Abstract and cover page

Table of Contents

[1 Introduction 1](#_Toc390794330)

[2 Assumptions and Exclusions 1](#_Toc390794331)

[3 Design Methodology 2](#_Toc390794332)

[4 Domain Driven Design 3](#_Toc390794333)

[4.1 Components of Domain Driven Design 4](#_Toc390794334)

[4.1.1 Core Domain 4](#_Toc390794335)

[4.1.2 Sub-Domains 5](#_Toc390794336)

[4.1.3 Generic Sub-Domains 5](#_Toc390794337)

[4.1.4 Aggregates, Entities and Value Objects 5](#_Toc390794338)

[4.1.5 Bounded Contexts 6](#_Toc390794339)

[4.1.6 Repositories 6](#_Toc390794340)

[5 The Strategic Vision and Design of APS 6](#_Toc390794341)

[5.1 APS Domains and Responsibility Decomposition 6](#_Toc390794342)

[5.1.1 Domain Components 7](#_Toc390794343)

[5.1.2 Core Domain 7](#_Toc390794344)

[5.1.3 Sub Domains 7](#_Toc390794345)

[5.1.4 Generic Sub-Domains 8](#_Toc390794346)

[5.1.5 Integrating the different domains 8](#_Toc390794347)

[6 Tactical Design 8](#_Toc390794348)

[6.1 Domain and feature allocation 8](#_Toc390794349)

[6.2 Project structure 9](#_Toc390794350)

[6.2.1 Unit Tests 9](#_Toc390794351)

[6.2.2 Fakes 9](#_Toc390794352)

[6.2.3 Domain Models 9](#_Toc390794353)

[6.2.4 Application Services 9](#_Toc390794354)

[6.2.5 Integration and Published Language 10](#_Toc390794355)

[6.2.6 Common Classes 10](#_Toc390794356)

[6.3 Development Strategy and Version Control 10](#_Toc390794357)

[6.4 Specifications by Example 11](#_Toc390794358)

[6.5 Test Driven Development 12](#_Toc390794359)

[6.6 Object Orientation Principles 12](#_Toc390794360)

[6.6.1 Single Responsibility Principle 12](#_Toc390794361)

[6.6.2 Dependency Inversion Principle 12](#_Toc390794362)

[6.6.3 Interface Segregation Principle 13](#_Toc390794363)

[6.7 Pair Programming and Code Reviews 13](#_Toc390794364)

[6.8 Domain Building Blocks 14](#_Toc390794365)

[6.8.1 Aggregates and why 14](#_Toc390794366)

[6.8.2 Entities 14](#_Toc390794367)

[6.8.3 Value Objects 14](#_Toc390794368)

[7 Proposed Enhancements 14](#_Toc390794369)

[8 Conclusion 14](#_Toc390794370)

[9 References 15](#_Toc390794371)

[10 Appendix 18](#_Toc390794372)

[A - Definition of terms and concepts used within the APS system (Ubiquitous Language) 18](#_Toc390794373)

[B – Domain Integration Diagram 19](#_Toc390794374)

[C – Code Listing 1 – Example Query 20](#_Toc390794375)

[D - Code Listing 2 – Repository Interface 21](#_Toc390794376)

[E – Group Two Members 22](#_Toc390794377)

[F – Responsibilities and Weightings 23](#_Toc390794378)

[G – Events used in the APS Solution 24](#_Toc390794379)

[H – Data Transfer Objects used in the APS Solution 25](#_Toc390794380)

[I – Web Interface mock-ups 26](#_Toc390794381)

# 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).

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 Design (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 [5] mechanism, project structure, as well Object-Oriented Programming (OOP) [6] principles followed will be evaluated.

Lastly, possible future enhancements will be discussed.

# Assumptions and Exclusions

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]) because it was mentioned that *the project brief is intentionally vague* [1]*.*

The Web Interface as required has not been implemented, however a mock-up of how it potentially would look, as well as how the data for it is expected to be retrieved and displayed can be seen in Appendix I.

# 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. [8]

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 an 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 [9] was decided on as it allowed the components to not be directly coupled to each other.

*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].

The implementation of how the shared Publish-Subscribe Messaging Pattern was used in the project will be discussed in 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 [10] 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.* [10]

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 [11] was created, and a comprehensive and shared set of terms are seen in Appendix A.

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* [11]*.* 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.

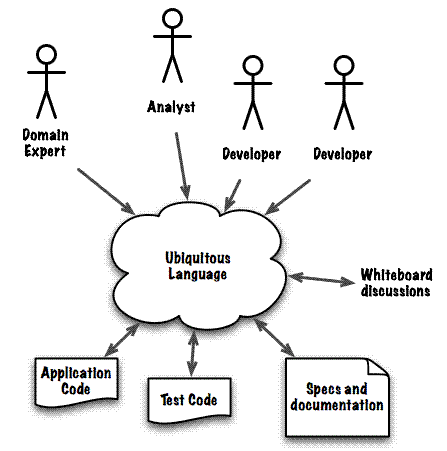


Figure 1- The central role of the Ubiquitous Language [11]

G2 concluded that DDD would be an appropriate technique to model the design of the system, based on the varying complexities of each Context, in particular the rules governing the Scrape Scheduling. DDD will be discussed further in the next section.

Throughout the APS Solution implementation an Agile [12] and TDD approach was used. This iterative style of development, in conjunction with Specifications by Example, facilitated the evolution of the APS Solution prototype.

Using DDD allowed the design and implementation of the system not to depend on the structure of the data but rather on the business requirements or Domain Models.

# Domain Driven Design

Domain Driven Design (DDD) is the philosophy of modelling the Software Engineering world as closely as possible to the real world. This happens while the Software Engineer designs the architecture and programmatic solution.

As mentioned previously, a Ubiquitous Language is defined per Context allowing all stakeholders to communicate without any technical jargon or confusion. This facilitates a faster evolution of the solution as problematic areas are discovered sooner.

*Domain-Driven Design (DDD) is a collection of principles and patterns that help developers craft elegant object systems. Properly applied it can lead to software abstractions called domain models. These models encapsulate complex business logic, closing the gap between business reality and code.* [2]

Figure 2 taken from [13] demonstrates the interactions of the DDD Building Blocks that will be discussed in the upcoming section.

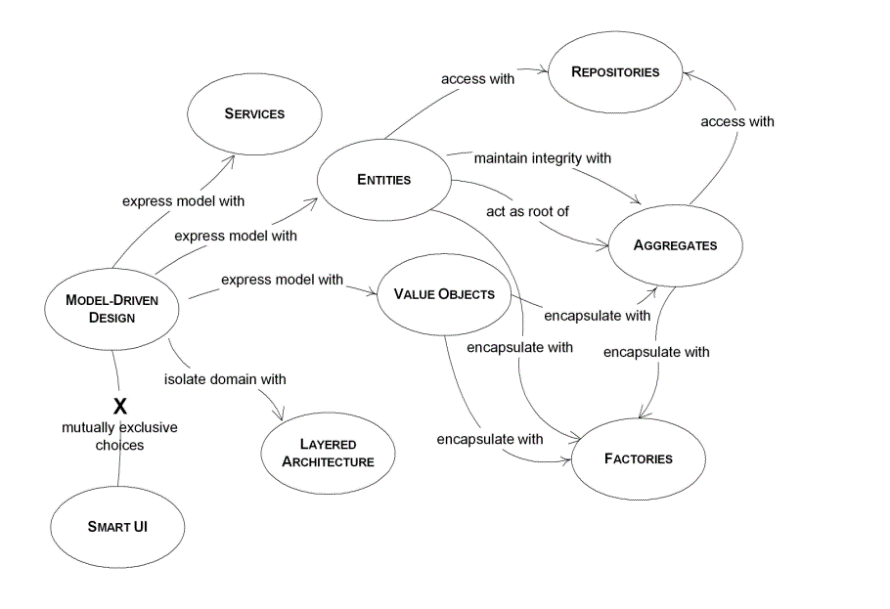


Figure 2 - Domain Driven Design Building Blocks

## Components of Domain Driven Design

Domains or Domain models within DDD are responsible for discrete business offerings.

These Domains Models are sub-divided into levels of importance starting with the Core Domain, Sub-Domains and, at the lowest level of importance, Generic Sub-Domains.

The Domains implemented in the APS Solution will be discussed in section 5.

### Core Domain

The Core Domain in the DDD approach is responsible for delivering the primary business offering. The Core Domain is the most valuable Domain in the solution, and this Core Domain deserves the utmost focus in design. [14]

Any fundamental changes to the business offering should flow out of this Domain.

### Sub-Domains

The Sub-Domains can be identified as important business elements that are not core to the business offering, but support the functioning of the solution.

### Generic Sub-Domains

Generic Sub-Domains contain functionality that is not developed internally but is used by the system to provide the required functionality. These typically are 3rd Party components such as data persistence.

### Aggregates, Entities and Value Objects

DDD contains some fundamental Building Blocks such as; Aggregates, Entities and Value Objects. These are briefly described below.

The implementation of how these Building Blocks are used within the APS Solution is described in section 6.

#### Aggregates

Martin Fowler defines an Aggregate as:

*A DDD Aggregate is a cluster of domain objects that can be treated as a single unit.* [15]

An Aggregate provides a consistency boundary where, all requests for changes to the Aggregate, or its child Objects (Entities or Value Objects), are requested through it.

The Aggregate directs requests to where they should be executed. Validations are performed by the Aggregate for grouped business rules. Child Entities will perform cascading validations on themselves and if successful execute the changes to their child Objects.

The Aggregate is also referred to as the Aggregate root. An Aggregate is also an Entity.

#### Entities

Entities can be described as Objects that require identity. Aggregates contain an identity value that is global to the entire solution.

Child Objects within Aggregates only require local identity due to them only ever being accessed via the Aggregate.

#### Value Objects

Value Objects are Objects that exist within Entities. These Objects do not have identity and are immutable. They are primarily used for their state values and due to their immutability can easily be interchanged without value corruption.

### Bounded Contexts

Bounded Contexts are the boundaries in which one or more Domains lie. In each of these Bounded Contexts, a particular Ubiquitous Language is used.

All of the Entities, Value Objects, Repositories, and other Building Blocks in the Domain within the Bounded Context.

### Repositories

The Repository Design Pattern [16] is used to create an abstraction between the Domain layer and the Data Persistence layer. To clarify, this means that there is no need for the components in the Domain layer to know how the data is provided.

# The Strategic Vision and Design of APS

## APS Domains and Responsibility Decomposition

During the requirement gathering processes for the APS Solution, three main areas of expertise or business needs were identified; Customer registration and maintenance, Billing Company or partner maintenance and Scheduling of Scrape Sessions with Account Statement generation.

These areas needed to be integrated into one solution to provide the required functionality of the system. A possible solution would have been to combine all three areas into a single application where all the functionality would reside.

This particular design was disregarded as it could easily have turned into a big ball of mud [17]. All the components would have been coupled to each other (albeit potentially just in the same application). Avoiding tight coupling was considered to be a design objective and can be justified by examining the following quote from Dino Esposito:

*Tight coupling is beneficial because it helps you write code faster, and that code will likely run faster. It doesn’t, however, make the code maintainable.* [17]

For the solution to be more robust, extensible and maintainable, it had to be more loosely coupled for the following reason:

*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.* [8]

During the requirements gathering process it became apparent that in order for the design to be loosely coupled, the registrations, Billing Company and Scheduling components of the system needed to be isolated from each other. The Domain Experts as previously mentioned had a requirement for each of the components to work autonomously and this required loose coupling.

An architectural example of how different Domains interact in the designed Solution is illustrated in Appendix B. The implemented design has multiple applications working together to accomplish the business requirements. The business requirements were divided into multiple Domains.

This domain architecture was adopted from Martin Fowler’s article:

*In short, the Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.* [18]

The implemented design differs in some ways from the statement above but the main idea behind it remains the same. This main idea is to create a suite of small applications each running autonomously and working together through an integration mechanism to provide the functionality required by the business.

When implementing DDD it is important to consider the business domain and requirements and then identify and separate the elements that are core to the business and the elements that support the core. The remainder of this section will discuss the implemented design from a high-level architectural perspective using the Domains as a basis.

### Domain Components

Appendix B shows two Domains and their composition. Each Domain is comprised of; an Application Service responsible for business process orchestration, a Repository per Aggregate in the Domain, an Event Integration Service (See Section 5.1.5) and customised queries that return Data Transfer Objects (DTO) [13] to external Domains.

### Core Domain

After analysing the system requirements it was identified that the main business objective was to be able to retrieve, collate and create statements for Customers from multiple external e-Billing providers.

### Sub Domains

In order to support the Core Domain, the following Sub-Domains were identified; Customer registration and maintenance and Billing Company creation and maintenance.

### Generic Sub-Domains

In the APS Solution there are the following Generic Sub-Domains; the 3rd Party Scraping component and the data persistence components for each Domain.

It could be argued that the 3rd Party Scraper and data persistence mechanisms could be considered services, however because they could be swapped for an alternate solution they are deemed generic.

### Integrating the different domains

The Domains discussed in the previous sub-sections, work independently from each other. Although each Domain executes and maintains its own data, it provides little business value when isolated. This means that there is a need to integrate these Domains so they could collaborate with each other in order to fulfil the business objective.

The different Domains communicate with each other through an integration mechanism called the Event Integration Service (See Appendix B). This service provides cross-domain logging, as well as a common language between the different Domains to facilitate information and event sharing (See Section 6.2.5).

The partial architecture diagram shows a *Query Returning DTO* (See Appendix B). This concept is responsible for the retrieval of Aggregate, Entity and Value Object data from one Domain and translating it into a DTO to be consumed by another Domain.

# 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 G2 members, the project structure, the method applied for Version Control, testing via TDD, collaboration and integration mechanisms, as well as OOP Principles followed.

## Domain and feature allocation

G2 had multiple sessions defining the Strategic design the project. Following that, G2 as discussed previously decomposed the requirements in the brief into the differing Domains and their interactions, while keeping the reporting/auditing requirements in mind.

These features were allocated effort weightings. The allocation of features initially were distributed on this weighting basis. The features and their weightings can be seen in Appendix F.

Understanding the complexity and type of work involved by using this breakdown was found to be a valuable tool for work allocation.

During the development process, the weightings were altered and new ones added as the solution evolved. This is typical of the Agile process.

## Project structure

The APS Solution, can be broken down into 6 categories, being; Unit Tests, Fakes, Domain Models [13], Application Services, the Integration and Published Language [13] 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 [19].

### 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.

The 3rd Party WebScraper has also been added as a Fake for testing.

### Domain Models

The Domain Model projects are what were described previously in section 5.1. 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. It does this 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.

A list of all the internal and external Events used within the prototype APS Solution is shown in Appendix G.

### Integration and Published Language

The *Aps.Integration* project is analogous to an *Open Host Service* [13] as defined by Eric Evans employing the DTO Pattern [13] with specific queries and common Classes to share data in a unified way amongst differing Domains, without exposing the internal Classes of the Domains.

A list of all the DTO’s used within the prototype APS Solution is shown in Appendix H.

The project also contains the Event Integration Service which allows publishing of and subscribing to, a common set of events [9].

### Common Classes

The common classes or *Aps.DomainBase* project contains Base Classes used by all Domains.

This project also contains 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* [20]within the Domain itself.

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

## Development Strategy and Version Control

An iterative method was chosen as the approach in which the prototype would be developed.

*The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental), allowing software developers to take advantage of what was learned during development of earlier parts or versions of the system* [21]*.*

This approach allowed the team to identify risks early in the life cycle and to deal with them in a timeous and well-organised way. Due to the project having to be continuously tested this gave a clear view of the project status.

An effective Branching and Merging Strategy [22] which would fit the development scenario of the team had to be chosen. The strategy needed to allow each team member to work individually without being subject to breaking changes.

A *Branch for Feature* [22] type of strategy was considered, this meant that the development branches would be organised based on product features. This strategy did not fit the development scenario because each team member needed to work in isolation and also on multiple features.

*A Branch for Team* type of approach was chosen due to the fact that it was similar to the *Branch for Feature* type of approach except that the development Branches were organized according to team member rather than by product feature.

The next decision was to use *Github* [23] as the version control system that would allow team members to collaborate and have revision control. All the source code for the project can be found at *https://github.com/thedarkjester/CPD7045*. This allowed the team to easily share ideas and to track changes made to the source code and project documentation.

## Specifications by **E**xample

*In recent years delivery speed has been become a key component in Software Development* [24]. When creating effective Requirement Documentation the goal is to create just enough Documentation at the right time and for the correct audience [25].

Since a proof of concept was being developed a decision was taken that the requirements gathering technique selected needed to be lightweight, easy to maintain and verifiable.

Three requirement gathering techniques were investigated, these included Use Cases [26], User Stories [27] and Specification by Example [24].

Specification by Example was selected due to the development happening in isolation. Examples were needed so that development could happen without the specifications being vague or ambiguous. The other two options were deemed less definitive in their use.

Each team member was assigned differing parts of the APS Solution as mentioned, however, the Specifications were written both individually and collaboratively.

These Specifications were then reviewed by other team members. This led to the team having a shared understanding of the low-level system requirements in Domains they were not working on.

Specification by Example provided the team the right documentation at the right time, which fits in well with the iterative development approach.

## Test Driven Development

TDD is a development methodology in which the system is written using an iterative approach in which tests are written initially, followed by software being written in order to pass these tests. TDD works is as follows:

1. *Create a test and make it fail.*
2. *Write code to make the test pass by any means necessary.*
3. *Refactor the code in order to remove duplication and improve the design, ensuring that all tests still pass.*
4. *Repeat the above steps in an iterative manner.* [4]

Using a Test Driven Development approach afforded the team the following benefits; Writing Unit Tests forced the Developer to have acceptance criteria of what is the definition of done [28]. The team was able to refactor system changes and have immediate feedback in the form of test results on whether the changes had any undesired effects.

## Object Orientation Principles

Throughout the Codebase [29] G2 has applied SOLID [30] programming principles. This section serves to highlight examples of the principles.

### Single Responsibility Principle

The Single Responsibility Principle (SRP) [31] states that *a class should have one, and only one, reason to change* [31]*,* 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* [30]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) [32] to retrieve the Repository. The provision of the Repository has been made possible by using Autofac’s Inversion of Control container implementation. [33]

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) [30], even though it is not an Interface.

The ISP states that Software Developers should only *make ﬁne grained interfaces that are client speciﬁc* [30]*,* 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 3 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.

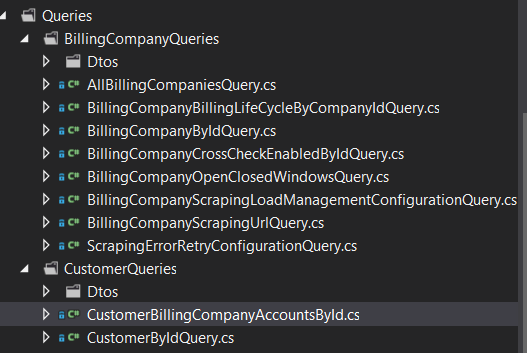


Figure 3- Specialised queries tailored to consumers

## 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.* [34]

The main reason this was done is because *the project ends up with multiple people understanding each piece of the system* [34]*.* 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.

## Domain Building Blocks

This section will show which elements was selected by G2 to fulfil the required Building Blocks of DDD within the prototype APS Solution.

### Aggregates

Section 5.1 shows that there are three different Domains in the prototype APS Solution.

To ensure that the data and Objects within each Domain could be treated as single unit and that it is kept in a consistent state, each Domain needs to have an Aggregate Root.

The Customer Domain is responsible for User registration and maintenance of all Customer related data. For this reason, the *Customer* Class was given identity and functions as the Aggregate Root of the Customer Sub-Domain.

The Billing Company Domain is responsible for the registration and maintenance of all external e-Billing Companies. For this reason, the *BillingCompany* Class was given identity and functions as the Aggregate Root of the Billing Company Sub-Domain.

The final Domain is the Core Domain which is responsible to collate and create statements for Customers from multiple external e-Billing providers as stated in section 5.1.2. To achieve this functionality the *ScrapingObject* Class was selected as the Aggregate Root of the Core Domain.

### Entities

Entities are Objects that have identity, but this identity is only local (identifiable within the same Domain) if the Object is not used as in Aggregate within the DDD framework as discussed in section 4.1.4.2.

The *Customer* Class located inside the Customer Domain contains a List of *CustomerBillingCompanyAccounts* as one of its instance variables. A *CustomerBillingCompanyAccounts* Object contains a *BillingCompanyId* and also all user authentication data needed for the APS Solution to perform a scrape for that user with the specific *BillingCompany*. During the requirements gathering process it was discovered that an APS Solution customer could potentially have multiple accounts with the same external e-Billing Company. An example of this would be a person that owns multiple properties; he/she would have multiple municipal accounts, but only one APS account. A design decision was made to give the *CustomerBillingCompanyAccounts* Class identity to ensure these multiple accounts could be identified correctly within the Customer Domain.

### Value Objects

The Value Objects used in the prototype solution enables the Entities in the different Domains to have state.

In the Customer Domain the Value Objects used includes the Customer’s APS Solution user credentials, email address, first name, last name, telephone number etc.

In the Billing Company Domain the Value Objects used includes the Billing Companies’ name, the scraping URL, Billing Companies’ statement life cycle, Open/Closed windows for the Billing Company etc.

In the Core Domain the Value Objects used includes the status of the specific scrape, the date the scrape is scheduled to execute, the date the scrape request was created etc.

# Proposed Enhancements

Where can we improve on our design?

# Conclusion

Was DDD good? Was TDD good?

# References

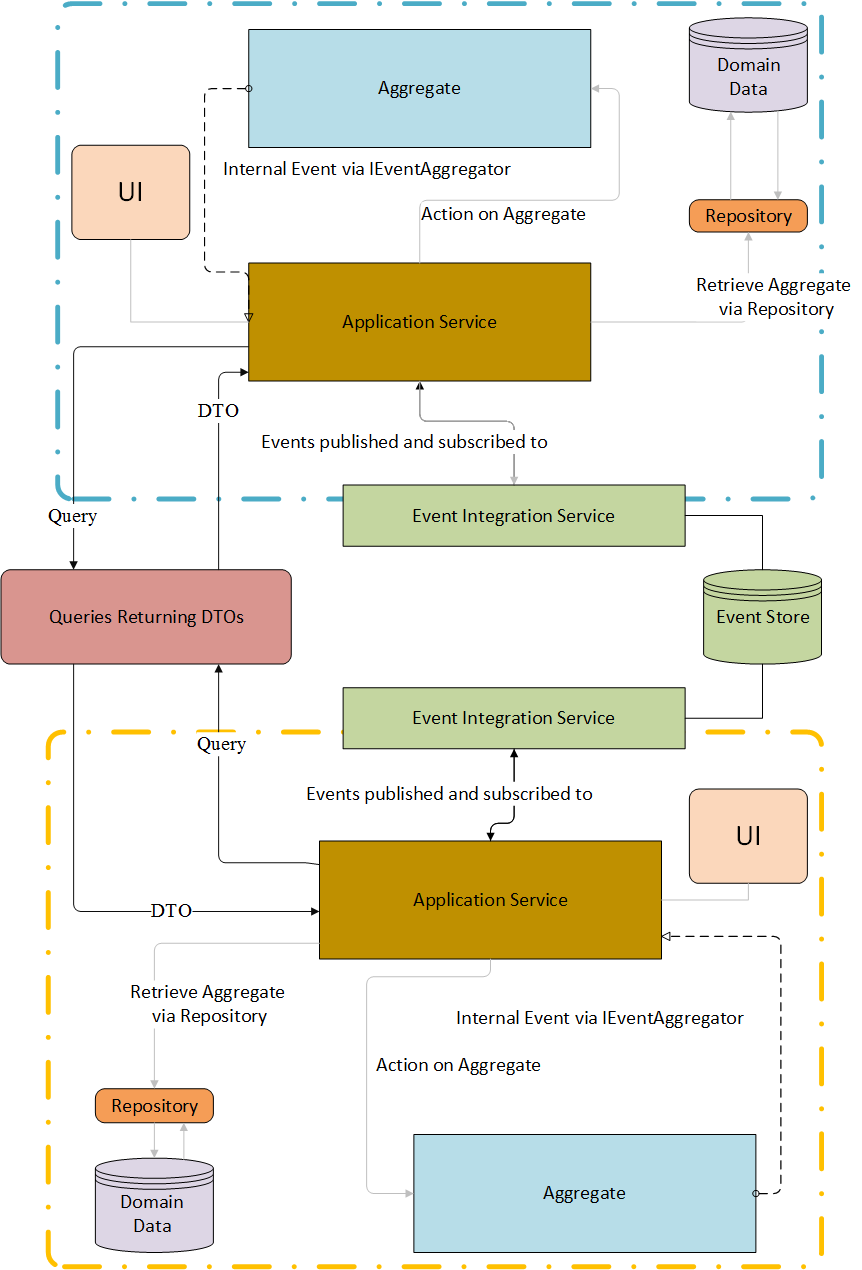
|  |  |
| --- | --- |
| [1] | J. Lewis and S. P. Levitt, “Course Project — Account Presentation System,” [Online]. Available: http://joshilewis.github.io/ELEN7045/elen7045-project-brief.pdf. [Accessed 11 06 2014]. |
| [2] | D. Laribee, “An Introduction To Domain-Driven Design,” Microsoft Developer Network, [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] | Microsoft and J. Palermo , “Guidelines for Test-Driven Development,” Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx. [Accessed 14 06 2014]. |
| [5] | 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]. |
| [6] | Wikipedia, “Object-oriented programming,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Object-oriented\_programming. [Accessed 12 06 2014]. |
| [7] | Microsoft, “Class Library,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/d11h6832(v=vs.71).aspx. [Accessed 09 06 2014]. |
| [8] | G. Hohpe and B. Woolf, Enterprise Integration Patterns - Designing, Building, And Deploying Messaging Solutions, Addision Wesley, p. 38. |
| [9] | S. Chatterjee, “Messaging Patterns in Service-Oriented Architecture, Part 1,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/aa480027.aspx. [Accessed 14 06 2014]. |
| [10] | Extreme Programming, “CRC Cards,” Extreme Programming, [Online]. Available: http://www.extremeprogramming.org/rules/crccards.html. [Accessed 14 06 2014]. |
| [11] | A. Brandolni, “Strategic Domain Driven Design with Context Mapping,” InfoQ, [Online]. Available: http://www.infoq.com/articles/ddd-contextmapping. [Accessed 14 06 2014]. |
| [12] | M. Fowler and E. al, “Principles behind the Agile Manifesto,” [Online]. Available: http://agilemanifesto.org/principles.html. [Accessed 10 06 2014]. |
| [13] | E. Evans and M. Fowler, Domain-Driven Design -Tackling Complexity in the Heart of Software, Prentice Hall, 2003. |
| [14] | E. Evans, “Domain Driven Design Reference,” [Online]. Available: https://domainlanguage.com/ddd/patterns/DDD\_Reference\_2011-01-31.pdf. [Accessed 14 06 2014]. |
| [15] | M. Fowler, “DDD Aggregates,” [Online]. Available: http://martinfowler.com/bliki/DDD\_Aggregate.html. [Accessed 14 06 2014]. |
| [16] | Microsoft, “The Repository Pattern,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/ff649690.aspx. [Accessed 14 6 2014]. |
| [17] | D. Esposito, “Software Disasters: Recovery and Prevention Strategies,” Microsoft Development Network, [Online]. Available: http://msdn.microsoft.com/en-us/magazine/hh394145.aspx. [Accessed 14 06 2014]. |
| [18] | M. Fowler, “Microservices,” [Online]. Available: http://martinfowler.com/articles/microservices.html. [Accessed 14 6 2014]. |
| [19] | Microsoft, “Understanding Build Configurations,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/kkz9kefa.aspx. [Accessed 09 06 2014]. |
| [20] | M. Fowler, “Event Aggregator,” [Online]. Available: http://martinfowler.com/eaaDev/EventAggregator.html. [Accessed 09 06 2014]. |
| [21] | Wikipedia, “Iterative and Incremental Development,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Iterative\_and\_incremental\_development. [Accessed 11 06 2014]. |
| [22] | Microsoft, “Chapter 5 – Defining Your Branching and Merging Strategy,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/bb668955.aspx. [Accessed 11 06 2014]. |
| [23] | GitHub, “GitHub,” [Online]. Available: https://github.com/. [Accessed 14 06 2014]. |
| [24] | G. Adzic, Specification by Example: How Successful Teams Deliver the Right Software, Manning Publications, 2010. |
| [25] | [Online]. Available: http://www.agilemodeling.com/essays/agileDocumentation.htm. [Accessed 10 06 2014]. |
| [26] | Microsoft Developer Network, “UML Use Case Diagrams: Reference,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/dd409427.aspx. [Accessed 17 6 2014]. |
| [27] | Microsoft Developer Network, “User Story (Agile),” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/dd380634%28v=vs.100%29.aspx. [Accessed 17 6 2014]. |
| [28] | S. Freeman and N. Pryce, Growing Object-Oriented Software, Guided by Tests, Addison-Wesley Professional, 2009. |
| [29] | Wikipedia, “Codebase,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Codebase. [Accessed 10 06 2014]. |
| [30] | J. Lewis, “The SOLID Principles,” [Online]. Available: http://joshilewis.github.io/ELEN7045/elen7045-solid.pdf. [Accessed 10 06 2014]. |
| [31] | J. Miller, “Patterns in Practice,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc546578.aspx#id0390008. [Accessed 10 06 2014]. |
| [32] | Microsoft, “The Service Locator Pattern,” [Online]. Available: http://msdn.microsoft.com/en-us/library/ff648968.aspx. [Accessed 10 06 2014]. |
| [33] | Autofac, “Autofac,” Autofac, [Online]. Available: http://autofac.org/. [Accessed 10 06 2014]. |
| [34] | 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]. |
| [35] | M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee and R. Stafford, Patterns of Enterprise Application Architecture, Addison Wesley, 2002, pp. 347-351. |
| [36] | R. C. Martin, “Object Mentor,” [Online]. Available: http://objectmentor.com/resources/articles/dip.pdf. [Accessed 10 06 2014]. |
| [37] | [Online]. Available: http://msdn.microsoft.com/en-us/library/bb668955.aspx. [Accessed 11 06 2014]. |

# Appendix

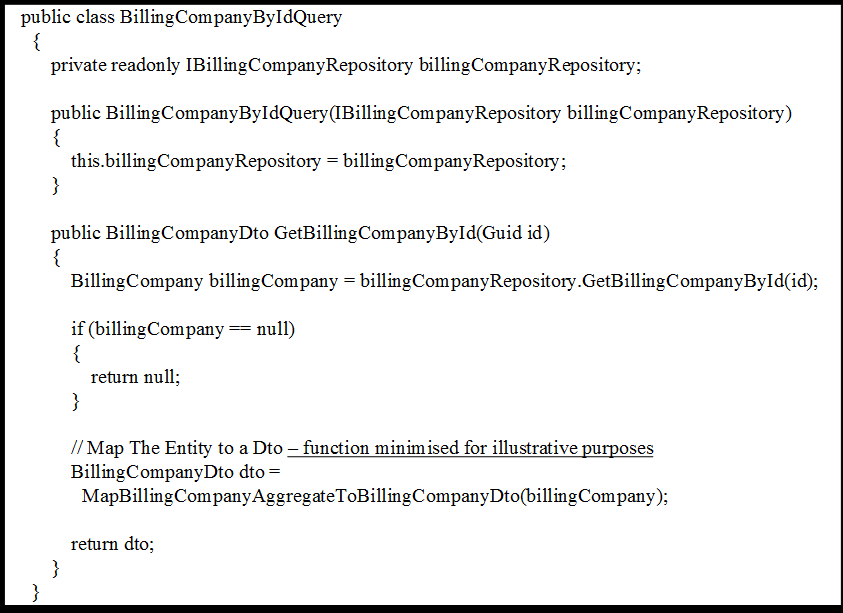
## A - Definition of terms and concepts used within the APS system (Ubiquitous Language)

|  |  |
| --- | --- |
| **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 |

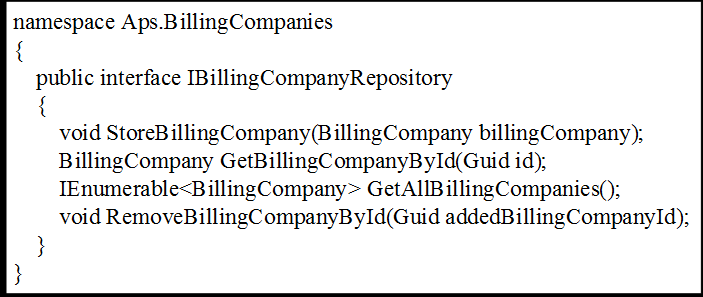
## B – Domain Integration Diagram



## 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 |

## F – Responsibilities and Weightings

Table 1: Responsibility Weightings

|  |  |
| --- | --- |
| **Responsibility** | **Weighting** |
| Scrape Session XML To Data Pair Converter | 2 |
| Scrape Session Data Validator | 8 |
| Scrape Session Failure Handler | 7 |
| Customer Registration and Maintenance | 3 |
| Scrape Session Scheduler | 8 |
| Customer Account Statement UI | 2 |
| Customer Billing Account Statement Composer | 7 |
| Billing Company Maintenance | 3 |
| Auditor | 5 |
| Scrape Orchestration | 6 |
| Domain base classes | 2 |
| Event Integration Service | 5 |

## G – Events used in the APS Solution

Table 2: Events in prototype design

|  |  |  |
| --- | --- | --- |
| **Event Name** | **Domain** | **Internal / External** |
| AccountStatementGenerated | Aps.IntegrationsEvents | External |
| BillingAccountDeletedFromCustomer | Aps.IntegrationsEvents | External |
| BillingCompanyAddedOpenClosedWindow | Aps.IntegrationsEvents | External |
| CrossCheckSessionCompletedSuccesfully | Aps.IntegrationsEvents | External |
| CrossCheckSessionCompletedWithErrors | Aps.IntegrationsEvents | External |
| CrossCheckSessionStarted | Aps.IntegrationsEvents | External |
| CustomerBillingAccountAdded | Aps.IntegrationsEvents | External |
| CustomerScrapeSessionFailed | Aps.IntegrationsEvents | External |
| IntegrationEvent | Aps.IntegrationsEvents | External |
| ScrapeSessionCompletedSuccessfully | Aps.IntegrationsEvents | External |
| ScrapeSessionCompletedWithErrors | Aps.IntegrationsEvents | External |
| ScrapeSessionDataInterpretered | Aps.IntegrationsEvents | External |
| ScrapeSessionDataRetrievalCompleted | Aps.IntegrationsEvents | External |
| ScrapeSessionDataValidated | Aps.IntegrationsEvents | External |
| ScrapeSessionDuplicateStatementReceived | Aps.IntegrationsEvents | External |
| ScrapeSessionFailedEvent | Aps.IntegrationsEvents | External |
| ScrapeSessionStarted | Aps.IntegrationsEvents | External |
| ScrapeSessionStatementComposed | Aps.IntegrationsEvents | External |
| ScrapingScriptUpdated | Aps.IntegrationsEvents | External |
| CrossCheckCompleted | Aps.Core | Internal |
| ScrapeSessionDuplicateStatement | Aps.Core | Internal |
| ScrapeSessionFailed | Aps.Core | Internal |
| ScrapeSessionSuccessfull | Aps.Core | Internal |
| BillingAccountAddedToCustomer | Aps.Customer | Internal |
| CustomerBillingCompanyAccountDeleted | Aps.Customer | Internal |

## H – Data Transfer Objects used in the APS Solution

Table 3: DTO’s in prototype design

|  |  |
| --- | --- |
| **Query Name & Target Domain** | **Returning DTO** |
| AllBillingCompaniesQuery  Aps.BillingCompany | IEnumerable <BillingCompanyDto> |
| BillingCompanyBillingLifeCycleByCompanyIdQuery  Aps.BillingCompany | BillingCompanyBillingLifeCycleDto |
| BillingCompanyByIdQuery  Aps.BillingCompany | BillingCompanyDto |
| BillingCompanyCrossCheckEnabledByIdQuery  Aps.BillingCompany | BillingCompanyCrossCheckDto |
| BillingCompanyOpenClosedWindowsQuery  Aps.BillingCompany | IEnumerable  <OpenClosedWindowDto> |
| BillingCompanyScrapingLoadManagementConfigurationQuery  Aps.BillingCompany | BillingCompanyScrapingLoadManagementConfigurationDto |
| BillingCompanyScrapingUrlQuery  Aps.BillingCompany | BillingCompanyScrapingUrlDto |
| ScrapingErrorRetryConfigurationQuery  Aps.BillingCompany | IEnumerable  <ScrapingErrorRetryConfigurationDto> |
| CustomerByIdQuery  Aps.Customer | CustomerDto |
| CustomerBillingCompanyAccountsById  Aps.Customer | CustomerBillingCompanyAccountDto |

## I – Web Interface mock-ups

Still to be added…

1. Web Interface mock-up
2. Customer Account Statement UI mock-up