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

, 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

Domain Driven Design (DDD) is the philosophy of modelling the Software Engineering world as closely as possible to the real world. This means that the Software Engineer closely analyses the system components during the design of the solution with a view of simulating the real world as closely as possible in the system that is being built. This happens before the Software Engineer starts to design an architecture or code a solution. DDD is about understanding the problem domain completely in a ubiquitous language. This ubiquitous language allows all stakeholders to speak on the same level, without any technical jargon or confusion. You can see the ubiquitous language used in the APS Prototype in Appendix A.

DDD is also a collection of principles and patterns that, if applied in the correct manner, can lead to software abstractions that are referred to as Domain Models. (Explained further in Section 3.1.1) These Domain models can then be sub-divided in to areas of expertise, known as Sub-Domains and Generic Domains. Sub-Domains and Generic Sub-Domains, as well as the other components and building blocks of DDD, are explained in a little more detail in sections 3.1.

The APS Prototype was a complex set of business rules that with substantial analysis proved to be a worthy project of this approach. DDD allowed the development team to analyse the problem domain in a different manner than usual, focusing specifically on the Aggregates, Entities and Value Objects. This DDD approach allows for the development team to easily make the necessary changes to the Prototype in order to move it forward into a Production system. This manner in which it could easily be modified and enhanced is further discussed in the APS Design section. (Section 4)

DDD is the design component of the project solution, and once the design has gone through multiple iterations of discussions with all the stake holders, the project’s requirements start to materialize. One of the methods for documenting these requirements is to use Specifications by Example. Specifications by Example inspire a positive environment for Test-Driven Development (TDD) and for that reason, the APS Development team chose to use Specifications by Example for the Requirements Analysis of the Business rules.

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. The process steps in which Test-Driven Development 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.

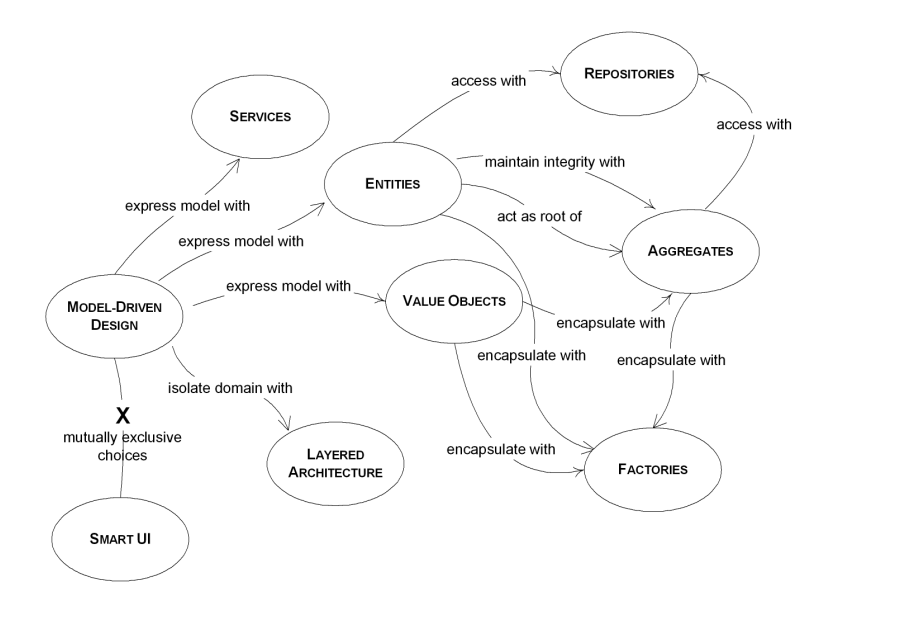


Figure - Domain Driven Design Building Blocks

## Components of Domain Driven Design

### Domains/Core Domain

The domain in the DDD approach is the actual problem statement that the system is being built to solve. This such domain is then modelled into a Domain Model and this Domain Model should align to the business processes and strategies. Core Domains typically are the most valuable and specialized concepts in the Domain model [5], and this Core Domain deserve the utmost focus in design.

The Domain in the APS Prototype was the need for an Account Payment System that would allow customers to log into one central website to view all of their billing accounts.

### Sub Domains

Typically, in large Problem Domains, the Domain Model can stretch and grow to be extremely large. In such cases, the Domain Model can be broken down into Sub-Domains. Sub-Domains are the areas of expertise within the Core Domain and add a logical separation of responsibilities. Having these Sub-Domains allows focus to be cast on logical, manageable chunks of the solution, without losing the strengths of the DDD approach. These Sub Domains are integral parts of the system, and without them, the system may not work as desired.

### Generic Sub Domains

Generic Sub Domains are Sub Domains that lack any need to be a specialized Sub Domains. Generally, these Generic Domains tend to be 3rd party software that the system is dependant on. Yet, however generic they may be in nature, the Core Domain and Sub Domains are fully dependant on them in order to complete the solution. Without these components, the system and project will fail.

### Aggregates, Entities and Value Objects

DDD contains some fundamental building blocks. These building blocks can be seen in Figure 1. Some of the more important building blocks, being Aggregates, Entities and Value Objects are briefly described below.

#### Aggregates

Martin Fowler defines an Aggregate as:

‘*A DDD Aggregate is a cluster of domain objects that can be treated as a single unit.*’[6]

An Aggregate provides a consistency boundary where, all requests for changes to it, or its child objects (entities or value objects), are requested through it. In turn,

direct requests to where it should be, with validations along the way (either by itself or its entities), primarily because the other entities should have no concept of each other,

but "something" should. This Aggregate also acts as an Anti-Corruption layer for the Aggregate, by not allowing any other component of the system direct access to its Entities and Value Objects. This cluster of domain objects that lives as a single unit will contain one component object that is considered to be the Aggregate root.

#### Entites

Within Aggregates, there may be the need to identify certain properties of the Aggregate. Because this property has an identity, it is no longer a stand-alone property. This property is termed an Entity as it can be identified within the Aggregate. An example of this is the *CustomerBillingCompanyAccount* Entity that is located within the *Customer* Aggregate in the APS Prototype. This property started off initially by being a Value Object, but through the iterations of our TDD, it materialized that there would need be a need to identify the *BillingCompany* within each *CustomerBillingCompanyAccount*. With the addition of this identity, this converted the Value Object (Discussed in Section 3.1.4.3) into an Entity.

#### Value Objects

Value Objects are immutable properties within the higher Aggregate grouping. This immutability means that the Object value cannot be changed. These Value Objects do not have identity, thus having an Interchangeable property, and can easily be swapped out with another Value Object of the same type. For Example, if there was a ‘*Money*’ Aggregate, then a R5 coin would be the Value Object within that Aggregate. You could swap out that R5 coin with any other R5 coin and the Aggregate would not be affected. Also, you would not be able to change that R5 coin into a R10 coin due to its immutable property.

### Bounded Contexts

Bounded Contexts are the boundaries in which the Sub Domains lie. Each Sub Domain will have its own Bounded Context, and these Bounded Contexts encapsulate the Sub Domain completely. A Bounded Context can be defined as:

*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]

All of the Entities, Value Objects, Repositories, Factories and other building blocks in the Aggregate reside within the Bounded Context for that Aggregate.

### Repositories

During the Requirements Analysis and Design of the APS, it was discovered that there was a need for Repositories. [] A decision was made by the project stakeholders that the Repository design pattern would be used within the project.

The Repository design pattern is used to create an abstraction between the domain layer and the data layer. [] In layman’s terms, this mean that there is no need for components in the domain layer needing the data to know where the data actually comes from. This data could come from a Database [], a Repository [], or it could even just be a Fake Repository. []

The APS project that was constructed makes use of a few Repositories, but seeing as it was a Prototype project, these Repositories have been made Fake Repositories. If the Prototype was successful, it would then be simple enough to swap out the Fake Repositories with a proper data structure. (ie. A Database or a Repository.)

# 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

## 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

Explain solution and project structure

## 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

Mention how the implementation follows good OO principles

Dependency injection for DIP – relying on abstractions vs concretions

Single Responsibility – one reason to change examples

ISP – queries tailored to consumers at the lowest level.

## 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] | J. Palerma, "Guidelines for Test-Driven Development", Microsoft Developer Network, [Online]. Available: <http://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx> [Accessed June 2014] |

[5] E. Evans, “Domain Driven Design Reference”, Domain Language Inc., [Online]. Available: <https://domainlanguage.com/ddd/patterns/DDD_Reference_2011-01-31.pdf> [Accessed June 2014]

[6] M. Fowler, “DDD\_Aggregate”, Martin Fowler, [Online]. Available: <http://martinfowler.com/bliki/DDD_Aggregate.html> [Accessed June 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

