Abstract and cover page

Table of Contents

[1 Introduction 1](#_Toc390200228)

[2 Design Methodology 1](#_Toc390200229)

[3 Domain Driven Design 1](#_Toc390200230)

[3.1 Components of Domain Driven Design 1](#_Toc390200231)

[3.1.1 Domains/Core Domain 1](#_Toc390200232)

[3.1.2 Sub Domains 1](#_Toc390200233)

[3.1.3 Generic Sub Domains 1](#_Toc390200234)

[3.1.4 Bounded Contexts 2](#_Toc390200235)

[3.1.5 Aggregates, Entities and Value Objects 2](#_Toc390200236)

[3.1.6 Repositories 2](#_Toc390200237)

[4 The Strategic Vision and Design of APS 2](#_Toc390200238)

[4.1 APS Domains and Responsibility Decomposition 2](#_Toc390200239)

[4.1.1 Core Domain 2](#_Toc390200240)

[4.1.2 Sub Domains 2](#_Toc390200241)

[4.1.3 Generic Sub Domains 2](#_Toc390200242)

[4.1.4 Integrating the different domains 2](#_Toc390200243)

[4.1.5 Queries, Single Responsibility Principle and Interface Segregation principle 3](#_Toc390200244)

[4.1.6 CQS? 3](#_Toc390200245)

[5 Tactical Design 3](#_Toc390200246)

[5.1 Domain and feature allocation 3](#_Toc390200247)

[5.2 Project structure 3](#_Toc390200248)

[5.2.1 Unit Tests 4](#_Toc390200249)

[5.2.2 Fakes 4](#_Toc390200250)

[5.2.3 Domain Models 4](#_Toc390200251)

[5.2.4 Application Services 4](#_Toc390200252)

[5.2.5 Integration and Published Language 5](#_Toc390200253)

[5.2.6 Common Classes 5](#_Toc390200254)

[5.3 Continuous Integration and Development Strategy 5](#_Toc390200255)

[5.4 Specifications by example and Test Driven Development 5](#_Toc390200256)

[5.5 Feature integration specification collaboration 5](#_Toc390200257)

[5.6 Object Orientation Principles 5](#_Toc390200258)

[5.6.1 Single Responsibility Principle 5](#_Toc390200259)

[5.6.2 Dependency Inversion Principle 6](#_Toc390200260)

[5.6.3 Interface Segregation Principle 6](#_Toc390200261)

[5.7 Pair Programming and Code Reviews 7](#_Toc390200262)

[6 Challenges and successes 7](#_Toc390200263)

[7 Conclusion 7](#_Toc390200264)

[8 References 8](#_Toc390200265)

[9 Appendix 9](#_Toc390200266)

[A - Definition of terms or concepts used within the APS system: 9](#_Toc390200267)

[B – Domain Integration Diagram 10](#_Toc390200268)

[C – Code Listing 1 – Example Query 11](#_Toc390200269)

[D - Code Listing 2 – Repository Interface 12](#_Toc390200270)

# Introduction

The report that follows is a high-level description and evaluation of the prototype implementation for the ELEN7045 Group Project [1] as designed and implemented by Group two (G2) (See Appendix E – Group Members).

The description that follows, will detail the series of design decisions and the process followed by G2 that is the basis of their Design Methodology.

Following on from their Design Methodology, a brief explanation and cursory introduction to Domain Driven Development (DDD) [2] will be provided, so as to contextualise some of the reasoning behind design decisions made.

G2 will show that by first following a Strategic and then a Tactical design approach [3], the solution prototype, provides a robust, extensible and maintainable solution by means of its Test Driven Development (TDD) [4] implementation.

The difference between Strategic and Tactical design decisions is pertinent to G2’s design and the differences are understood to be as quoted below from the article by Amnon H. Eden on how *they* separate the two:

*We seek to distinguish Strategic design decisions (e.g., to adopt a programming paradigm, architectural style, CBSE standard or application framework) from tactical design decisions (e.g. to use a design pattern, refactoring or programming idiom). This distinction is important since strategic statements carry far-reaching implications over the implementation and therefore must be made early in the development process, whereas tactical statements have localized effect and must be deferred to a later stage in the process.* [3]

The DDD paradigm will be focused on to highlight the Strategic design that led to the overarching solution design that provided the basis for the implementation methodology.

Within the Tactical design, work allocation, the Version Control [4] mechanism, project structure, as well Object-Oriented Programming (OOP) [5] principles followed will be evaluated.

Lastly, some challenges, successes and potential enhancements will be discussed.

For the purposes of this report, the members of G2 are assumed to be the ‘APS Solution Owners’ and are the subject matter experts for the APS Solution (*Domain Experts* [2]) of the APS Solution because it was mentioned that *the project brief is intentionally vague* [1]*.*

# Design Methodology

To solve the problem or requirements of the APS Solution, it was necessary to look at the project brief decomposing it into high-level requirements, and then breaking those down further into smaller sets of requirements.

These requirements potentially could be differing systems, components and eventually at the lowest level; projects and code Classes [7].

As part of the highest level of the design, component interaction was also considered to be a design factor, as this would have future implications for extensibility.

The components that could be distinguished to operate independently were determined to be as follows; Billing Companies, Customers, the 3rd Party Web Scraper, Scrape Scheduling Service. Auditing functionality was determined to be a separate component that depended on component integration as it would log or keep and audit log of the changes within a component, as well as the interactions between the components. These will be discussed in detail in further sections.

As mentioned, component integration formed part of the design process methodology, a shared Publish-Subscribe Messaging Pattern [8] was decided on as it allowed the components to not be directly coupled to each other. This too will be discussed in more detail within later sections.

To further understand the requirements, each component was then analysed, by means of highlighting the actions and responsibilities within them, to allow the formation of more detailed specifications for TDD.

These specifications were used when further decomposing the requirements, and contributed to the identification of Classes, their interactions and responsibilities. Class Responsibility and Collaboration (CRC) Cards [9] were used in this process in a peer reviewing environment for a group understanding of the requirements as *the more people who can help design the system the greater the number of good ideas incorporated.* [9]

Throughout the entire design process, a common language was determined to be critical, to avoid confusion when referring to differing aspects of the solution. For this a Ubiquitous Language [10] was created, and a comprehensive and shared set of terms are seen in Appendix A - *Definition of terms and concepts used within the APS system (Ubiquitous Language).*

The Ubiquitous Language was found to be effective in communicating clearly and accurately amongst the G2 members.

Figure 1 illustrates the Ubiquitous language and its central role as the single language of expression between all stakeholders for the solution.

This language was used to support the DDD approach taken for the APS Solution. The solution was broken into multiple Context.

A Context is *the setting in which a word or a statement appears that determines it’s meaning* [10]*.* Although the solution is broken into multiple Contexts the language used is still shared. This might suggest a single Context implementation should have been used, however a requirement as determined by the Domain Experts was to allow each Context to be autonomous.

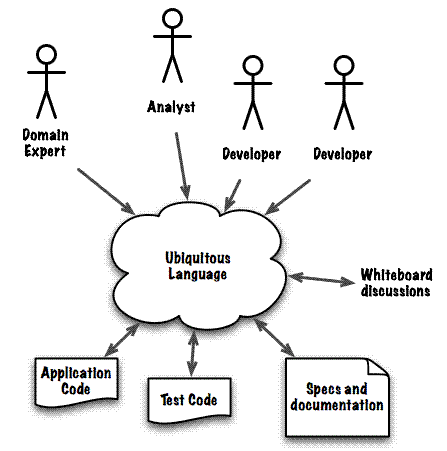


Figure - The central role of the Ubiquitous Language [10]

*A Messaging Pattern is a network-oriented*[*architectural pattern*](http://en.wikipedia.org/wiki/Architectural_pattern)*which describes how two different parts of a*[*message passing*](http://en.wikipedia.org/wiki/Message_passing) *system connect and communicate with each other* [4].

Based on the strategic design, it was concluded that DDD would be an appropriate technique to model the design of the system, based on the varying complexities of each component, as well as the emphasis will be on the Core Domain and its logic. Another technique known as TDD with the aid of Specification by Example, will be used in conjunction with DDD to develop the software.

DDD is used to guide the design of the system from the top down, and TDD would direct the design through the development life cycle (bottom up). These techniques work well together because DDD systems make it easier to use TDD as it allows for easily navigating the system through unit tests.”who says”

Using DDD allows the design of the system not to depend on the infrastructure (for example, a database). This is possible because the model code is separated from the infrastructure. This makes DDD persistence ignorant, therefore allowing the design to focus on solving the business problem rather than how the data would be structured. These logical modules are as known as domain models.

In order to successfully execute DDD there needs to be a common language between the Domain Experts and the team that will be developing the product.

*Since the model is the common ground, the place where the software meets the domain, it is appropriate to use it as the building ground for this language* [3].

This language is called the Ubiquitous Language. The requirement for the Ubiquitous Language is so that all team member are able to communicate clearly and accurately (Appendix A outlines the Ubiquitous Language used for the project). This helps when a work breakdown structure needs to be created, as the language will allow differing Software Developers to work on different components with a common understanding.

Figure 1 demonstrates the key role of Ubiquitous Language in DDD [1].

*Figure 2 - The Ubiquitous Language should be the only language used to express a model [1]*

Discuss how using the responsibility breakdown simplified the process of work allocation and implementation, which allows for developer swap out if it was needed.

The following sections.. will describe all the sections to come as well as detailing how a Domain Driven Design(DDD) approach was taken for determining the design and development strategy.

This was done by doing a high level strategic design first to see if the scope of work fit a level of complexity/triviality necessitating DDD. After the decision was taken to use a DDD approach, the development methodology that was taken was one of TDD using the Specification By Example technique.

Cite how these concepts are known to work well together especially in a DDD context

Discuss creating a ubiquitous language and how a set of definitions (see appendix A ) was created.

Discuss how using the responsibility breakdown simplified the process of work allocation and implementation, which allows for developer swap out if it was needed.

Discuss how the design was not data centric, but domain and functionality design.[ref?]

# Domain Driven Design

Discuss DDD:

What is DDD, naming the components?

Why we chose to take a DDD approach to our project (What is good about DDD and what is not) our expected complexity in the future, and focusing the departments

Discuss development being iterative and test driven, collaborative

When to use DDD, when not to - Lessons we have learnt about it.

## Components of Domain Driven Design

### Domains/Core Domain

### Sub Domains

### Generic Sub Domains

### Bounded Contexts

Explain what they are and site an example

*Bounded Contexts needn't be organized solely by the functional area of an application. They're very useful in dividing a system to achieve desired architectural examples. The classic example of this approach is an application that has both a robust transactional footprint and a portfolio of reports.* [2]

### Aggregates, Entities and Value Objects

Explain what they are and site an example explain

### Repositories

Explain the repository pattern and how are using it

# The Strategic Vision and Design of APS

## APS Domains and Responsibility Decomposition

Discuss breaking the requirements into separate areas of responsibilities or departments/domains (explaining their types) that would deal with data/domain models in their own way – at a high level

### Core Domain

Discuss the decision that was taken of the core business proposition being that of collating and creating statements from multiple providers

Discuss function and responsibilities at a high level and how any changes of how we do business should flow out from there.

### Sub Domains

Discuss that customer registration and business company creation are sub domains and could potentially be separated into their own systems depending on how customers and billing companies grow in the future.

We wanted to allow customer registration and company additions as well as scraping to be scalable and autonomous.

Discuss function and responsibilities at a high level

### Generic Sub Domains

3rd Part scraping component – should we ever want a different one, we are not coupled to it.

Data persistence is also generic by means of the repository pattern. Discuss function and responsibilities at a high level

### Integrating the different domains

Due each of the Domains and their Application Services running autonomously an integration mechanism was required that would provide cross-domain logging, as well as a common language spoken between systems.

Discuss loose coupling

*The core principle behind loose coupling is to reduce the assumptions two parties (components, applications, services, programs, users) make about each other when they exchange information. The more assumptions two parties make about each other and the common protocol, the more efficient the communication can be, but the less tolerant the solution is of interruptions or changes because the parties are tightly coupled to each other.* [6]

Discuss the interaction requirements and how we planned to integrate the different systems once they we split into their own

Explain the event integration service – how it works, how it can be swapped to another integration mechanism

### Queries, Single Responsibility Principle and Interface Segregation principle

Explain the “queries” and how we use the DTO Pattern. [7] Show how they also adhere to single responsibility

### CQS?

# Tactical Design

DDD as previously mentioned is split into two distinct designs; the Strategic Design and the Tactical Design, the Tactical being more implementation focused.

The following section highlights multiple Tactical aspects, including; the allocation of work for the Group two members, the project structure, the method applied for Continuous Integration, testing, collaboration and integration mechanisms, as well as OO Principles followed .

## Domain and feature allocation

Describe who did what at a high level and note these sections will be discussed in detail in the individual reports.

## Project structure

The APS Solution, can be broken down into 6 categories, being; Unit Tests, Fakes, Domain Models [8], Application Services, the Integration and Published Language [8] as well as common Classes.

A short description and/or reasoning for each of the categories is as follows:

### Unit Tests

All Unit Test Projects are separated from the Classes being tested, so that when the APS Solution is no longer a Prototype, the solution can be deployed without the tests using a customised Build Configuration [9].

### Fakes

The Fakes project is a collection of placeholder Implementations of all of the Aggregate Repositories in the solution. These placeholders facilitate testing functionality by acting as rudimentary in-memory collection stores. The intention was to not design the solution around the data, but around the Domain requirements, with the understanding the data structure will follow.

### Domain Models

The Domain Model projects are what was described previously in section 4.1. They house Domain Aggregates, Entities and Value Objects performing a specific business function.

Examples are of managing Customers, managing Billing Companies or even running the *Scrape Sessions.*

Each project is isolated for any Software Developer who is working on a Domain Model to work independently of anything else.

### Application Services

An Application Service in the context of the APS Solution is a Class within a project that has the responsibility of integrating and coordinating a corresponding Domain Model Project to other Domains, by reacting to Events internal and external to the Domain Model Project.

An example of this is the *Aps.Customer.ApplicationService* project.

The main purpose of this Service is to tell the Customer Aggregate to perform certain actions based on Integration Events being raised from other Domains, such as telling the Domain Model to store a reference to an *Account Statement* when an *Account Statement* has been *composed* and stored in the *Aps.AccountStatements* Domain*.*

Other Domains are notified using the *Event Integration Service* from the Application Serviceof changes within the Customer Aggregate, such as when a new *Billing Company Account* has been *added* to the Customer Aggregate.

The Application Service could have be placed in the same project as the corresponding Domain Model project, however, the decision was taken to separate these to allow the Domain Model logic to be isolated from the processing logic for ease of development and future Domain Model or Application Service extension.

### Integration and Published Language

The *Aps.Integration* project is analogous to an *Open Host Service* [8] as defined by Eric Evans employing the DTO Pattern [8] with specific queries and common Classes to share data in a unified way amongst differing Domains, without exposing the internal Classes of the Domains.

The project also contains the Event Integration Service which allows publishing of and subscribing to a common set of events [pub/sub – event store reference here]

### Common Classes

The common classes or *Aps.DomainBase* project which contains Base Classes used by all Domains as well as the *Caliburn.Micro Event Aggregator* which allows each Domain to have the ability to *channel events from multiple objects into a single object to simplify registration for clients* [10]within the Domain itself..

Currently our Domains have a single Aggregate, but should the Domain grow to more, integrating the Application Service is simplified, and done in a uniform manner.

The Event Aggregator allows for decoupling the handling code from the Event raising code, and in turn allows for asynchronous event handling.

## Continuous Integration and Development Strategy

Iterative development

Github ( point to url for Joshua to see )

Problems encountered and resolutions taken (see section 6)

## Specifications by example and Test Driven Development

Discuss how features were tackled by means of specifications by example with 1 or two examples

## Feature integration specification collaboration

Discuss how TDD and integration took place by means of specification by example.

e.g. As a scheduling engine I need create default scheduling when a customer adds a billingcompanyaccount. This allowing integration to be facilitated without actually integrating. Compare this to interface contracts

## Object Orientation Principles

Throughout the Codebase [11] Group two, has applied SOLID programming principles. This section serves to highlight an examples of the principles.

### Single Responsibility Principle

The Single Responsibility Principle (SRP) [12] states that *a class should have one, and only one, reason to change* [12]*,* and by looking at Code Listing 1 in Appendix C, it can be seen that all the Class does is return a Data Transfer Object (DTO) by running a call on the Repository.

It could be argued that the mapping function should be contained in a resource given to the Class to perform the mapping of the DTO from the Entity, however the point of view taken was that the Repository would change if the querying mechanism changed, and not the Class in question. All this class is responsible for, is creating the DTO. This pattern has been applied throughout the Codebase for all implemented queries.

### Dependency Inversion Principle

Because the Dependency Inversion Principle (DIP) states, you should *depend on abstractions, not on concretions* [13]the Codebase has been written in such a way to adhere to this principle.

To illustrate this point, again looking at Code Listing 1 in Appendix C, it can be seen that the Repository has been provided to the Class to run the query, rather than the Repository being constructed in the Class, or by using the Service Locator Pattern (SLP) [14] to retrieve the Repository. The provision of the Repository has been made possible by using Autofac’s Inversion of Control container implementation. [15]

The premise for not using the SLP, was that the query Class would need to have knowledge of *how* to retrieve the resource of Repository. This was considered a violation of the SRP. If the SLP implementation changed, the query Class would have a secondary reason to change, other than its reason mentioned in the point above.

### Interface Segregation Principle

The implementation of the query Class in Code Listing 1 in Appendix C is a prime example of the Interface Segregation Principle (ISP) [13], even though it is not an Interface.

The ISP states that Software Developers should only *make ﬁne grained interfaces that are client speciﬁc* [13]*,* meaning that creating Classes or Interfaces that clients use should not contain Methods or Properties that are not used by the clients. The inference is to rather customise the requirements to a specialised Interface or Class specific to a client’s needs.

In the Code Listing mentioned above, the query serves one function. By looking at the naming of the queries in Figure 1 below, the prevalence of ISP can be seen.

Contrary to this, in Code Listing 2 in Appendix D, the Repository Interface exposes more Methods than *all* consumers will use, however, the purpose of the Repositories implemented in the solution is to abstract the data layer which is being used to facilitate a TDD approach by means of implemented in-memory Fake Repositories.

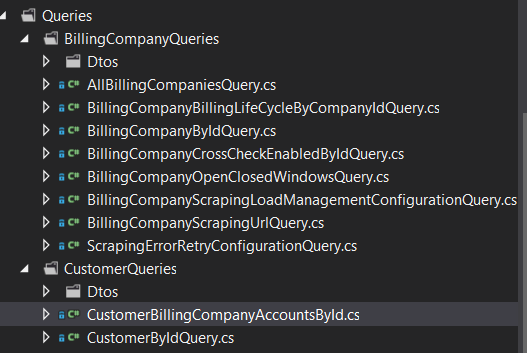


Figure - Specialised queries tailored to consumers

## Pair Programming and Code Reviews

Additional development tools that were used to facilitate the APS Solution included Collaborative or Pair Programming sessions.

*Pair or collaborative programming is where two programmers develop software side by side at one computer.* [16]

The main reason this was done is because *the project ends up with multiple people understanding each piece of the system* [16]*.* This allowed integration sessions to flow more smoothly, where each member who was developing a different part of the solution could foresee potential issues relating to expected integrations.

# Challenges and successes

# Conclusion

Was DDD good? Was TDD good? Where can we improve on our design?

# References

|  |  |
| --- | --- |
| [1] | J. Lewis and S. P. Levitt, “Course Project — Account Presentation System,” [Online]. Available: http://joshilewis.github.io/ELEN7045/elen7045-project-brief.pdf. [Accessed 11 06 2014]. |
| [2] | D. Laribee, “An Introduction To Domain-Driven Design,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/magazine/dd419654.aspx. [Accessed 03 06 2014]. |
| [3] | A. H. Eden, “Strategic Versus Tactical Design,” in *HICSS '05 Proceedings of the Proceedings of the 38th Annual Hawaii*, 2005. |
| [4] | GIT, “1.1 Getting Started - About Version Control,” GIT, [Online]. Available: http://git-scm.com/book/en/Getting-Started-About-Version-Control. [Accessed 12 06 2014]. |
| [5] | Wikipedia, “Object-oriented programming,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Object-oriented\_programming. [Accessed 12 06 2014]. |
| [6] | G. Hohpe and B. Woolf, Enterprise Integration Patterns - Designing, Building, And Deploying Messaging Solutions, Addision Wesley, p. 38. |
| [7] | M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee and R. Stafford, Patterns of Enterprise Application Architecture, Addison Wesley, 2002, pp. 347-351. |
| [8] | E. Evans and M. Fowler, Domain-Driven Design -Tackling Complexity in the Heart of Software, Prentice Hall, 2003. |
| [9] | Microsoft, “Understanding Build Configurations,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/kkz9kefa.aspx. [Accessed 09 06 2014]. |
| [10] | M. Fowler, “Event Aggregator,” [Online]. Available: http://martinfowler.com/eaaDev/EventAggregator.html. [Accessed 09 06 2014]. |
| [11] | Wikipedia, “Codebase,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Codebase. [Accessed 10 06 2014]. |
| [12] | J. Miller, “Patterns in Practice,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc546578.aspx#id0390008. [Accessed 10 06 2014]. |
| [13] | J. Lewis, “The SOLID Principles,” [Online]. Available: http://joshilewis.github.io/ELEN7045/elen7045-solid.pdf. [Accessed 10 06 2014]. |
| [14] | Microsoft, “The Service Locator Pattern,” [Online]. Available: http://msdn.microsoft.com/en-us/library/ff648968.aspx. [Accessed 10 06 2014]. |
| [15] | Autofac, “Autofac,” Autofac, [Online]. Available: http://autofac.org/. [Accessed 10 06 2014]. |
| [16] | A. Cockburn and L. Williams, “The Costs and Benefits of Pair Programming,” [Online]. Available: http://www.cs.pomona.edu/classes/cs121/supp/williams\_prpgm.pdf. [Accessed 10 06 2014]. |
| [17] | Microsoft, “Class Library,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/d11h6832(v=vs.71).aspx. [Accessed 09 06 2014]. |
| [18] | R. C. Martin, “Object Mentor,” [Online]. Available: http://objectmentor.com/resources/articles/dip.pdf. [Accessed 10 06 2014]. |

# Appendix

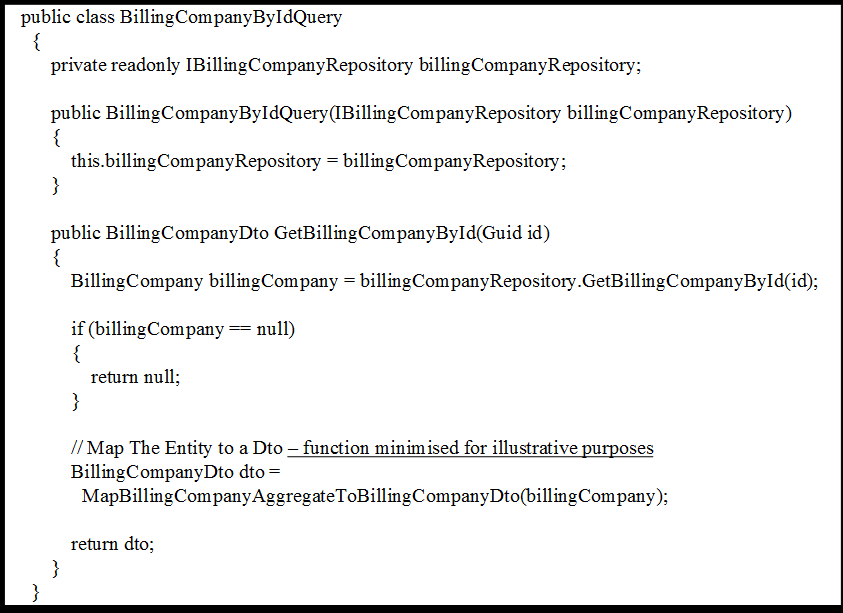
## A - Definition of terms and concepts used within the APS system (Ubiquitous Language)

|  |  |
| --- | --- |
| **Term/Concept** | **Definition** |
| Customer | Person or persons who register as a customer of the APS system |
| Customer Registration | Details of customer used/stored on APS |
| Billing Company | Business that APS interacts with to retrieve customer statements from on behalf of customers |
| Customer Billing Account | Credentials and information pertaining to the account information as held by a customer at a billing company |
| Scrape Session | Process or workflow used by APS to collect, Interpret, Validate and compose statements for a customer from a billing company |
| Scrape Session Data | Information received from a billing company via the scraper for a customer |
| Scrape Session Converter | Conversion of scrape session data into an APS specific format determining success or failure of the scrape session. |
| Scrape Session Failure Handling | Processing of different errors that could be returned in the Scrape Session data |
| Scrape Session Data Pairs | Key value pairs of data returned from the billing company when scraping converted into the APS format |
| Scrape Session Validation | Process of taking the Scrape Session Data Pairs and analysing them for inconsistencies and performing differing forms of integrity checking |
| Customer Billing Account Statement Composition | Creation of a customer statement from valid Scrape Session Data Pairs |
| Scrape Session Queued | Defines that a Scrape Session has been stored for later triggering |
| Scrape Session Scheduler | Means by which Scrape Sessions are stored and retrieved for execution |
| Static page on front end | Non-customer interactive web page which may/may not pull data from a data storage mechanism and display to a customer |

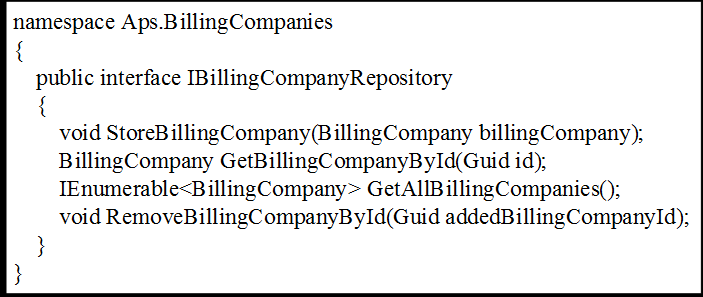
## B – Domain Integration Diagram



## C – Code Listing 1 – Example Query



## D - Code Listing 2 – Repository Interface



## E – Group Two Members

|  |  |
| --- | --- |
| **Student** | **Student Number** |
| Kgang Moloke | 779050 |
| Jignesh Narain | 779086 |
| Carlos Ribeiro | 778903 |
| Grant Southey | 9500318V |
| Wynand Viljoen | 764746 |