AN ANALYSIS OF BUSINESS IMPACT FROM CQRS PATTERN-BASED MISSION CRITICAL APPLICATIONS

VITU HANSAKUL

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR
THE DEGREE OF MASTER OF SCIENCE (TECHNOLOGY OF INFORMATION SYSTEM MANAGEMENT)
FACULTY OF GRADUATE STUDIES
MAHIDOL UNIVERSITY
2014**

Thesis
entitled

AN ANALYSIS OF BUSINESS IMPACT FROM CQRS PATTERN-BASED MISSION CRITICAL APPLICATIONS

……..……..……
Mr.Vitu Hansakul
Candidate

……..……..……
Asst. Prof. Supaporn Kiattisin, Ph.D.
(Electrical and Computer Engineering)
Major advisor

……..……..……
Asst. Prof. Adisorn Leelasantitham,
Ph.D. (Electrical Engineering)
Co-advisor

……..……..……
Lect.Waranyu Wongseree, Ph.D.
(Electrical Engineering)
Co-advisor

……..……..……
Prof. Banchong Mahaisavariya,
M.D., Dip Thai Board of Orthopedics
Dean
Faculty of Graduate Studies
Mahidol University

……..……..……
Asst. Prof. Supaporn Kiattisin, Ph.D.
(Electrical and Computer Engineering)
Program Director
Master of Science Program in
Technology of Information System
Management
Faculty of Engineering
Mahidol University

Thesis
entitled

AN ANALYSIS OF BUSINESS IMPACT FROM CQRS PATTERN-BASED MISSION CRITICAL APPLICATIONS

was submitted to the Faculty of Graduate Studies, Mahidol University
for the degree of Master of Science (Technology of Information System Management)
on
November 1, 2014

……
Mr.Vitu Hansakul
Candidate

……
Lect. Sorarat Thammaboosadee, Ph.D.
(Information Technology)
Chair

……
Asst. Prof. Kairoek Choeychuen, Ph.D.
(Electrical and Computer Engineering)
Member

……
Asst. Prof. Supaporn Kiattisin, Ph.D.
(Electrical and Computer Engineering)
Member

……
Asst. Prof. Adisorn Leelasantitham,
Ph.D. (Electrical Engineering)
Member

……
Lect.Waranyu Wongseree, Ph.D.
(Electrical Engineering)
Member

……
Prof. Banchong Mahaisavariya,
M.D., Dip Thai Board of Orthopedics
Dean
Faculty of Graduate Studies
Mahidol University

……
Lect. Worawit Israngkul, M.S.
Dean
Faculty of Engineering
Mahidol University

ACKNOWLEDGEMENTS

I would like to express my special appreciation and thanks to my advisor Asst. Prof. Supaporn Kiattisin, Ph.D., together with my co-advisors Asst. Prof. Adisorn Leelasantitham, Ph.D. and Lect.Waranyu Wongseree, Ph.D., for the useful comments, remarks and engagement through the learning process of this Master thesis. Furthermore I would also like to thank Theera Piroonratana, Ph.D. for introducing me to the topic as well for the aspiring guidance, invaluably constuctive criticism and friendly advice during the research.

A special thanks to my family. Words cannot express how grateful I am to my mother, father, mother-in law, elder sister for all of the sacrifices that you have made on my behalf. Your prayer for me was what sustained me thus far. I would also like to thank all of my friends who supported me in writing, and incented me to strive towards my goal.

At the end I would like express appreciation to my beloved Nichakant Soontharanont who has supported me throughout entire process, both by keeping me harmonious and helping me putting pieces together. I will be grateful forever for your love.

Vitu Hansakul

AN ANALYSIS OF BUSINESS IMPACT FROM CQRS PATTERN-BASED MISSION CRITICAL APPLICATIONS

VITU HANSAKUL 5537221 EGTI/M

M.Sc. (TECHNOLOGY OF INFORMATION SYSTEM MANAGEMENT)

THESIS ADVISORY COMMITTEE : SUPAPORN KIATTISIN, Ph.D., ADISORN LEELASANTITHAM, Ph.D., WARANYU WONGSEREE, Ph.D.

ABSTRACT

This thesis discusses an application of the Command Query Responsibility Segregation (CQRS) architectural pattern in mission critical business software. It can be used within collaborative domains to decouple the requests of the system (commands) from the processing of that request (queries). We built a proof-of-concept application to give us ideas on what types of impact we would face when implementing the CQRS pattern in the real world. Finally, we ran scalability tests and found that the CQRS pattern, which focused on collaborative domains, has far more potential to improve your system scalability while decreasing long-term operational costs for your business than just applying the traditional CRUD N-tier architecture in every portion of the system.

KEY WORDS: COMMAND QUERY RESPONSIBILITY SEGREGATION/ DOMAIN-DRIVEN DESIGN/ENTERPRISE PATTERN

44 pages

บทวิเคราะห์ผลกระทบทางธุรกิจที่เกิดจากการพัฒนาแอพพลิเคชันที่สำคัญในองค์กร โดยใช้ CQRS PATTERN

AN ANALYSIS OF BUSINESS IMPACT FROM CQRS PATTERN-BASED MISSION CRITICAL APPLICATIONS

วิทู หงสกุล 5537221 EGTI/M

วท.ม. (เทคโนโลยีการจัดการระบบสารสนเทศ)

คณะกรรมการที่ปรึกษาวิทยานิพนธ์ : สุภาภรณ์ เกียรติสิน, Ph.D., อดิศร ลีลาสันติธรรม, Ph.D., วรัญญู วงษ์เสรี, Ph.D.

บทคัดย่อ

วิทยานิพนธ์ฉบับนี้กล่าวถึงแนวทางการพัฒนาแอพพลิเคชันที่สำคัญในองค์กร โดยใช้แบบแผนสถาปัตยกรรม Command Query Responsibility Segregation (CQRS) ท่านสามารถนำหลักการของ CQRS มาใช้ออกแบบซอฟต์แวร์ส่วนที่รองรับ Collaborative Domain ซึ่งมีการแยกโมเดลเขียนคำสั่ง (คอมมานด์) กับโมเดลอ่านข้อมูล (คิวรี) ออกจากกัน ผู้วิจัยได้พัฒนาแอพพลิเคชันจำลองขึ้นเพื่อศึกษาผลกระทบต่างๆ ที่องค์กรจะต้องเผชิญเวลาอิมพลีเม็นต์ CQRS ในสถานการณ์จริง พร้อมทั้งทดสอบประสิทธิภาพในการรองรับภาระงานขนาดต่างๆ เปรียบเทียบกันระหว่างแอพพลิเคชันที่ออกแบบตามหลักการของ CQRS กับแอพพลิเคชันที่ออกแบบโดยใช้สถาปัตยกรรม CRUD N-tier ดั้งเดิม ซึ่งผลลัพธ์ที่ปรากฏเป็นเครื่องพิสูจน์ว่า การออกแบบแอพพลิเคชันเฉพาะในส่วน Collaborative Domain ให้เป็นไปตามหลักการของ CQRS นั้นช่วยเพิ่มศักยภาพในการรองรับภาระงานที่เพิ่มขึ้น และลดค่าใช้จ่ายของธุรกิจในระยะยาวได้มากกว่าการประยุกต์ใช้สถาปัตยกรรม CRUD N-tier ในทุกส่วนของแอพพลิเคชัน

44 หน้า

CONTENTS

CONTENTS (cont.)

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER I
INTRODUCTION

1.1 Motivation

CQRS stands for Command Query Responsibility Segregation, which is a software design pattern behind a really simple idea: using a different model to update information than the model you use to read information. This separation occurs based upon whether the methods are “queries” asking to read data, or “commands” asking to change the state of the system.

As you can see, the idea is quite simple and not attractive by itself. What makes it especially attractive is the architectural possibilities that it opens. CQRS has been generating a great deal of interest among developer communities since the first introduction by Greg Young in 2010 (1), which reflects the enthusiasm amongst the many software architects from across the industry who are recomposing them in new ways to solve today’s business challenges.

The situation described above made us become interested with CQRS and started a research to find out how to implement CQRS on selected collaborative domain, instead of traditional Create, Read, Update and Delete (CRUD) approach, in ways that it will make a mission-critical business applications more flexible, more maintainable, and more scalable.

1.2 Problem Statement

A mission-critical application is the heart of most business operations occurred today. It means if this kind of software crashes or responding slowly to high volume transactions, you may be out of business.

Maintainability and long-term sustainability will become two crucial factors when designing mission-critical applications. After that, you must take the following two main concerns into consideration (2):

- Managing complexity in the core domain of the business application.
    - How to effectively apply complex business rules into the software.
    - What best design patterns and practices to choose, depending on the context.
- Providing enough quality of service to end users
    - How crosscutting measures (such as availability, scalability, security, operations, and so on) are designed and implemented.
    - What infrastructure architecture type to choose to meet the given service level agreement.

One software design paradigm that is found in common today is the N-tier architecture, which in general, the Graphical User Interface (GUI) is built on the top-tier while the bottom-tier is the database. This kind of system usually revolves around the Create, Read, Update and Delete (CRUD) operations that apply to the same subset of rows in one or more tables in a single data repository. In other words, user interactions on the GUI will finally be translated into some forms of CRUD.

Old-styled CRUD works well when there are only a few business logics applied to the data operation, but this approach still has some weak spots (3):

- When executing an Update operation, redundant columns may have to be updated correctly even though they are not required as part of the operation.
- Data contention risks that occurred when there are some records locked in the database, or update conflicts caused by concurrent updates. These risks increase as the complexity and result in slower system performance.
- Since each entity experiences both read and write operations, it might accidentally expose data in the wrong context, which means it’s more difficult to manage security and permissions of the system.

The solution we would like to experiment here is using CQRS (Command Query Responsibility Segregation) pattern. Its main idea is to separate the commands (Insert, Update, and Delete operations) model from the queries (Read operations) model (4), which means executing commands is done by different components from the one that executing queries, all of which can be done in a distributed and parallel manner. Also, the data store can be split into main database and read database in

different physical stores to further enhance performance, scalability and security of the system.

We started by building a fairly simple proof-of-concept business application in traditional CRUD N-tier style to see if we can find an example of the very specific situation to which the pattern of CQRS can apply. And then we will rewrite the code in that one particular module to see how it improves. Next, we will be running comparison tests to measure the performance of the system before and after applying CQRS. Lastly, we analyzed what kind of significant impact will CQRS software development pattern have on the real-world business.

1.3 Research Objectives

- Demonstrate how CQRS pattern can be implemented in mission-critical applications by creating a small prototype application using CQRS concepts.
- Develop a decision criteria that can help business organizations choose whether to go with CQRS or with CRUD N-tier architecture for their next mission-critical application projects.
- Analyze what business impacts will CQRS-based software have on the real world organization, both on the positive and negative sides.

1.4 Research Scopes

- We choose the online order processing as a problem domain to create our sample product purchase and inventory application, for proof-of-concept purpose.
- The prototype application, which have been developed using traditional CRUD N-tier and CQRS patterns, is written using Microsoft C#.NET language, and run on the local Windows PC.
- The prototype application will be designed using Domain-Driven Design (DDD) principle.
- Microsoft Visual Studio 2013 is selected as our integrated development environment (IDE) tool.

- In this prototype application, the commands and queries model is backed by Microsoft SQL Server relational database management system (RDBMS).
- In this thesis, we focus on measuring and comparing the scalability between our CRUD N-tier and CQRS-based sample applications under different concurrent transactions, and analyzing the business impact from CQRS using lessons learned from the software prototypes.

1.5 Expected Results

- Practical knowledge about how to apply CQRS pattern to enhance the software scalability, simplify complex aspects of the business domain, increase flexibility of an application, and increase adaptability to changing business requirements.
- Scalability performance test results between our example CRUD N-tier and CQRS application focusing on the collaborative domain, to confirm that we should go with CQRS for our next projects.
- The summarization of impact that CQRS-based application will have on the business organizations, both on positive and negative sides.

CHAPTER II
LITERATURE REVIEW

2.1 N-tier Applications

2.1.1 Separation of Concerns

The N in N-tier refers to the number of tiers. A very common value for N is 3, made up of the three tiers of the presentation, business logic, and data access logic (5). When an application's behavior is splitted among multiple tiers, these tiers are said to be logically separated. In some cases, the separate tiers are deployed to different devices too.

The logical separation of concerns (6) is expressed in the source code of the application. The code's organization and design reflect the layers that are in use. At a minimum, individual classes should not include responsibilities that correspond to separate layers. Moreover, typically, each layer will have its namespace, assembly, and project in the solution. The logical separation of an application across layer boundaries represents a software design pattern.

The physical separation of the application refers to how it is deployed. It is possible to have a logically separated application that is physically deployed in a single location. For instance, a web application with different assemblies for business logic and data access, all of which are deployed in the same web server. However, it is possible to separate these assemblies and deploy them to different physical locations, whether on a single device or separate machines.

2.1.2 Traditional Data-Centric N-tier Design

See figure 2.1 showing different types of N-tier or several levels of data-centric tier application design. This figure presents a number of different options for separating application responsibilities into layers and tiers.

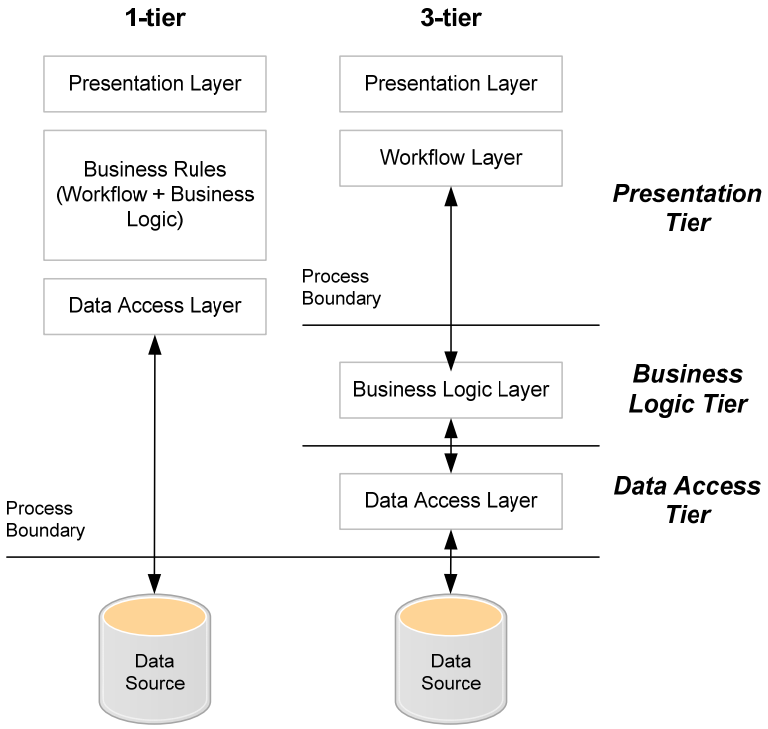

Figure 2.1 – 1-tier and 3-tier data-centric architecture

From the figure above, a typical application is composed of a presentation layer, some business rules, a data access layer, and a data source. The different layers are divided into their own separate deployable components. These could be different assemblies in the application, or they could be separate applications themselves in a more distributed architecture. The process boundaries might be in between separate processes on a single machine, or they could reside on different machines and communicate over a network. The three-tier design is one of the most common best practice application designs.

The workflow is encapsulated into services and different presentation layers which will utilize various services with interfaces appropriate to their workflow. For instance, the workflow of a large desktop's line of business application is likely to be rather dissimilar from the workflow expressed by a single screen of a mobile phone application, even if both of them are attempting to achieve the same task.

2.1.3 Domain-Centric Design

One of the problems with traditional data-centric N-tier approach is that it tends to contribute to a substantial amount of coupling between the applications at every tier in the database. In this domain-centric design, we can invert this dependence on external systems like databases so that the application can remain more loosely coupled, maintainable, and testable.

Domain-centric design differs from data-centric design and that its emphasis is on the domain. The domain simply describes the problem-space of your application. It puts domain objects that model the problem domain, which includes all business logic and behavior required to model the system, at the center of the design.

The domain objects should include any state involved to model the system's behavior properly, but they should not depend on external infrastructure concerns such as databases or file systems. The other important thing for domain-centric design is the Dependency Inversion Principle (7), which states that the high-level module should not depend on low-level modules, both should depend on abstractions.

This particular architectural designer pattern goes by various names including the most recent name Onion Architecture (8), which is named by Jeffrey Palermo because of the circular layer surrounding the core as shown in Figure 2.2.

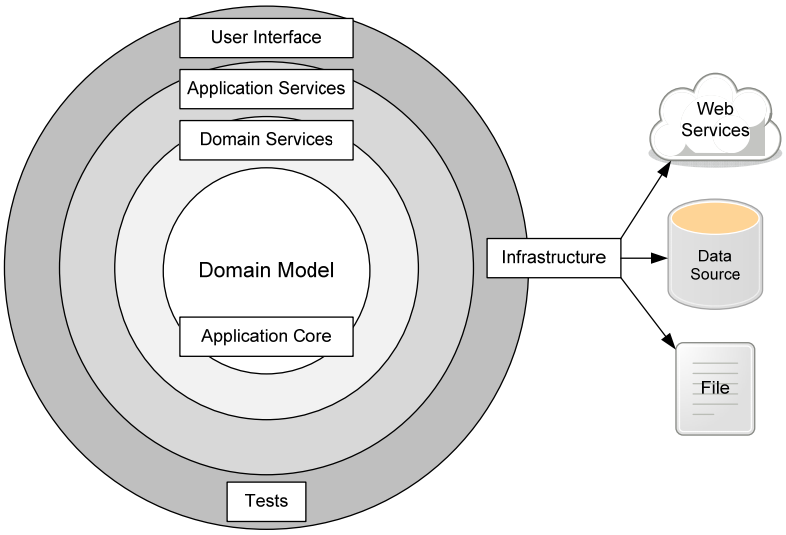

Figure 2.2 – Onion architecture

The figure expresses how the domain model and services live at the center of the Onion while the UI, test, and infrastructure all live at the edge, and depend inwardly on the core. This domain-centric architecture can also be shown as layers like Figure 2.3.

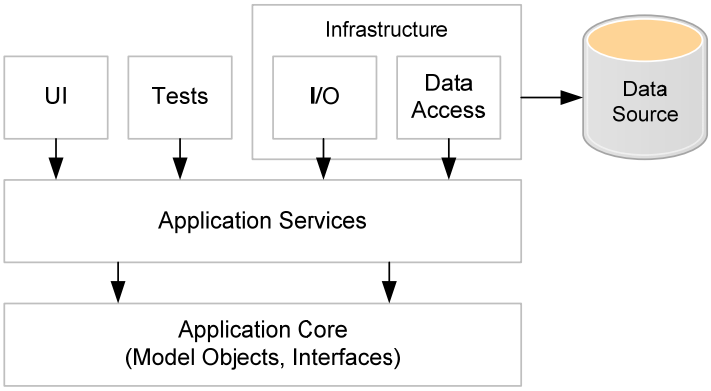

Figure 2.3 – Domain-centric architecture

The arrows represent dependency. There is no transitive dependency between the UI layer or test in the database, which means that we can test this application without the need to setup brittle, slow, and difficult to test infrastructure. The result will be a match more loosely coupled system with separate modules that can truly be swapped in and out and tested in isolation from another.

In order to correctly use the domain-centric architecture, it is important to follow certain principles. The first one is that you should ensure that all code is written to only reference code at the same layer, or layers closer to the core. The Domain Model objects themselves should all live at the core and thus should not have any dependency on infrastructure concerns. Lower layers defined the interfaces that implemented by upper layers. These interfaces are the abstractions that are described in the definition of the Dependency Inversion Principle.

Behavior and services will be also layered around the domain model. Some services will exist in the application core while other implementations will need to live in infrastructure because of their dependencies. Strive to push as much dependency free code into the core as possible.

To summarize, we found that the domain centric design forces you to focus your application on the applications domain logic rather than on the database. Typical traditional data centric N-tier design tended to force all of your dependencies in your system to point toward the database. By using the domain-centric design, we are able to use something called dependency injection (7) which relies heavily on the dependency inversion principle to make it so that our application logic was at the center of the dependencies in the system.

2.2 CAP Theorem

To understand the need for CQRS patterns, we must first realize the fundamental nature of distributed systems, and the best spot to begin is with the CAP theorem. It was first postulated by Dr. Eric Brewer (5), then proved by Gilbert and Lynch (6). This theorem describes the behavior of a distributed system, and here the definition of a distributed system is rather specific. It is a collection of interconnected nodes that all share data. A client can write data to a distributed system by talking to any one of these nodes, and then it can read data from a distributed system by talking to either that same node or a different node.

This theorem talks about how the system reacts when it gets a write request followed by a read request. The theorem states that for any given pair of requests, a write followed by a read, a distributed system can promise to guarantee only two out of three attributes. These attributes are consistency, availability, and partition tolerance. You can get any two at any given time, but you cannot have all three. You have to give up the third one for that particular pair of requests. Here's what those three guarantees mean.

- Consistency
- Availability
- Partition tolerance

Consistency means that the system promises to read data that is at least as latest as what you just wrote. So, whether the client reads from the same node that I just wrote to or from a different node, that node is not allowed to return stale data. So, somebody else might have written something newer, and the client might see their

change, but consistency guarantees that the client will not see older data than what it just wrote.

Availability means that a non-failing node will give the client a reasonable response within a reasonable amount of time. Now, all that's relative, but what that means is that it will not hang indefinitely, and it will not return an error. It applies to both the read and to the write request. So, that means that the write request will acknowledge that the data was actually written, and the read request will return valid data. Neither of these requests can returns an error, and neither one is allowed to hang indefinitely. So, this guarantee only applies to non-failing nodes. A node itself could actually be down, and the system would remain available. If the client can get access to any non-failing node and that node respond without an error in a reasonable amount of time, then the availability guarantee is upheld.

Partition tolerance guarantees that a distributed system will continue to function in the face of network partitions. A network partition is a breaking connectivity. It means that nodes within the system cannot communicate with one another. A partition could be isolated to just the connection between two distinct nodes or it could run through the entire system. On the other hand, a partition could be just a temporary loss of connectivity like maybe the loss of a single packet due to line noise or a partition could refer to something permanent like a backhoe cutting through a buried cable. However, if the distributed system continues to work when the network is partitioned, then it's said to be partition tolerant.

So, those are the three guarantees: Consistency, Availability, and Partition Tolerance. The CAP Theorem tells us that we can only have two. We cannot have all three at any given point in the system, so we have to choose, and our choices are limited.

2.3 Domain Driven Design

We are going to explain Domain Driven Design (DDD) mentioned in the book written by Eric Evans (11). It is about how to work with business owners to create customs software. The purpose of this book is that we should permit the problem domain drive the software design and not try to make the problem conform to

the capabilities of the software. Frequently, we approach the problem by modeling it as rows and columns in a database and then writing CRUD operations to manage that data. Once we deliver such applications to the business owners, they often find no relationship between the data grids and input forms that we have delivered and the actual business processes and workflows that they were trying to model. Evan's method seeks to change all that. In this topic we are going to concentrate on three core concepts of DDD, which are the ubiquitous language, bounded contexts and aggregate roots.

2.3.1 Ubiquitous Language

The process of Domain Driven Design begins by agreeing upon a ubiquitous language between the developers and the business owners. It is not something that we can just jot down in one or two meetings. It is a continuous cultivation and curation of language over the entire lifetime of the project. All of the code is modeled and written using the terms from this ubiquitous language, so any change to the ubiquitous language necessitates a change to the model. That indicates that the understanding has evolved in some way, and so the model has to evolve to capture that new understanding.

A ubiquitous language allows developers and business owners to communicate with one another without translation. Assumptions tend to get lost in the translation because each party defines their own meaning to the words they use to communicate with other people. However, by agreeing upon a ubiquitous language that forces the two parties to define their terms and flushes assumptions into the light.

2.3.2 Bounded Contexts

The next task in a DDD project is to carve the domain into bounded contexts. The term bounded context refers to the circumstances in which certain words of the ubiquitous language have particular meanings. Each context uses a particular dialect of the ubiquitous language, and each one is optimized to solve a specific problem.

Bounded contexts also offer benefits over enterprise data models in terms of the CAP Theorem. An enterprise data model represents one tightly interconnected

cluster of nodes in a distributed system where consistency is prioritized over availability. Any party that needs a consistent view of any part of the enterprise needs to make a connection to this one cluster. The more consumers that connect to it, the less available it becomes. The only method to combat this trend is to reduce the likelihood of a network partition by spending more money on expensive hardware and network management. An enterprise data model forces you to scale up rather than scaling out.

However, a bounded context represents a smaller cluster of nodes. It is decoupled from the other contexts. Because it has its model, it is not reliant upon its connections to those other contexts. It can make decisions based only on the information that it has on hand. A bounded context could be written to favor an availability over consistency. If the system of record is down, then the downstream system is unaffected. It already has a cache of the data that it needs to be organized and optimized for the problems that it is designed to solve. This process isn't automatic. You have to conscientiously choose to build this kind of capability using CQRS, or any of the other patterns and techniques that we're going to look at, but this is a choice that you don't easily have if you use an enterprise data model.

2.3.3 Aggregate Roots

Each bounded context has its domain model. A domain model is made up of entities and value objects. An entity is something that has inherent identity, and that can change over time whereas a value object has no identity of its own, and it is immutable. When modeling the entities and value objects in a bounded context, Evans describes them in terms of aggregates (11).

An aggregate is made up of several different objects composed into a single business unit. Business operations are performed on this aggregate rather than on the individual entities and value objects within the unit. The top level entity within this unit of objects is called the aggregate root. There are a few rules that Evans lays out related to aggregates and aggregate roots. We can categorize these rules as identity rules, reference rules, and operation rules.

So, let's look at the identity rules. The root of the aggregate is an entity, not the value object. The root has a global identity whereas other entities within the

aggregate have a local identity. So, the root is an entity and has a global identity. Remember the difference between entities and value objects. Entities have identity, and value objects do not.

Identity is the property of an object that uniquely differentiates it from other objects. It allows us to act upon one object without affecting another one, so this first rule ensures that we can uniquely identify the root of the aggregate and differentiate it from any other aggregate root in the system.

That second rule is further stating that we cannot globally identify any other entity in the aggregate. We cannot search for one of the child objects in order to find which aggregate it belongs to. We have to first identify the root, and then I can walk down to the child.

Next, let's look at the reference rules. There are three of these. The first one is the root, which is the only member of an aggregate that outside objects can hold a reference to. Secondly, objects within an aggregate can hold references to one another. And third, objects within an aggregate can hold references to other aggregate roots.

So, if you take these three rules together, then we should be able to draw a boundary around an aggregate where only the root is on the boundary. All the other objects are completely inside the boundary and wholly owned by the root. Any references, any arrows that are coming into the aggregate can only point at the root. They cannot cross the boundary and point at any of the children. We could have references coming from the children out of the aggregate pointing to other aggregate roots, but we cannot have any arrows that are pointing into the boundary.

Let's take a look at the operation rules. First of all, the query process. Only aggregate roots can be obtained through database queries. And then we have the update operations. Invariants are enforced only within the scope of a single aggregate. And then the delete operation. Deleting an aggregate root must delete all other objects within the boundary. So, this set of rules tells us how we can query, alter, and delete aggregates.

We can only query for aggregate roots. We cannot query for children. Remember, only the aggregate root has the global identity, so all of the other entities must have local identity, and that means we cannot query for them. Second, when we

alter the aggregate, the aggregate root is responsible for validating its invariants. An invariant is a rule that relates all the objects within the aggregate, and this rule must always be true. An aggregate root can relate the objects within the aggregate, but it cannot enforce behaviors outside of the aggregate boundary. Next one is the third rule. Since a child object is wholly owned by the aggregate root, then deleting the root must also delete all of the objects within the aggregate, all of the children.

2.4 CQRS

2.4.1 Origins in CQS

Bertrand Meyer wrote about CQS in Object-Oriented Software Construction (12). This book lays out a system of reasoning about the behavior of objects. It says that a method should either change the state of an object or return a result, but not both. When we are following this principle, we'll separate our methods into these two sets, those that can change states and those that return results. Methods that change state are called commands, and methods that return results are called queries. Both kinds of methods can be seen in Figure 2.4

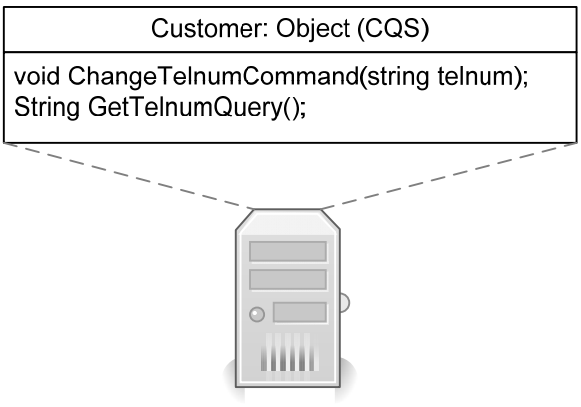

Figure 2.4 – An object constructed according to CQS

Meyer feels that adhering to a strict separation between commands and queries in a large system is key to success. Modules can safely call into other modules knowing that any queries they run will be without side effect. He goes on to suggest that a class that implements CQS should implement at least two interfaces, one which describes the functions and one that describes the procedures. If it is undesirable that other modules be able to update the state of the system, then all that need be exported is the interface that defines the methods.

This is a theory which is in strong agreement with the SOLID principles of software design (13) specifically the Interface Segregation Principle which states that "Clients should not be forced to depend upon interfaces that they do not use". Interfaces should be as narrow as possible to reduce the risk from changing the underlying class. If the method to change was not one of the ones presented in an interface to another component then the component need not worry about changes to the underlying object.

When you separate the methods like this, the caller can be confident that they can execute whatever series of queries they need to in order to arrive at the desired result. Then they can arrive at a determination based on that result and record that decision with a single command. This style of use is much easier to reason about because there's only one state change, and that occurs at the end of the sequence.

CQRS as an architectural pattern takes CQS idea even further by separating the responsibilities by creating two models, one for reads and one for write. An example of how the CQS object in Figure 2.4 can be transformed into a CQRS style object can be seen in Figure 2.5.

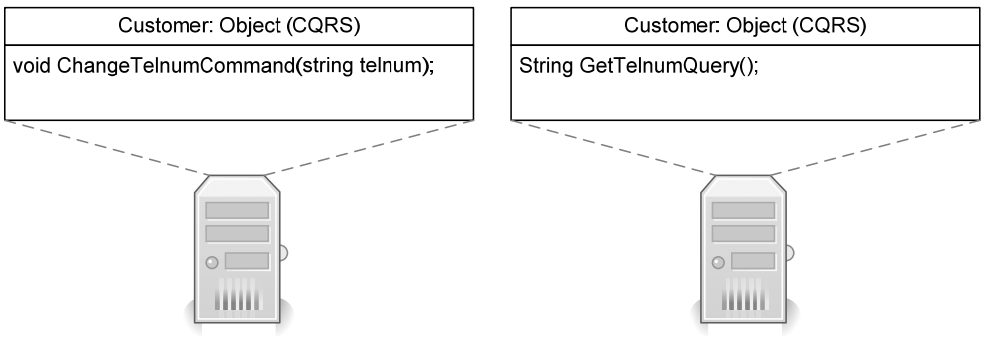

Figure 2.5 – An object constructed according to CQRS

Please notice that CQS is broadly applicable, so you should try to use it whenever you can. CQRS, on the other hand, exists to solve only a specific problem. It is not broadly applicable.

2.4.2 Collaborative Domains

The problem that CQRS is good at solving is a problem of blocking the user when locking the data. This kind of problem is what Udi Dahan and Greg Young have called a collaborative domain (4). An inherent property of the domain is that a large set of actors (such as people or other parts of the system) operate in parallel on the same small set of data.

Whenever this scenario occurs, you run the risk of forcing the users to try and then retry their operations until they get the desired result (14). The CQRS is especially beneficial where the collaboration causes complex decisions about what the outcome should be when there are many actors operating on the same, shared data.

2.4.3 Context and Problem

In traditional data management systems, both commands and queries are executed against the same set of entities in a single data repository. These entities may be a subset of the rows in one or more tables in a relational database such as SQL Server.

All CRUD operations are applied to the same representation of the entity (7). For instance, the data access layer (DAL) retrieves a data transfer object (DTO) from the data store and displays on the screen. After a user updates some DTO fields, the DAL will save the DTO back in the data store. Therefore, this same DTO is utilized for both read and write operations as shown in Figure 2.6.

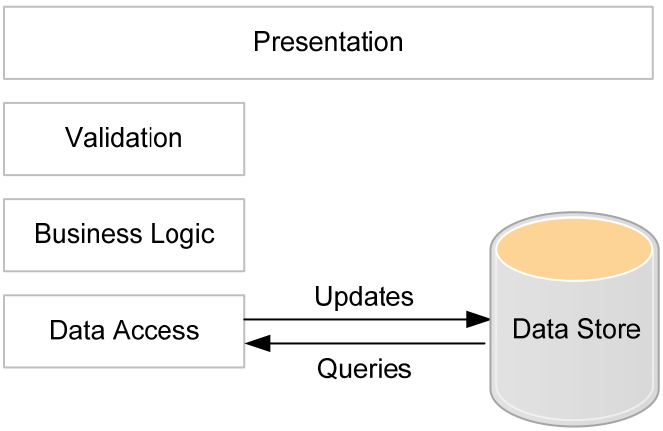

Figure 2.6 – CRUD N-tier architecture

In contrast, the CQRS pattern uses separate interfaces to segregate the process that write data from the process that update data (15). This characteristic makes the querying data models and updating data models isolated from each other as shown in Figure 2.7.

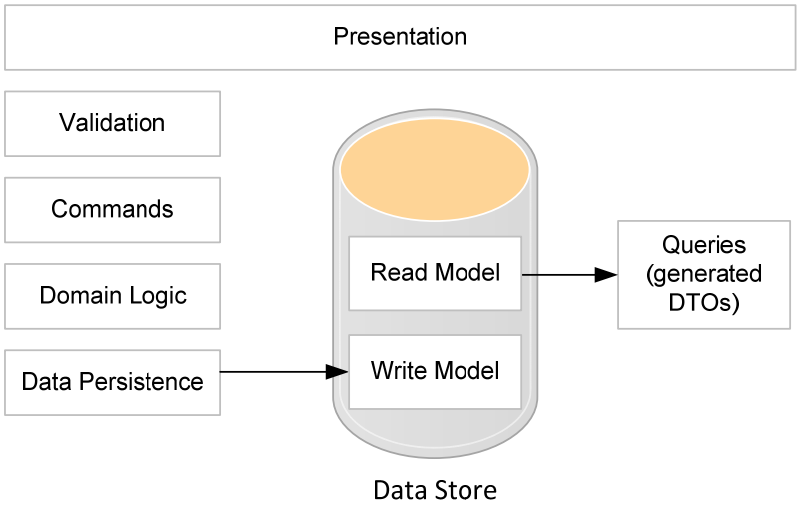

Figure 2.7 – A basic CQRS architecture

Compared with the single model of the data that exists in CRUD-based systems, the use of separate query and update models in CQRS considerably simplifies design and implementation. However, one disadvantage is that CQRS code cannot be automatically generated using scaffold mechanisms.

2.5 Related Works

CQRS pattern concept is very new in the industry. Its popularity as software implementation technique is still at an early stage. Many online articles are found arguing in favor of using CQRS. The online articles written by Martin Fowler (16), Udi Dahan (17), and Greg Young (1) are considered key knowledge sources in regards to this area. The most important definitions along with guidance and best practice are given here. We used (2, 4, 7, 8, 9) as a beginning point in our research.

Since CQRS is often implemented based on Domain-Driven Design (DDD) methodology, we found that Eric Evan's book (11) is a great knowledge source providing a helpful introduction into the area of DDD. Although it does not directly affect the outcomes of our study, it provides a strong background as a context in which a CQRS application would work.

We also found several fascinating examples showing different areas where CQRS has been successfully applied. In (19) the CQRS pattern is a part of how to be more efficient for developing web applications. This paper describes a solution for getting rid of the underlying challenges associating with object-relational impedance mismatch and the consequences of CAP theorem when scaling out, by applying CQRS pattern.

Thesis (20) was important to us, because we could found a detailed description of the process of building code within CQRS framework. The author provided an overview on a visual CQRS workbench used to define the model that used for generating code. Only parts of the code that should be written manually in the system are event handlers on the query/read side.

CHAPTER III
METHODOLOGY

This chapter discusses the methodology used in our research. We began with building a proof of concept product purchase and inventory using C#.NET programming language. We did it by modifying the source code written by Michael Perry (https://github.com/michaellperry/pharmanet) correctly to satisfy Entity Framework Code First, DDD and unit testing. Moreover, we separated it into two different versions:

- CRUD N-tier applied to every bounded context, in which we model the problem as rows and columns in the database and then write CRUD operations to manage that data
- CQRS pattern applied to our selected collaborative domain only.

3.1 CRUD N-tier: Design and Coding

Starting with CRUD N-tier, we carve the domain into bounded contexts. Each bounded context is a separate solution in Microsoft Visual Studio 2013. Within each solution the projects are organized into layers, which are Domain, Application, and Presentation. The Domain layer contains the model. It has all value objects and entities of the problem domain.

Above the Domain model, we have the Application layer containing all of the business services. At the topmost is the Presentation layer. This layer exposes all of the services and domain objects to the outside users.

Next step, we must look for a bounded context in which many users are working together on a small set of data. Moreover, finally we choose the OrderProcessing subsystem as a collaborative domain because it supports a large number of customers placing orders simultaneously while all those orders have to be picked.

Right now the subsystem (or a solution in Visual Studio) is not yet using CQRS. Let's see what problems OrderProcessing has without CQRS. This solution consists of three domain-specific layers, which are Domain, Application, and Presentation. The Domain layer captures the objects of the domain. In this case, OrderProcessing.Domain has customers (Listing 3.1) that are placing orders. Each line of an order has a certain quantity of some product (Listing 3.2), and then we turn these OrderLines (Listing 3.3) into PickLists (Listing 3.4).

```
namespace CRUDApp.OrderProcessing.Domain
{
    public class Customer
    {
        public int Id { get; set; }
        public string Name { get; set;  }
        public string ShippingAddress { get; set; }
    }
}
```

Listing 3.1 – Customer class

```
namespace CRUDApp.OrderProcessing.Domain
{
    public class Product
    {
        public int ProductId { get; set; }
        public int ProductNumber { get; set; }
    }
}
```

Listing 3.2 – Product class

```
namespace CRUDApp.OrderProcessing.Domain
{
    public class OrderLine
    {
        public Customer Customer { get; set; }
        public Product Product { get; set; }
        public int Quantity { get; set; }
    }
}
```

Listing 3.3 – OrderLine class

```
namespace CRUDApp.OrderProcessing.Domain
{
    public class PickList
    {
        public virtual int PickListId { get; set; }
        public virtual Guid OrderId { get; set; }
        public Warehouse Warehouse { get; set; }
        public Product Product { get; set; }
        public int Quantity { get; set; }
    }
}
```

Listing 3.4 – PickList class

A PickList is instructions for people in a warehouse to pick and ship a certain quantity of the product. Moreover, then finally, we keep track of the inventory (Listing 3.5) of each warehouse (Listing 3.6).

```
namespace CRUDApp.OrderProcessing.Domain
{
    public class Inventory
    {
        [Key, Column(Order = 1)]
        public int WarehouseId { get; set; }
        public virtual Warehouse Warehouse { get; set; }

        [Key, Column(Order = 2)]
        public int ProductId { get; set; }
        public virtual Product Product { get; set; }

        public int QuantityOnHand { get; set; }
    }
}
```

Listing 3.5 – Inventory class

```
namespace CRUDApp.OrderProcessing.Domain
{
    public class Warehouse
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public virtual IList<Inventory>
          Inventory { get; set; }
    }
}
```

Listing 3.6 – Warehouse class

Inventory is a certain quantity of the product within a warehouse. The goal of the OrderProcessing bounded context is to figure out how to select the products from the warehouses in order to satisfy the orders, which is the responsibility of the Application layer. It composes of an AllocateInventory method (Listing 3.7) under InventoryAllocationService to allocate the inventory of a set of orderLines, locate the products, and pick the product by adjusting the quantity on hand and generates a PickList.

```
public List<PickList> AllocateInventory
                      (List<OrderLine> orderLines)
 {
         List<PickList> pickLists = new List<PickList>();
         foreach (var orderLine in orderLines)
         {
             Warehouse warehouse = LocateProduct(
                 orderLine.Product,
                 orderLine.Quantity);

             if (warehouse != null)
             {
                 PickList picklist = PickProduct(
                     orderLine.Product,
                     orderLine.Quantity,
                     warehouse);
                 pickLists.Add(picklist);
             }
         }
         return pickLists;
 }
```

Listing 3.7 – AllocateInventory method

The PickList is then saved to the database and then returned to the caller, which is in the Presentation layer. This topmost layer presents the application to the outside consumer through a WCF service. The OrderProcessingService (Listing 3.8) picks up the application services that it needs such as the InventoryAllocationService. It gives each of these services the repositories that it needs. Here is a kind of dependency injection implementation.

```
public OrderProcessingService()
{
     OrderProcessingDB.Initialize();
     OrderProcessingmentDB context = new OrderProcessingDB();

     _customerService = new CustomerService(context.GetCustomerRepository());
     _productService = new ProductService(context.GetProductRepository());
     _warehouseRepository = context.GetWarehouseRepository();
     _inventoryAllocationService
                      = new InventoryAllocationService(_warehouseRepository);
}
```

Listing 3.8 – OrderProcessingService

Next, the WCF PlaceOrder method (Listing 3.9) takes an Order type, and this is a DataContract (Listing 3.10) together with a ServiceContract (Listing 3.11).

```
public Confirmation PlaceOrder(Order order)
{
  Customer customer = _customerService.GetCustomer(order.CustomerName,
                                                    order.CustomerAddress);
  List<OrderLine> orderLines = order.Lines
                .Select(line => new OrderLine
                {
                    Customer = customer,
                    Product = _productService.GetProduct(line.ProductNumber),
                    Quantity = line.Quantity
                })
                .ToList();
  List<PickList> pickLists =
                 _inventoryAllocationService.AllocateInventory(orderLines);
  _warehouseRepository.SaveChanges();

            return new Confirmation
            {
                Shipments = pickLists
                    .Select(pickList => new Shipment
                    {
                        ProductId = pickList.Product.ProductId,
                        Quantity = pickList.Quantity,
                        TrackingNumber = "123-456"
                    })
                    .ToList()
            };
}
```

Listing 3.9 – WCF PlaceOrder method

```
namespace CRUDApp.OrderProcessing.Contract
{
    [DataContract]
    public class Order
    {
        [DataMember]
        public Guid OrderId { get; set; }

        [DataMember]
        public string CustomerName { get; set; }

        [DataMember]
        public string CustomerAddress { get; set; }

        [DataMember]
        public List<Line> Lines { get; set; }
    }
}
```

Listing 3.10 – WCF DataContract

```
namespace CRUDApp.OrderProcessing.Contract
{
    [ServiceContract]
    public interface IFulfillmentService
    {
        [OperationContract]
        Confirmation PlaceOrder(Order composite);
    }
}
```

Listing 3.11 – WCF ServiceContract

This is the data structure that is optimized for sending messages between the consumers and the service, which isn't the same thing as the objects in the domain. Those objects are optimized for representing business logic in the repository.

The PlaceOrder service in Listing 3.9 needs to translate from the DataContract into the domain. So the first thing it does is to translate the customer information that it pulls off of the wire into a real customer object. So, it grabs this

information, the name and address, and passes that to the _customerService to do the work.

Then it does something similar for each OrderLine with the product. So, it takes the ProductNumber, and then it is passed to the _productService to look up the product object. By the time it's finished, it has translated an order object, which is made up entirely of WCF data contracts into a list of orderLines, which are composed solely of domain objects.

After finish _inventoryAllocationService operation, the same process has to happen in reverse. We have to convert domain objects into data contracts. Thus, PlaceOrder turns the list of orderLines, which are domain object back into WCF DataContract objects to be returned to the caller. Next, we have to traverse all the way down our stack starting at the Presentation layer, Application layer, and Domain layer.

Next, there is an implementation of the repositories (Listing 3.12) in the SQL project using Entity Framework. All they do is setting up a DbContext, and it has a DbSet for each aggregate root, which is the top-level entity of an aggregate. It is the only entity in the aggregate that has a global identity and can, therefore, be queried. For other entities in the aggregate, we do not create DbSets for them since we cannot query it.

```
public DbSet<Customer> Customers { get; set; }
public DbSet<Warehouse> Warehouses { get; set; }
public DbSet<Product> Products { get; set; }

public IRepository<Customer> GetCustomerRepository()
{
    return new SQLRepository<Customer>(this, Customers);
}

public IRepository<Warehouse> GetWarehouseRepository()
{
    return new SQLRepository<Warehouse>(this, Warehouses);
}

public IRepository<Product> GetProductRepository()
{
    return new SQLRepository<Product>(this, Products);
}
```

Listing 3.12 – The implementation of repositories

Moreover, we have a simple method that will turn each of the aggregate root DbSets into a repository using the SQLRepository class (Listing 3.13). It is just another implementation of IRepository and is done inside the Infrastructure solution. This solution is based on DbSet and DbContext, so it uses Entity Framework to implement IRepository.

```
public class SQLRepository<T> : IRepository<T>
        where T : class
    {
        private readonly DbContext _context;
        private readonly DbSet<T> _dbSet;

        public SQLRepository(DbContext context, DbSet<T> dbSet)
        {
            _context = context;
            _dbSet = dbSet;
        }

        public T Add(T item)
        {
            _dbSet.Add(item);
            return item;
        }

        public IQueryable<T> GetAll()
        {
            return _dbSet;
        }

        public void Remove(T item)
        {
            _dbSet.Remove(item);
        }

        public void SaveChanges()
        {
            _context.SaveChanges();
        }
    }
```

Listing 3.13 – SQLRepository class

The last project is the Consumer (Listing 3.14). It is a console application that places an order for the same product in a tight loop, by referencing the WCF service contract interfaces and DataContract classes. It prints out the confirmation and time elapsed for each loop so that we can see what's happening.

```
static void Main(string[] args)
        {
            ServiceClient<IOrderProcessingService> client = new
                                    ServiceClient<IOrderProcessingService>();
            var order = new Order
            {
                CustomerName = "Vitu Hansakul",
                CustomerAddress = "316 Serene Park",
                Lines = new List<Line>
                {
                    new Line { ProductNumber = 67890, Quantity = 2 }
                }
            };

            while (true)
            {
                try
                {
                    PlaceOrder(client, order);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }

        private static void PlaceOrder(ServiceClient<IOrderProcessingService>
                                        client, Order order)
        {
            var confirmation =
                client.CallService("BasicHttpBinding_IOrderProcessingService",
                            s => s.PlaceOrder(order));

            String.Format("Confirmed {0} shipments:",
                          confirmation.Shipments.Count);

            foreach (var shipment in confirmation.Shipments)
            {
               Console.WriteLine(String.Format("{0} {1}: {2}",
               shipment.Quantity, shipment.ProductId, shipment.TrackingNumber));
            }
        }
    }
```

Listing 3.14 – Contents in the Consumer project

3.2 CQRS: Design and Coding

Next, we modified the software architecture for OrderProcessing collaborative domain to CQRS version as shown in Figure 3.1.

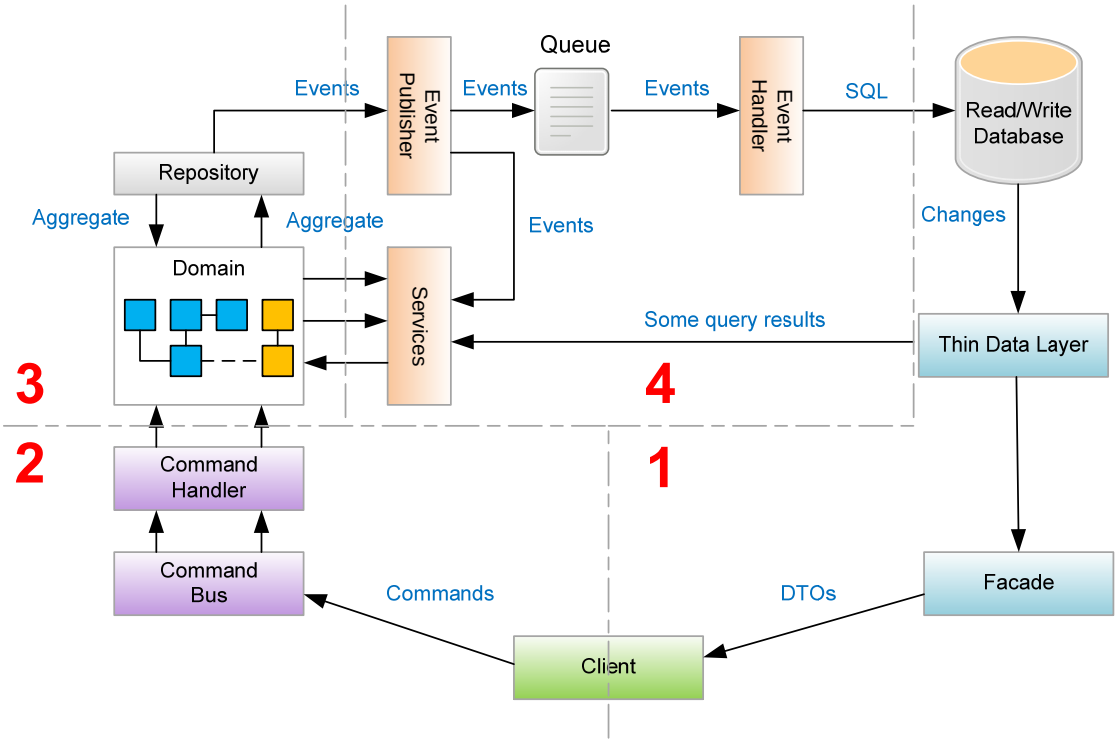

Figure 3.1 – Overall architecture of our sample CQRS application

This architecture is splitted into four parts: 1. Queries, 2. Commands, 3. Internal Events and 4. External Events.

We started modifying this solution by changing the OrderProcessing service interface from Listing 3.11 to Listing 3.15 so that PlaceOrder returns void.

This code block allows us to call PlaceOrder without processing the order. Then the client can call back and check the order status. Now, they are passing in this orderId that come from the modified WCF DataContract (Listing 3.16).

```
[ServiceContract]
    public interface IFulfillmentService
    {
        [OperationContract]
        void PlaceOrder(Order order);

        [OperationContract]
        Confirmation CheckOrderStatus(Guid orderId);
    }
```

Listing 3.15 – Modified WCF ServiceContract

```
[DataContract]
    public class Order
    {
        [DataMember]
        public Guid OrderId { get; set; }

        [DataMember]
        public DateTime OrderDate { get; set; }

        [DataMember]
        public string CustomerName { get; set; }

        [DataMember]
        public string CustomerAddress { get; set; }

        [DataMember]
        public List<Line> Lines { get; set; }
    }
```

Listing 3.16 – Modified WCF DataContract

The Order class shown above is something that the client, not the server, allocates when it creates an order, and that is an important distinction. The point is for the client to do all the work that it needs to and then just pass it off the server without expecting any result back, so no processing on the server-side.

Now we made the same change to the server-side WCF PlaceOrder method (Listing 3.17) to enable it implements the right interface, so it would return void. Before the modification, it processed the order inside of PlaceOrder method and returned the confirmation but we do not want to do that.

```
public void PlaceOrder(Order order)
 {
      _messageQueue.Send(new Messages.PlaceOrder
      {
          OrderId = order.OrderId,
          OrderDate = order.OrderDate,
          CustomerName = order.CustomerName,
          CustomerAddress = order.CustomerAddress,
          Lines = order.Lines.Select(l =>
              new Messages.Line
              {
                  ProductNumber = l.ProductNumber,
                  Quantity = l.Quantity
              })
              .ToList()
      });
  }
```

Listing 3.17 – Modified WCF PlaceOrder method

We instead put the Order into a MessageQueue to send the order to the queue, so the OrderHandler can process it later. This handler spins in a tight loop until it is told to stop, and it receives orders from the queue, processes them, puts PickLists back into the database. So that means we need to finish the contract by adding CheckOrderStatus method (Listing 3.18) to look for the matching PickLists by orderId and return PickLists.

```
public Confirmation CheckOrderStatus(Guid orderId)
        {
            var pickLists = _pickListService
                .GetPickLists(orderId);

            if (!pickLists.Any())
                return null;

            return new Confirmation
            {
                Shipments = pickLists
                    .Select(pickList => new Shipment
                    {
                        ProductId = pickList.Product
                            .Id,
                        Quantity = pickList.Quantity,
                        TrackingNumber = "123-456"
                    })
                    .ToList()
            };
        }
```

Listing 3.18 – CheckOrderStatus method

Next step, OrderHandler will process the order and save the PickLists to the database. Moreover, when consumer calls back and checks the order status, we just get those PickLists out of the database and turn them into a Confirmation.

Once we have got the server modified to implement the new contract, we made a change to the Consumer. What it used to do is allocating order IDs, placing those orders, getting back a confirmation from PlaceOrder call and then print that confirmation. However, after the code modification, the PlaceOrder does not return a confirmation anymore. The client now needs to come back later and check the order status, so we have to modify the code to save the orderIds placed into the list.

In order to check the order status, we need to make another service call CheckOrderStatus. It takes an orderId as input and return the confirmation. Returning not null means the order has been processed. We will loop through all of the orderIds, look for ones where CheckOrderStatus return true, record the orderIds to processOrderIds, remove them from the list of pending orders.

As we loop though here we are no longer going to be checking the order status of those processed IDs, we just come back, place another order, and then begin checking the order status of that new order. Thus, this change to the Consumer client makes it not expect the results immediately because it can come back later and check.

3.3 Scalability Performance Testing

During this stage, we ran the performance test to measure response times of our proof-of-concept application both CRUD N-tier and CQRS versions. We used Visual Studio Load Test to simulate different numbers of customers placing order through the WCF consumer service using PlaceOrder method, and added additional monitoring code for recording iming information for detailed analysis.

We created the performance test environment in Windows 7 machine, then started the service and triggered a customer instance to run simultaneously and locally from one to five instances, step up by one. For each instance, we set the PlaceOrder method to run 100 times to order the product two units at a time. The results obtained will be average response times or average times that our system took to return notification on the screen, for each iteration, that it finished adjusting both our product quantity on hand and number of PickList in the SQL database table.

3.4 Analyze the Impact to Businesses

From our scalability performance testing results, we will summarize the pros and cons brought about by using CQRS to implement a bounded context.

3.5 Develop a Decision Criteria

Moreover, we are going to use our lesson learned along the journey to make a decision criteria whether or not you should implement the CQRS pattern in bounded contexts in your next mission-critical applications.

CHAPTER IV
RESULT

From the scalability performance test results shown in Table 4.1 and Figure 4.1, they implicitly show us how the systems handled concurrency. While we added more instances, every instance were trying to update the same inventory record at the same time. So, they took a read lock on the record to get the current inventory, and then tried to escalate to a write lock in order to update it. Now, you can't escalate to a write lock while another reader still holds the record, and neither one of these readers is going to give it up. That made the response times increased with more concurrent instances.

Table 4.1 – Raw performance test results in seconds

1 Instance (100 iterations)										
Pattern	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
CRUD	8.149	8.197	8.107	8.247	8.287	8.096	7.888	7.943	8.0452	8.058
CQRS	1.718	1.621	1.844	1.591	2.011	1.672	1.728	1.626	1.66	1.742

2 Instance (100 iterations each)										
Pattern	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
CRUD	14.355	14.132	14.259	14.554	14.237	14.187	14.36	14.254	14.144	14.526
CQRS	2.629	2.6345	2.535	2.5755	2.7955	2.6705	2.549	2.622	2.611	2.7725

3 Instance (100 iterations each)										
Pattern	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
CRUD	20.632	21.393	20.908	20.438	20.693	20.638	21.017	21.336	21.159	21.066
CQRS	3.403	3.4557	3.001	3.5757	3.4087	3.3817	3.926	3.8157	3.7157	3.415

4 Instance (100 iterations each)										
Pattern	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
CRUD	28.208	27.783	27.926	27.593	27.338	28.002	28.265	28.054	28.085	27.793
CQRS	4.0933	4.6988	4.008	4.364	4.5673	4.3853	4.1158	4.1328	3.817	4.1808

5 Instance (100 iterations each)										
Pattern	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
CRUD	34.864	34.534	34.335	34.411	34.853	34.726	34.698	34.567	34.298	34.736
CQRS	5.359	5.1094	5.3156	5.328	4.949	5.0996	5.0256	5.2906	5.1306	5.1258

Table 4.2 – Average performance test results

Concurrent Consumer Instances	Avg. Response Time per Transaction in msec (CRUD N-tier)	Avg. Response Time per Transaction in msec (CQRS)
1	81.02 ± 1.26	17.21 ± 1.25
2	143.01 ± 1.47	26.39 ± 0.86
3	209.28 ± 3.22	35.10 ± 2.63
4	279.05 ± 2.85	42.36 ± 2.66
5	346.02 ± 2.05	51.73 ± 1.41

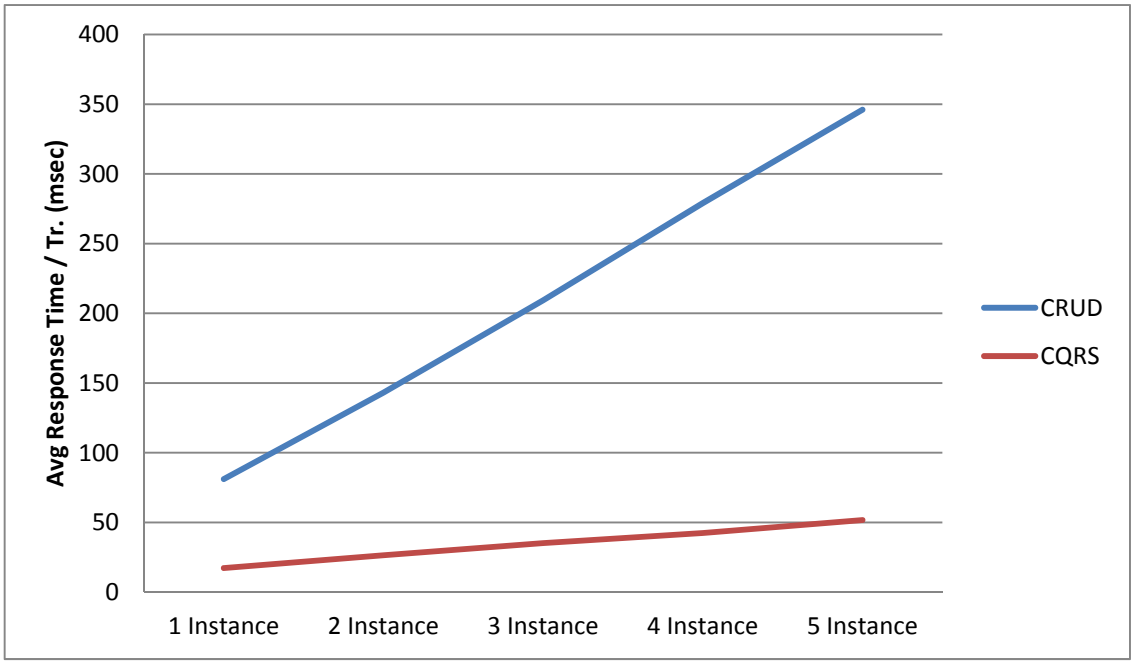

Figure 4.1 – Average performance graphs

As we expected, the application built in CQRS version yielded better average response times than the CRUD N-tier. The problem of the CRUD version is that we are trying to block the user while we are locking the record. The locking is a necessary part of maintaining data integrity in a relational database, but the problem is that we are forcing the user to suffer that lock. Moreover, the thing that is forcing our hand is a service contract. This contract says that we are going to return a tracking number for the products that you just ordered, and demand us to allocate the inventory during the customer waits.

Consider the WCF contract in Listing 3.11, this method changes the system's state and return a result. Covered up inside this result is a tracking number. To process an order and get a tracking number, the OrderProcessing system needs to:

- Locate inventory
- Allocate that inventory to this order
- Determine the shipment method
- Allocate space on a pallet
- Obtain a tracking number

Since the result is returned back from a method, these steps must be performed synchronously. The caller will hold up while those steps are performed before they realize that their request has even been submitted

For the CQRS adaptation, the initial step in amending this issue is to divide the Placeorder command from the query CheckOrderStatus as indicated in Listing 3.15. We additionally changed the OrderProcessing service interface with the goal that Placeorder returns void.

Now the service can output the result quickly after finish storing the incoming order. It does not need to experience the greater part of steps before telling the caller that the order was gotten. The caller can return back and check the status of the request by running a query.

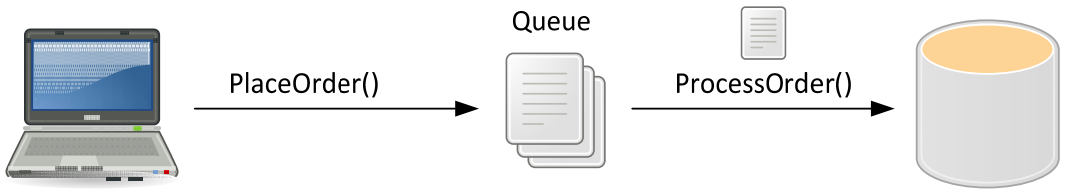

Figure 4.2 – MessageQueue

Instead of processing the order inside of PlaceOrder method and then returning the confirmation, we just put the order into a MessageQueue (Figure 6.2) so that it can be processed later without having the users to wait. Therefore, we are not blocking the users while we are locking the data. We have funneled all of that into another background process that is doing all of the locking and ensuring the consistency of the system. That makes our process a lot easier to scale.

CHAPTER V
DISCUSSION

This chapter summarizes the findings from our study. It highlights the most significant lesson we learned along the way.

5.1 Performance

At the start of our research, one of our ideas about the CQRS pattern was that by differentiating the read and write models of the application we can optimize each for performance. The idea was borne out in real practice during our study, and we benefited significantly from this partition when we did need to improve the performance.

When we found the bottlenecks, fixing them ended up being simple due to the way the CQRS empowers you to separate different components of the system, for example, reads and writes. Although the separation of concerns which results from implementing the CQRS can make it harder to recognize an issue, once you have recognized one, it is simpler to fix, as well as less demanding to prevent its return. The decoupled design also makes it less complicated to compose unit tests.

5.2 System Complexity

Implementing the CQRS pattern is far more complicated than implementing a traditional CRUD N-tier system. For this research, there was additionally the overhead of looking into CQRS for the first time, and building an asynchronous messaging infrastructure.

Our experiences during the study have apparently confirmed to us why the CQRS cannot be a top-level architecture. You must make sure that expenses occurred from implementing a CQRS-based bounded context with this high level of intricacy

are worth it. In general, it is in collaborative domains that you will see the benefits from CQRS.

5.3 Impact to Business

From our scalability performance test results, we can summarize the benefits brought about by using CQRS to implement a bounded context. Overall, we think that their impact on the solution, though, is dramatic to business.

Scalability has numerous faces and factors; the characteristic for scalability has a tendency to be particular for every system you are considering. In general, scalability defines the system's capability to keep the same level of performance as the number of users increases. A system with more users will perform certain operations more often while scalability depends on the margins that you need to alter the system to make it executes more operations in the same amount of time.

The approach to attain the scalability relies on the type of operations most commonly performed. In the case reads are the most prevalent operation, you can apply caching techniques to decrease the number of accesses to the database. If writes are sufficient to slow down the system at peak hours, you may need to consider changing from a classic synchronous writing model to asynchronous writes or even commands queues. Segregating commands from queries enables you to take a shot at the scalability parts of both models in total isolation.

An illustration of what it intends to treat reads and writes independently is given by a cloud platform, for example, Microsoft Azure. You can implement command and query models as different Azure web or worker roles and scale them autonomously in term of instances.

Similarly, when reads are the most prevalent, you can make a decision to offload some pages to another server with a thick layer of caching on top. The capability to manipulate command and query model separately is invaluable. You can get the OPEX cheaper when implementing CQRS on the cloud since the solution needs less web/worker roles to handle the users.

Moreover, if you use an approach like CRUD N-tier, then the technology has a tendency to shape the solution. Embracing the CQRS helps you to concentrate

on the business and build task-oriented UIs. Dividing the distinctive concerns into the both sides is a solution that is more flexible in the face of changing business environments. This results in less development and support costs in the long term.

5.4 Operation Cost Reduction

Suppose we meet with a food raw material supplier looking to upgrade their online order processing system. At peak times, they need to handle five simultaneous customers placing ten orders/second/customer for a sustained period of 10 minutes. That does not seem like many scales, but they require all orders to complete successfully, they cannot allow any dropped requests or connection timeouts as that would leave customers gazing at an hourglass.

Therefore, how much Azure kit you need to support this requirement?

- **Option 1 – Synchronous End-to-End**

From Table 6.2, average response time of the CRUD N-tier versus is 346.02 msec per transaction for five simultaneous consumer instances. They can be translated to around two orders/second/customer. Thus, we need five basic-package Web Role Instance to host the WCF server-side service which would cost around 5 x 112 USD/month = 560 USD/month (as of November 2014.)

- **Option 2 – Client-Driven CQRS**

From Table 6.2, average response time of the CQRS version is 51.73 msec per transaction for five concurrent consumer instances. They can be translated to around 19 orders/second/customer. Thus, we need only one basic Web Role Instance to host the WCF server-side service which would cost around 112 USD/month (as of November 2014.)

So the option 2 is a beautiful, clear winner with using just one Web Role plus all the infrastructure that places behind, indeed.

Note: for more information about Microsoft Azure's current pricing details, see http://azure.microsoft.com/en-us/pricing/details/websites.

CHAPTER VI
CONCLUSION

As we see things, CQRS is the most valuable pattern today for building about any applications. CQRS is about utilizing separate models for commands and queries. It is similar to having two simple domain models rather than only one that deals with both viewpoints in this way increasing complexity.

The query stack of a CQRS is straightforward and comprises only a simple query layer on top of LINQ or ADO.NET. Instead, the command stack regularly expresses the business logic in tasks and actualizes them through commands, events and their synchronization.

Now we would like to recommend the criteria you must take to assess whether you should implement the CQRS in bounded contexts of your mission-critical application or not. The greater amount of these inquiries you can answer positively, the more probable it is that applying the CQRS will benefit your solution:

- Does the bounded context implement a range of business handiness that is a key differentiator in your market sector?
- Is the bounded context collaborative in nature with components that are liable to have a considerable amount of contention at run time?
- Is the bounded context prone to experience steadily changing business rules?
- Is scalability one of the difficulties confronting this bounded context?
- Is the business logic in the bounded context complicated?

Anyway, we do not think CQRS has any significant drawbacks except the initial development cost and overhead of implementing the pattern. Everything depends on what you mean exactly by applying CQRS to your software architecture. It is a design pattern that recommends you have two different layers: one loaded with the model and services essential for reading and one with the model and services for

writing. Whatever a layer is - a library of functions, an object model, or a group of DTOs – Its is up to your implementation detail.

With the above definition in place, about any system can benefit from CQRS and coding. It does not require you to do things in a different way. Neither does it mean learning anything new and frightening.

However, The point is that CQRS pattern will lead to some deeper architectural changes, especially in the collaborative domains. It requires you to learn and perform a preliminary analysis to achieve the maximum return in terms of scalability. In the end, CQRS will be discovered by software architects who are searching for more efficient approach to supporting complex mission-critical applications in which multiple actors operate on the data simultaneously and ever-changing and sophisticated business rules apply.

REFERENCES

1 Young, G. *CQRS Documents*. [Online]. [Accessed 22 May 2014]. Available from: http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.

2 Torre, C. and Carmona, D. *.NET Technology Guide for Business Applications.* Washington: Microsoft Press, 2013.

3 Microsoft. *Command and Query Responsibility Segregation (CQRS) Pattern*. [Online]. [Accessed 8 September 2014]. Available from: http://msdn. microsoft.com/ en-us/library/dn568103.aspx.

4 Betts, D. et al. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft, 2013.

5 Microsoft Patterns & Practices Team. *Microsoft Application Architecture Guide*. 2nd ed. Microsoft Press, 2009.

6 Chakrabarty, I. *Exam Ref 70-484: Essentials of Developing Windows Store Apps Using C#*. California: O'Reilly Media, 2013.

7 Vernon, V. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.

8 Palermo, J. *The Onion Architecture : part 1*. [Online]. 2008. [Accessed 8 September 2014]. Available from: http://jeffreypalermo.com/blog/the-onion-architecture-part-1.

9 Brewer, E. *Towards robust distributed systems.* Symposium on Principles of Distributed Computing (PODC 2000), Portland, OR, 19 July 2000.

10 Gilbert, S. and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*. 2008, **33**(2), pp. 51–59.

11 Evans, E. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

12 Meyer, B. *Object-Oriented Software Construction*. 2nd ed. New York: Prentice Hall PTR, 1988.

13 Martin, R. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

14 Dahan, U. *Why you should be using CQRS almost everywhere....* [Online]. 2011. [Accessed 8 September 2014]. Available from: http://www.udidahan.com/2011/10/02/why-you-should-be-using-cqrs-almost-everywhere.

15 Microsoft. *Command and Query Responsibility Segregation (CQRS) Pattern*. [Online]. [Accessed 8 September 2014]. Available from: http://msdn.microsoft.com/en-us/library/dn568103.aspx.

16 Fowler, M. *CQRS*. [Online]. [Accessed 29 October 2014]. Available from: http://martinfowler.com/bliki/CQRS.html.

17 Dahan, U. *Clarified CQRS*. [Online]. [Accessed 9 December 2009]. Available from: http://www.udidahan.com/2009/12/09/clarified-cqrs.

18 Microsoft Developer Network. *CQRS Journey*. [Online]. 2012. [Accessed 29 October 2014]. Available from: http://msdn.microsoft.com/en-us/library/jj554200.aspx.

19 Hendrikse, Z. and Molkenboer, K. *A radically different approach to enterprise web application development*. [Online]. 2012. [Accessed 29 October 2014]. Available from: http://www.codeboys.nl/white-paper.pdf.

20. Fitzgerald, S. *State Machine Design, Persistence and Code Generation using a Visual Workbench, Event Sourcing and CQRS*. M.Sc. thesis, University College Dublin, 2012.

BIOGRAPHY

NAME	Mr. Vitu Hansakul
DATE OF BIRTH	25 APRIL 1979
PLACE OF BIRTH	Bangkok, Thailand
INSTITUTIONS ATTENDED	Kasetsart University, 2000-2004 Bachelor of Engineering (Electrical Engineering) Mahidol University, 2012-2014 Master of Science (Technology of Information System Management)
HOME ADDRESS	316/34 SERENE PARK 1 VILLAGE, WONGSAWANG RD., BANGSUE, BANGKOK Tel. 668-6624-9128 E-mail : vitu888@gmail.com