Abstract and cover page

Table of Contents

[1 Introduction 1](#_Toc389915697)

[2 Design Methodology 1](#_Toc389915698)

[3 Domain Driven Design 1](#_Toc389915699)

[3.1 Components of Domain Driven Design 1](#_Toc389915700)

[3.1.1 Domains 1](#_Toc389915701)

[3.1.2 Sub Domains 1](#_Toc389915702)

[3.1.3 Generic Sub Domains 1](#_Toc389915703)

[3.1.4 Bounded Contexts 1](#_Toc389915704)

[3.1.5 Aggregates, Entities and Value Objects 2](#_Toc389915705)

[3.1.6 Repositories 2](#_Toc389915706)

[4 The Strategic Vision and Design of APS 2](#_Toc389915707)

[4.1 APS Domains and Responsibility Decomposition 2](#_Toc389915708)

[4.1.1 Core Domain 2](#_Toc389915709)

[4.1.2 Sub Domains 2](#_Toc389915710)

[4.1.3 Generic Sub Domains 2](#_Toc389915711)

[4.1.4 Integrating the different domains 2](#_Toc389915712)

[4.1.5 Queries, Single Responsibility Principle and Interface Segregation principle 3](#_Toc389915713)

[5 Tactical Design 3](#_Toc389915714)

[5.1 Domain and feature allocation 3](#_Toc389915715)

[5.2 Project structure 3](#_Toc389915716)

[5.3 Continuous Integration and Development Strategy 3](#_Toc389915717)

[5.4 Specifications by example and Test Driven Development 3](#_Toc389915718)

[5.5 Feature integration specification collaboration 3](#_Toc389915719)

[5.6 Object Orientation Principles 3](#_Toc389915720)

[5.7 Peer Programming and Code Review 4](#_Toc389915721)

[6 Challenges and successes 4](#_Toc389915722)

[7 Conclusion 4](#_Toc389915723)

[8 References 5](#_Toc389915724)

[9 Appendix 6](#_Toc389915725)

[A - Definition of terms or concepts used within the APS system: 6](#_Toc389915726)

# Introduction

The report that follows is

, mention how we took a DDD design methodology, which incorporates a strategic design, with an explanation of the components of DDD, using our overarching architecture and how all the systems fit together to illustrate.

Talk about the implementation of our solution at a tactical design level (but not too in depth )

# Design Methodology

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.* [1]

### 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.* [2]

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. [3] 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 [4], Application Services, the Integration and Published Language [4] 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 no longer is a Prototype, the solution can be deployed without them based on a customised Build Configuration [5].

### Fakes

The Fakes project is a collection of placeholder Implementations of all of the Aggregate Repositories in the solution. These placeholders facilitate testing functionality acting as rudimentary in-memory collection stores. The intention was to 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 the Domain Model to work independently of anything else.

### Application Services

An Application Service in the context of the APS Solution is a Class and 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 then stored 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* [4] as defined by Eric Evans employing the DTO Pattern [4] 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]

### Common Classes

The common classes or *Aps.DomainBase* project contains Base Classes used by all Domains as well as the *Caliburn.Micro Event Aggregator* allowing each Domain to have the ability to *channel events from multiple objects into a single object to simplify registration for clients* [6].

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 service from the Event raiser, as well as 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 [7] Group two, has applied SOLID programming principles. This section serves to highlight an example or two for some of the principles.

### Single Responsibility Principle

The Single Responsibility Principle (SRP) [8] states that *a class should have one, and only one, reason to change* [8]*,* 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 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* [10]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 Resolved by using the Service Locator Pattern (SLP) [11] to retrieve the Repository. The provision of the Repository has been made possible by using Autofac’s Inversion of Control container implementation. [12]

The premise for not using the SLP, was that the query Class need to have knowledge of *how* to retrieve the resource of Repository is a violation of the SRP. If the SLP implementation changed, the query Class would have a secondary reason to change, other than that 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) [9], even though it is not an Interface.

The ISP states that Software Developers should only *make ﬁne grained interfaces that are client speciﬁc* [10]*,* meaning that creating Classes that clients who use a Class or an Interface should not have Method calls or Properties that are not uses on the Class or Interface. In the Code Listing mentioned, the query does one thing alone. By looking at the naming of the queries in Figure 1 below, the prevalence of ISP can be seen.

In Code Listing 2 in Appendix D, the Repository Interface exposes more Methods than all consumers will use, however the purpose of the Repository was to abstract the data layer which we were using to facilitate our TDD approach. The queries using the Repository is where we mainly apply the ISP.

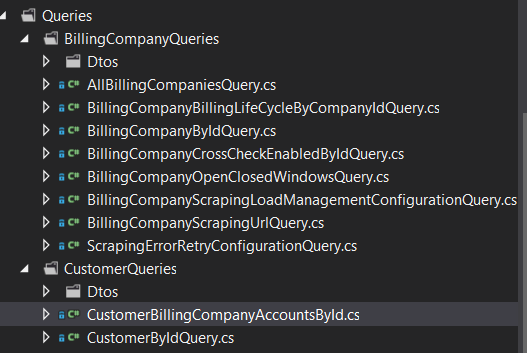


Figure 1- Specialised queries tailored to consumers

## Peer Programming and Code Reviews

### Integration sessions

### Code / Test Reviews

### 2 Peers Programming

# Challenges and successes

# Conclusion

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

# References

|  |  |
| --- | --- |
| [1] | 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]. |
| [2] | G. Hohpe and B. Woolf, Enterprise Integration Patterns - Designing, Building, And Deploying Messaging Solutions, Addision Wesley, p. 38. |
| [3] | M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee and R. Stafford, Patterns of Enterprise Application Architecture, Addison Wesley, 2002, pp. 347-351. |
| [4] | E. Evans and M. Fowler, Domain-Driven Design -Tackling Complexity in the Heart of Software, Prentice Hall, 2003. |
| [5] | Microsoft, “Understanding Build Configurations,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/kkz9kefa.aspx. [Accessed 09 06 2014]. |
| [6] | M. Fowler, “Event Aggregator,” [Online]. Available: http://martinfowler.com/eaaDev/EventAggregator.html. [Accessed 09 06 2014]. |
| [7] | Wikipedia, “Codebase,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Codebase. [Accessed 10 06 2014]. |
| [8] | J. Miller, “Patterns in Practice,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc546578.aspx#id0390008. [Accessed 10 06 2014]. |
| [9] | R. C. Martin, “Object Mentor,” [Online]. Available: http://objectmentor.com/resources/articles/dip.pdf. [Accessed 10 06 2014]. |
| [10] | J. Lewis, “The SOLID Principles,” [Online]. Available: http://joshilewis.github.io/ELEN7045/elen7045-solid.pdf. [Accessed 10 06 2014]. |
| [11] | Microsoft, “The Service Locator Pattern,” [Online]. Available: http://msdn.microsoft.com/en-us/library/ff648968.aspx. [Accessed 10 06 2014]. |
| [12] | Autofac, “Autofac,” Autofac, [Online]. Available: http://autofac.org/. [Accessed 10 06 2014]. |
| [13] | Microsoft, “Class Library,” Microsoft Developer Network, [Online]. Available: http://msdn.microsoft.com/en-us/library/d11h6832(v=vs.71).aspx. [Accessed 09 06 2014]. |

# Appendix

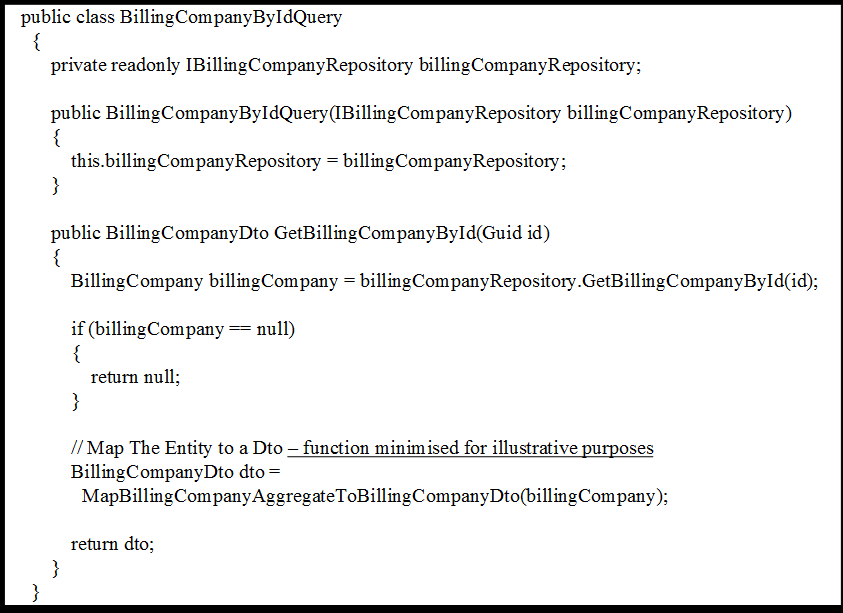
## A - Definition of terms or concepts used within the APS system:

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

