of query insertion and adds the value into the field. This ensures that the timestamps are always stored in the correct format for SQL. Lastly, the Primary Keys (PK) will be automatically generated for each new record, ensuring all values are unique.

## 6.7 Conclusion

This chapter focused on designing a software-oriented architecture for the ABC Banking Group by using unified modelling language. A logical class diagram was developed to match the client's functional and non-functional requirements. Furthermore, a component diagram was created to illustrate how the distributed components will communicate to form a complex system using SOA. Presently, the architecture of the system is fully designed, ready for technology selection and implementation during the next software development phase.

This section has continued analysing the system and designing an appropriate solution. Several sequence and activity diagrams were created to show interaction and workflow of the user goals. An ER diagram was presented in 3<sup>rd</sup> Normal Form which can be used to develop a suitable database. Lastly, a data dictionary containing the proposed structure of each database table was developed. Having completed the design phase of the project, the report will proceed onto reviewing the technologies which can be used to implement the system.

## 7 Review of Technologies

### 7.1 Introduction

This chapter will examine several technologies which can be used to implement the distributed system using SOA paradigm. Appropriate tools will be selected for the implementation of the system in order to conform to the earlier defined requirements and the selected architecture.

### 7.2 Technology Review

Having previously selected SOA, it is necessary to select the appropriate technologies enabling to implement the system. Based on the findings, the selected technology must allow easy scaling and maintenance in the future, leading to the selection of OO language. Presently, there are countless languages which offer such features, but this report will review the main three.

C# is a type-safe OO language which can be used to build client-server applications, web services and distributed components using ASP.NET framework. The most commonly used IDE is Visual Studio which supports 36 different programming languages as well as countless plugins, allowing to create various enterprise-level systems in a single place. By adopting CLR (Common Runtime Language) the framework allows to code and reuse components developed in the other programming languages (Microsoft, 2018). However, the code developed in ASP.NET is managed, which is slower when compared to the native code. ASP.NET allows the development of SOAP and RESTful web-services.

The extensive built-in tools in ASP.NET enable the easy development of native windows or web-based front-ends for different platforms and complex, distributed back-ends to process the business logic. Furthermore, by reusing existing components, Rapid Application Development (RAD) can be adopted to quickly develop the product and automatic monitoring prevents a range of issues such as; infinite loops and memory leaks (Hasan, 2017). ASP.NET framework also strongly supports security by providing authentication, authorization, confidentiality and integrity aspects. However, it is primarily focused on Windows OS, implying that the applications cannot be developed and maintained on another OS, suggesting vendor lock-in.

Java is a widely-used cross-platform OO language allowing to develop secure systems, web services and client interfaces. Being widely used, there are several IDEs which support Java, namely Eclipse, NetBeans and IntelliJ IDEA. The Java code is compiled into bytecode, allowing it to run on any platform using JVM (Java Virtual Machine). Java was designed to support secure distributed systems enabling to write web services and components using JAX-WS and JAX-RS. JAX-WS is a Java API for XML-based protocols, such as SOAP, allowing the development of message-oriented and RPS-oriented web services (Oracle, 2010). The primary advantage of JAX-WS is platform independence and the ability to access web-services that are not running on Java platform. JAX-RS on the other hand provides a Java API for RESTful services using specific annotations (Oracle, 2013). However, the GUI support in Java is lacking, meaning that the front-end developed in Java Swing or JavaFX has a negative impact on UX and looks different to the native applications.

Ruby is a general-purpose multi-paradigm language which supports OO. The use of Ruby on Rails framework allows the development of MVC applications by providing default structure for web services. It enables the use of web standards to facilitate data transfer among the components and provides the tools to effortlessly develop user interfaces. Ruby on Rails can be developed in NetBeans

or Visual Studio after downloading a plug-in, however RubyMine IDE is more native to the framework. Ruby on Rails provides a range of publicly available libraries and tools allowing to quickly develop the applications (MacDonald, 2015). However, it is known for slow runtime speed (MacDonald, 2015) which is a major disadvantage in a banking application dealing with transactions. Furthermore, the documentation is lacking for certain libraries and gems, meaning that the developer may not know how to program a particular feature. While Ruby on Rails supports multithreading, certain libraries do not support that, resulting in performance issues.

Since the system requires a centralised data storage, a database will be used. All three languages allow to write database queries using Sequential Query Language (SQL). The main threat of using SQL is injections which enable the attackers to perform malicious actions on the database. In order to avoid this, parameterised queries may be used to run queries on the database. Java also offers JPQL (Java Persistence Query Language) which is a platform-independent OO query language, inspired by SQL. It enables the developers to define queries based on the entity model, suggesting that entity object is used instead of database table to define a query.

### 7.3 Chosen Technology Justification

Having explored the available technologies, it became apparent that this project will be implemented using C# ASP.NET in Visual Studio. Ruby on Rails was disregarded because it offers limited documentation and sometimes no documentation for the rarer libraries which is a major issue to the developer who has no previous experience with this technology. Similarly, Ruby on Rails was disregarded due to performance issues. ASP.NET and Java APIs on the other hand, provide thorough documentation which the developers may consult, enabling them to write components. Both frameworks also offer countless libraries and tools available for reuse. The deciding factor for disregarding Java was the lack of GUI components which would allow to build interfaces with good UX. Java could be used to develop the back-end and another technology could be used to develop a GUI by adopting SOA, however, Visual Basic offers all of these features within the same IDE.

C# ASP.NET was also chosen because the developer wanted to learn the language and the framework. Having previously done Java, C# syntax looked familiar but the framework introduced more possibilities. This technology is also considered to have easy learning curve and allows to easily write self-contained components which can be made available as services. ASP.NET allows to effortlessly develop SOAP and RESTful web-services using a singular framework, as opposed to Java which splits them into different modules. The ASP.NET framework provides all necessary classes for services and components, preventing the developer from spending time setting-up the project and resolving calls. It also offers several useful features, such as; automatically generating WSDL for web services, offering Windows native forms to be used as GUI and automatically generating the necessary classes. The database will be accessed using parameterised SQL queries in order to prevent SQL injections during the interaction.

### 7.4 Conclusion

This chapter has reviewed the three main technologies which could be used to develop a banking system using SOA. By examining each technology in detail, a suitable approach was selected to ensure the deployed system meets all requirements. The report will now proceed onto applying the chosen technology in practice in order to implement the first iteration of the system.

## 8 Implementation

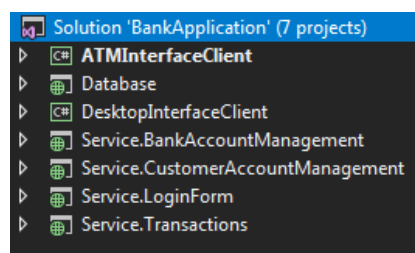
### 8.1 Introduction

The implementation chapter will demonstrate various aspects of the system and present several figures illustrating the implementation process. Additionally, various complications and limitations which arose during the system implementation using SOA will be discussed.

### 8.2 Application Functionality

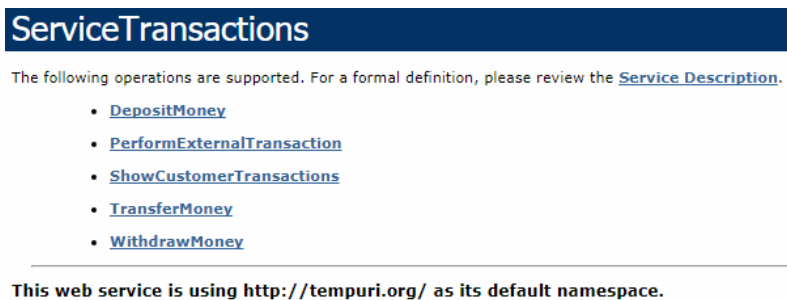
#### 8.2.1 Implementing SOA

In order to implement the system using SOA, it was necessary to create seven projects, shown in *Figure 21*. Each project showcases a single component from the component diagram. As a result of this, the deployed system fully matches the designed component diagram and is fully modularised.



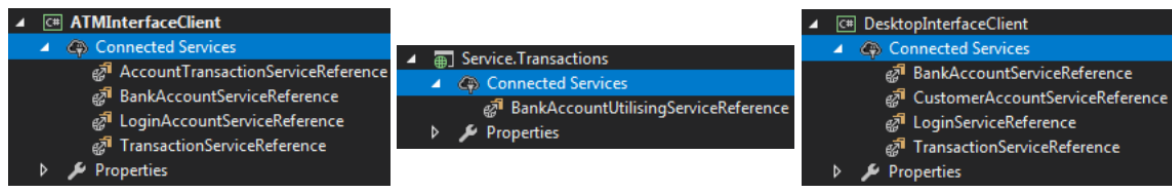
**Figure 21 – Banking Application Projects**

The Database provided data storage for the system by means of an SQL database on a localhost. The two clients provided a graphical user interface (GUI) to the users, allowing them to perform various operations using different interfaces. This is done to show that the same services can be consumed by a range of other components within the system. The four remaining projects are loosely-coupled components which provide specific services to the system. Please refer to *Appendix A* to view the classes of each component/service.



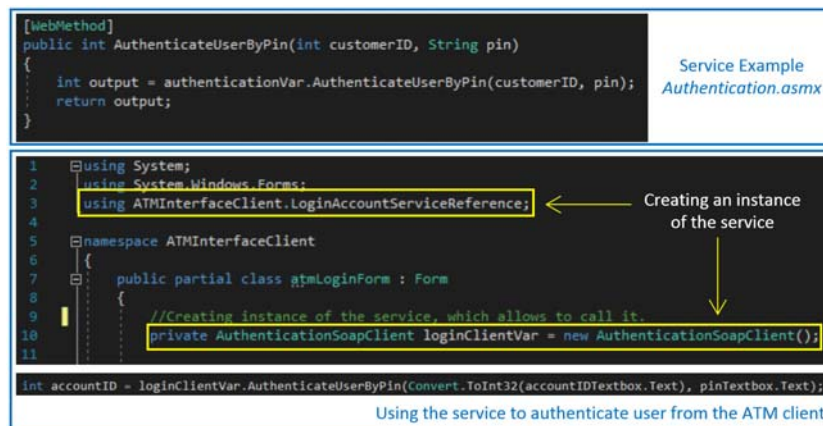
**Figure 22 – Service Interface Example**

Each service/component consisted of several classes to follow the Object Oriented (OO) paradigm. In order to make the services available, Visual Studio offered *.asmx* file which can be treated as an interface. This is because an instance of each class within the component was created in the *.asmx* file and all methods were invoked there, allowing the services to be discoverable. Once the file is run, all available services within the component are shown to the user, demonstrated in *Figure 22*. By clicking Service Description, any developer wishing to use the service can view the generated WSDL. *Appendix B* shows all four service components developed in this project.



**Figure 23 – Connecting services**

In order to link a service to another service or a component, it was necessary to locate the required service and add it to the Connected Services. *Figure 23* demonstrates how the three components utilised the services created within the project. As seen from the figure, the services are correctly linked to follow the component diagram design.

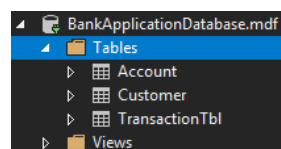


**Figure 24 – Service invocation example**

*Figure 24* outlines the general pattern of using the service in a stand-alone component. Firstly, the service is made available in the Authentication.asmx file whereby the implementation of the service is hidden in another class, commonly known as black-box approach. The .asmx file can be seen as an interface for the services because it lists all of the services available within the component and provides a method signature, allowing other developers to reuse these services. Once the service is made available and linked to the necessary component, the developer must create an instance of the service, allowing to call its methods. This then calls the service within the class in order to validate the user by passing the necessary parameters, in this example.

## 8.2.2 Database Implementation

The database is used to store all customers, accounts and transactions in a centralised place. The database fully follows the ER diagram created during the design phase.



**Figure 25 – Database Tables**

*Figure 25* shows all three tables which were used in this project. Please view *Appendix C* for the database table structure which is identical to the data dictionary created during design phase.



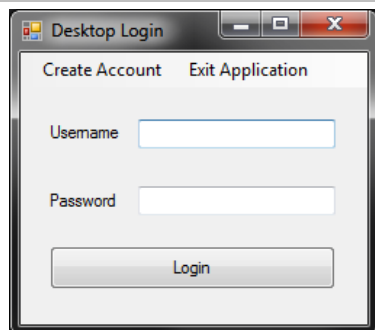
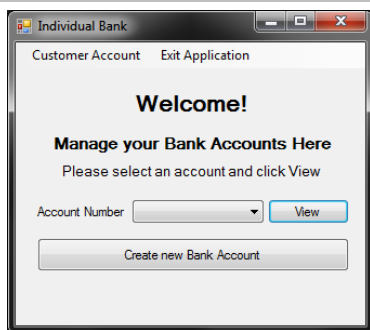
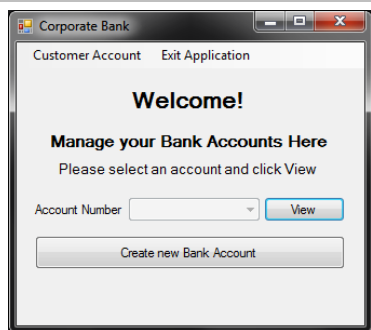
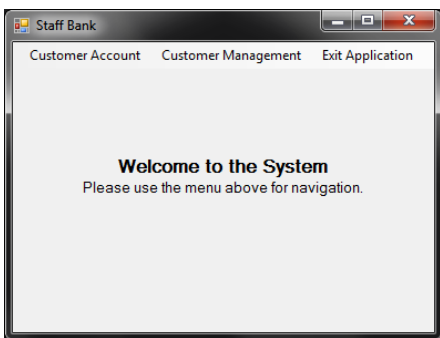
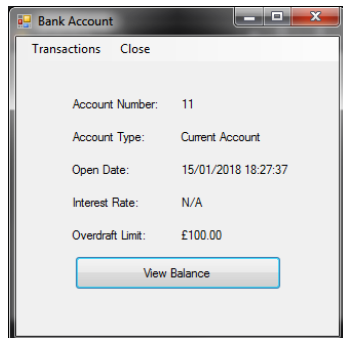
### 8.2.3 Offered Functionality

The system can be used by three different user types whose access privileges differ. These user types are individual customer, corporate customer and staff. The desktop client is accessible by all user groups and allows to perform functions shown in *Table 7*.

| DESKTOP CLIENT FUNCTIONALITY         | INDIVIDUAL USER | CORPORATE USER | STAFF USER |
|--------------------------------------|-----------------|----------------|------------|
| Create system Account                | ✓               | ✓              | ✓          |
| Delete system Account                | ✓               | ✓              | ✓          |
| Edit system Account Details          | ✓               | ✓              | ✓          |
| View All Customers                   |                 |                | ✓          |
| View Total Customer Count            |                 |                | ✓          |
| View All Accounts                    |                 |                | ✓          |
| Get Total Account Count              |                 |                | ✓          |
| Delete Bank Account                  |                 |                | ✓          |
| Perform External Transaction         |                 |                | ✓          |
| Change Interest Rate                 |                 |                | ✓          |
| Change Overdraft                     |                 |                | ✓          |
| Open Bank Account (saving & current) | ✓               | ✓              |            |
| View Bank Account Details            | ✓               | ✓              | ✓          |
| View Balance                         | ✓               | ✓              | ✓          |
| View Transaction History             | ✓               | ✓              |            |
| Withdraw Money                       | ✓               | ✓              |            |
| Deposit Money                        | ✓               | ✓              |            |
| Transfer Money                       | ✓               | ✓              |            |

**Table 7 – Desktop Client Functionality**

*Table 8* below demonstrates a selection of main application pages developed using Windows Forms.

|   |   |   |
|---|---|---|
|  |    |  |
| Central Login   | Individual User Home  | Corporate User Home   |
|  |  |   |
| Staff User Home   | Bank Account Page   |   |

**Table 8 – Desktop Interface Example**

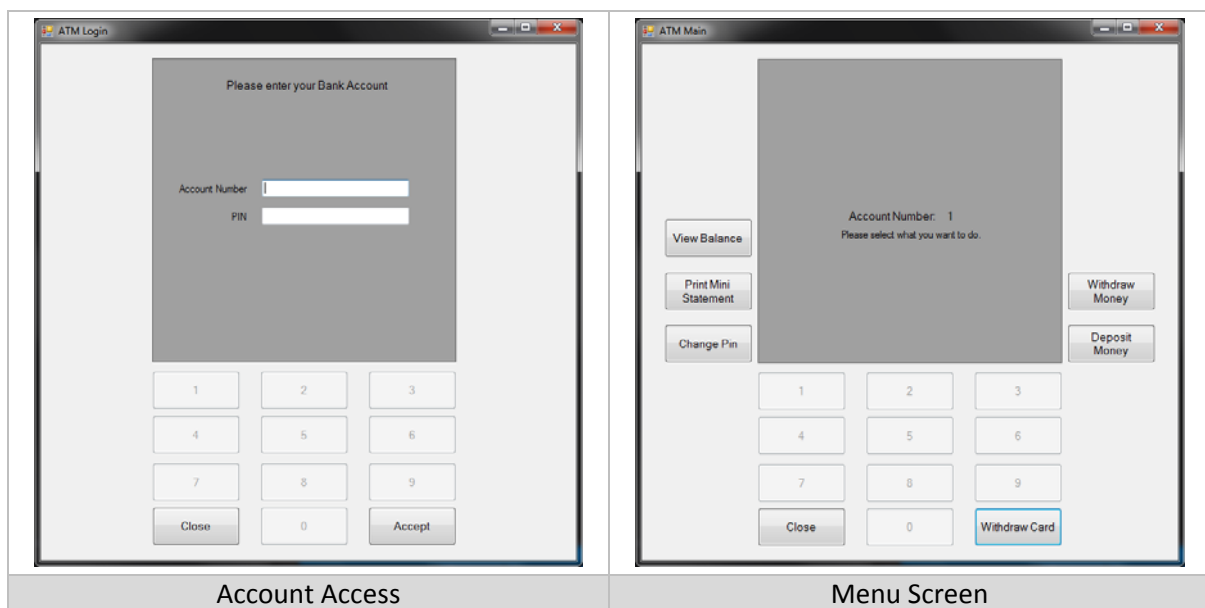


It was necessary to create unique home page for each user group because generating content based on user type had negative effect on the loading time due to the use of distributed services over the network. The ATM machine client was limited to individual and corporate customers who may perform the operations listed in *Table 9*.

| ATM CLIENT FUNCTIONALITY | INDIVIDUAL USER | CORPORATE USER |
|--------------------------|-----------------|----------------|
| View Account Balance     | ✓               | ✓              |
| Print Mini Statement     | ✓               | ✓              |
| Change Account Pin       | ✓               | ✓              |
| Withdraw Money           | ✓               | ✓              |
| Deposit Money            | ✓               | ✓              |

**Table 9 – ATM Client Functionality**

*Table 10* below demonstrates a selection of two ATM pages developed using Windows Forms.



**Table 10 – ATM Interface Example**

Due to time limit, it was not possible to make the number keys work correctly, therefore they have been disabled. This is not a major issue as the real ATM machine would provide real keys which could be pressed and linked to the client interface. Currently, the user can use the keyboard to input the numbers into the system.

The developed system fully matches the component diagram which was designed earlier in the project. Each input field has at least one type of validation to prevent users from entering incorrect data. The system is also equipped with a range of error messages which aid the user in using the system and understanding what was done incorrectly. All but one functional requirement identified were implemented into the current version of the system. The 'set-up direct debit' requirement could not be implemented because further research has shown that such operation is performed by external companies. However, it would be possible to use the 'WithdrawMoney' service with a set interval, achievable using SQL triggers. This would achieve a similar effect if the direct debit service is not available.

### 8.3 Challenges

This project had several challenges and complications which translated into limitations, preventing the system from being perfect. The following challenges arose during the project:

1. **All used technology is new for the developer** – The developer has no previous experience writing web-services and it was not explained well during the practical lectures. To overcome this, a number of video guides and books were examined to understand the best practices and adapt them in this project. Furthermore, C# in Visual Studio is also novel for the developer as previously the developer used Java OO language in NetBeans. However, the two languages are very similar and the developer could easily adapt to C# and IDE.
2. **Developing GUI with good UX in Visual Studio was challenging** – The main problem with Windows Forms is that they do not refresh after the data was updated in the model. The *Update()* and *Refresh()* functions did not help the situation as well as other suggested methods in the guides. To overcome this, the developer has created several forms which are opened when the menu item is clicked. This is believed to have a negative effect on the UX, although the users are able to perform all tasks as a result of this solution in the limited time.
3. **Unable to connect to an external database** – The external databases had access rights which stopped all connections. It is possible to add trusted IP addresses to the database, but the KUnet hosts do not provide such services. To overcome this, the database was hosted on a localhost of the machine where the system was developed.

### 8.4 Conclusion

The implementation section has focused on demonstrating the implemented system using the technology selected earlier. The developed system follows the diagrams developed during design phase and implements the requirements elicited during the analysis phase. All the requested functionality in the brief was implemented in the current revision of the system. The code follows OO concepts and contains comments to aid in system maintenance.

## 9 Conclusion

As a result of this report, a thorough analysis was carried-out in order to fully understand the problem and provide an appropriate solution consequently. A textual-description was used to investigate the brief and begin outlining the system classes, attributes, methods and relationships. The use case diagram was created to capture the user goals which helped form a list of prioritised functional requirements. Additionally, the system qualities were documented as non-functional requirements, along with a detailed measuring criteria.

A number of modern architectural styles were deeply reviewed and two suitable paradigms were selected for a deeper comparison. This evaluation assessed the suitability of both approaches in terms of the non-functional requirements which constitute the quality of the finished system. Subsequently, the SOA was selected for this project. A clear migration strategy from a LAN-based to a cloud-based system was established. Once the analysis was complete, an architecture design was created to illustrate the deployment of the system using SOA. The report then proceeded onto reviewing the potential technologies which could be used to implement the system using SOA. Having examined the three main approaches, the most suitable technology was selected which was used to implement the system. The developed system fully matches the diagrams created throughout the design phase by using loosely-coupled services and components in order to make the system modularised. The code follows OO concepts and contains comments to aid in system maintenance. The system can be used by three different user types via two clients which offer various functionality, allowing the users to manage their bank accounts and perform transactions. Each client offered a native Windows form GUI with meaningful error messages to aid the users. All functionality requested in the brief is available in the system to be used by the clients.

This project had a number of challenges which arose during the implementation which had a negative impact on the system, but were resolved by the developer. In future iterations of the system it would be advantageous to add security features such as encryption, enhanced UI with more input validations, new transactions such as; direct debit, standing orders and introduce a web-based client which can be accessible through a browser. While the current iteration of the system fully implemented the requirements discussed in the brief in order to demonstrate SOA, the system does not have bespoke UI (User Interface) and security features which would make the system usable by the real clients. Overall, this project was a successful learning journey which taught the developer new technologies.

## References

- Antonopoulos, N., Gillam, L. (2010) *Cloud Computing*. Springer-Verlag London Limited.
- Chatterjee, A. (2008) 'Testing Service-Oriented Architectures', *Dr Dobbs Journal*, 33(11), pp. 46-48.
- Chihi, H., Chainbi, W., Ghdira, K. (2016) 'Cloud computing architecture and migration strategy for universities and higher education', *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications*, vol.2016-. DOI: 10.1109/AICCSA.2015.7507140.
- Crotty, J., Horrocks, I. (2017) 'Managing legacy system costs: A case study of a meta-assessment model to identify solutions in a large financial services company', *Applied Computing and Informatics*, 13(2), pp.175-183. DOI: 10.1016/j.aci.2016.12.001.
- Dustdar, S., Haslinger, S. (2004) 'Testing of service-oriented architectures: A practical approach', *Lecture Notes in Computer Science*, vol.3263, pp.97-109.
- Goyal, S. (2014) 'Public vs Private vs Hybrid vs Community – Cloud Computing: A Critical Review', *I. J. Computer Network and Information Security*, vol.3, pp.20-29. DOI: 10.5815/ijcnis.2014.03.03.
- Hasan, A. (2017) 'Top 12 Main Advantages of ASP.NET Framework', *Arpatech*, 24 March 2017. Available at: <http://www.arpatech.com/blog/top-advantages-of-asp-net-framework/>. (Accessed: 24 January 2018).
- Hill, R., Hirsch, L., Lake, P., Moshiri, S. (2013) *Guide to Cloud Computing Principles and Practice*. London: Springer London: Imprint: Springer.
- Jyotsna, S. (2014) 'Component-Based Development Technologies and Limitations', *International Journal of Engineering and Computer Science*, 3(10), pp.8835-8838.
- Khaddaj, S. (2012) 'Cloud Computing: Service Provisioning and User Requirements', *11<sup>th</sup> International Symposium on Distributed Computing and Application to Business, Engineering & Science*, Oct.2012, pp.191-195, fig.3. DOI: 10.1109/DCABES.2012.76.
- Khaddaj, S. (2017) 'CI7250 Software Architecture and Programming Models Assessment Compendium', *CI7250: Software Architecture and Programming Models*. Kingston University. Unpublished.
- Koskela, M., Rahikainen, M., Wan, T. (2007) 'Software development methods: SOA vs. CBD, OO and AOP', *Proceedings of the seminar on Enterprise Information Systems: Service-Oriented Architecture and Software Engineering, Helsinki University of Technology*, 2007.
- MacDonald, R. (2015) 'Pros and Cons of Ruby on Rails', *Made Tech*, 8 September 2015. Available at: <https://www.madetech.com/blog/pros-and-cons-of-ruby-on-rails>. (Accessed: 24 January 2018).
- Microsoft (2018) *ASP.NET Overview*. Available at: <https://msdn.microsoft.com/en-us/library/4w3ex9c2.aspx>. (Accessed: 24 January 2018).
- Oracle (2010) *Building Web Services with JAX-WS*. Available at: <https://docs.oracle.com/javaee/5/tutorial/doc/bnayl.html>. (Accessed: 24 January 2018).

Oracle (2013) *Developing RESTful Web Services with JAX-RS*. Available at: <https://docs.oracle.com/javaee/6/tutorial/doc/gilik.html>. (Accessed: 24 January 2018).

Oussalah, M. (2014) *Software Architecture 1*. John Wiley & Sons, Incorporated.

Schulmeyer, G.G. (2007) *Handbook of Software Quality Assurance*. 4<sup>th</sup> edn. Norwood: Artech House.

Shim, B., Choue, S., Kim, S., Park, S. (2008) 'A Design Quality Model for Service-Oriented Architecture', *15<sup>th</sup> Asia-Pacific Software Engineering Conference*, Dec. 2008, pp.403-410. DOI: 10.1109/APSEC.2008.32.

Siau, K., Chiang, R., Hardgrave, B. (2011) *Systems Analysis and Design: People, Processed, and Projects*. Taylor and Francis.

Sommerville, I. (2010) *Software Engineering*. 9<sup>th</sup> edn. Harlow: Addison-Wesley.

Szyperski, C., Gruntz, D., Murer, S. (2002) *Component software: beyond object-oriented programming*. 2<sup>nd</sup> edn. New York: ACM Press; London: Addison-Wesley.

Voelz, D., Goeb, A. (2010) 'What is Different in Quality Management for SOA', *14<sup>th</sup> IEEE International Enterprise Distributed Object Computing Conference*, Oct. 2010, pp.47-56. DOI: 10.1109/EDOC.2010.27.

Vogel, O., Arnold, I., Chughtai, A., Kehrer, T. (2011) *Software Architecture*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Yadava, S., Singh, S. (2009) *An introduction to client/server computing*. New Delhi: New Age International P Ltd., Publishers.

Zaigham, M. (2013) *Cloud Computing Methods and Practical Approaches*. London; New York: Springer.

Zhang, Z., Zhou, D., Tang, H., Zhong, S. (2010) 'A service composition approach based on sequence mining for migrating e-learning legacy system to SOA', *International Journal of Automation and Computing*, 7(4), pp.584-595. DOI: 10.1007/s11633-010-0544-2.

Zhu, H. (2005) *Software Design Methodology: From Principles to Architectural Styles*. Elsevier Science.

## Appendix A – Component Classes


|                                      |  |
|--------------------------------------|--|
|                                      |  |
| <b>BankAccountManagement Service</b> | <b>Transactions Service</b>              |
|                                      |  |
| <b>LoginForm Service</b>             | <b>CustomerAccountManagement Service</b> |
|                                      |  |
| <b>Database</b>                      |  |
|                                      |  |
| <b>ATM Client</b>                    | <b>Desktop Client</b>                    |

.asmx files are web service interfaces.


## Appendix B – Service Examples

|   |   |
|---|---|
| <div>ServiceTransactions</div> <p>The following operations are supported. For a formal definition, please review the <a href="#">Service Description</a>.</p> <ul style="list-style-type: none"> <li>• <a href="#">DepositMoney</a></li> <li>• <a href="#">PerformExternalTransaction</a></li> <li>• <a href="#">ShowCustomerTransactions</a></li> <li>• <a href="#">TransferMoney</a></li> <li>• <a href="#">WithdrawMoney</a></li> </ul> <p>This web service is using <a href="http://tempuri.org/">http://tempuri.org/</a> as its default namespace.</p>   | <div>Authentication</div> <p>The following operations are supported. For a formal definition, please review the <a href="#">Service Description</a>.</p> <ul style="list-style-type: none"> <li>• <a href="#">AuthenticateUserByPassword</a></li> <li>• <a href="#">AuthenticateUserByPin</a></li> </ul> <p>This web service is using <a href="http://tempuri.org/">http://tempuri.org/</a> as its default namespace.</p>   |
| <div>Transactions Service</div>   | <div>LoginForm Service</div>  |
| <div>ServiceBankAccountManagement</div> <p>The following operations are supported. For a formal definition, please review the <a href="#">Service Description</a>.</p> <ul style="list-style-type: none"> <li>• <a href="#">CreateCurrentBankAccount</a></li> <li>• <a href="#">CreateSavingsBankAccount</a></li> <li>• <a href="#">DeleteBankAccount</a></li> <li>• <a href="#">GetAccountPin</a></li> <li>• <a href="#">GetAccountType</a></li> <li>• <a href="#">GetInterestRate</a></li> <li>• <a href="#">GetOpenDate</a></li> <li>• <a href="#">GetOverdraftLimit</a></li> <li>• <a href="#">GetRunningTotal</a></li> <li>• <a href="#">SetAccountPin</a></li> <li>• <a href="#">SetInterestRate</a></li> <li>• <a href="#">SetOverdraftLimit</a></li> <li>• <a href="#">SetRunningTotal</a></li> <li>• <a href="#">ShowAccountsForCustomer</a></li> <li>• <a href="#">ShowAllAccounts</a></li> <li>• <a href="#">ShowTotalAmountOfAccounts</a></li> </ul> <p>This web service is using <a href="http://tempuri.org/">http://tempuri.org/</a> as its default namespace.</p> | <div>ServiceCustomerAccountManagement</div> <p>The following operations are supported. For a formal definition, please review the <a href="#">Service Description</a>.</p> <ul style="list-style-type: none"> <li>• <a href="#">CreateNewCustomerAccountCorporate</a></li> <li>• <a href="#">CreateNewCustomerAccountIndividual</a></li> <li>• <a href="#">CreateNewCustomerAccountStaff</a></li> <li>• <a href="#">DeleteCustomerAccount</a></li> <li>• <a href="#">GetCustomerAddress</a></li> <li>• <a href="#">GetCustomerCompanyType</a></li> <li>• <a href="#">GetCustomerContactNumber</a></li> <li>• <a href="#">GetCustomerEmail</a></li> <li>• <a href="#">GetCustomerGender</a></li> <li>• <a href="#">GetCustomerName</a></li> <li>• <a href="#">GetCustomerPassword</a></li> <li>• <a href="#">GetCustomerSurname</a></li> <li>• <a href="#">GetCustomerType</a></li> <li>• <a href="#">SetCustomerAddress</a></li> <li>• <a href="#">SetCustomerCompanyType</a></li> <li>• <a href="#">SetCustomerContactNumber</a></li> <li>• <a href="#">SetCustomerEmail</a></li> <li>• <a href="#">SetCustomerGender</a></li> <li>• <a href="#">SetCustomerName</a></li> <li>• <a href="#">SetCustomerPassword</a></li> <li>• <a href="#">SetCustomerSurname</a></li> <li>• <a href="#">SetCustomerType</a></li> <li>• <a href="#">ShowAllCustomers</a></li> <li>• <a href="#">ShowTotalAmountOfCustomers</a></li> </ul> <p>This web service is using <a href="http://tempuri.org/">http://tempuri.org/</a> as its default namespace.</p> |
| <div>BankAccountManagement Service</div>  | <div>CustomerAccountManagement Service</div>  |


## Appendix C – Database Table Structure

| Update  |                | Script File: dbo.Account.sql |                                     |             |
|---|----------------|------------------------------|-------------------------------------|-------------|
|   | Name           | Data Type                    | Allow Nulls                         | Default     |
|  | accountID      | int                          | <input type="checkbox"/>            |             |
|   | openDate       | datetime                     | <input type="checkbox"/>            | (getdate()) |
|   | runningTotal   | decimal(15,2)                | <input type="checkbox"/>            |             |
|   | interestRate   | decimal(15,2)                | <input checked="" type="checkbox"/> |             |
|   | overdraftLimit | decimal(15,2)                | <input checked="" type="checkbox"/> |             |
|   | accountType    | varchar(1)                   | <input type="checkbox"/>            |             |
|   | customerID     | int                          | <input type="checkbox"/>            |             |
|   | accountPin     | varchar(4)                   | <input type="checkbox"/>            |             |
|   |                |                              | <input type="checkbox"/>            |             |

**Account table**

| Update  |                       | Script File: dbo.Customer.sql |                                     |         |
|---|-----------------------|-------------------------------|-------------------------------------|---------|
|   | Name                  | Data Type                     | Allow Nulls                         | Default |
|  | customerID            | int                           | <input type="checkbox"/>            |         |
|   | customerName          | varchar(30)                   | <input type="checkbox"/>            |         |
|   | customerAddress       | varchar(30)                   | <input type="checkbox"/>            |         |
|   | customerContactNumber | varchar(11)                   | <input type="checkbox"/>            |         |
|   | customerEmail         | varchar(40)                   | <input type="checkbox"/>            |         |
|   | customerPassword      | varchar(20)                   | <input type="checkbox"/>            |         |
|   | customerSurname       | varchar(30)                   | <input checked="" type="checkbox"/> |         |
|   | customerGender        | varchar(6)                    | <input checked="" type="checkbox"/> |         |
|   | customerCompanyType   | varchar(10)                   | <input checked="" type="checkbox"/> |         |
|   | customerType          | varchar(1)                    | <input type="checkbox"/>            |         |
|   |                       |                               | <input type="checkbox"/>            |         |

**Customer table**

| Update  |                  | Script File: dbo.TransactionTbl.sql |                                     |             |
|---|------------------|-------------------------------------|-------------------------------------|-------------|
|   | Name             | Data Type                           | Allow Nulls                         | Default     |
|  | transactionID    | int                                 | <input type="checkbox"/>            |             |
|   | timeStamp        | datetime                            | <input type="checkbox"/>            | (getdate()) |
|   | amount           | decimal(15,2)                       | <input type="checkbox"/>            |             |
|   | transactionType  | varchar(2)                          | <input type="checkbox"/>            |             |
|   | branchCode       | varchar(20)                         | <input checked="" type="checkbox"/> |             |
|   | accountID        | int                                 | <input type="checkbox"/>            |             |
|   | otherInformation | varchar(20)                         | <input checked="" type="checkbox"/> |             |
|   |                  |                                     | <input type="checkbox"/>            |             |

**TransactionTbl table**

It was not possible to use *Transaction* table name instead of *TransactionTbl* because it is a reserved keyword in SQL. This would prevent from writing queries on that table.