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)

# The Strategic Vision and Architecture 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 (*[*http://msdn.microsoft.com/en-us/magazine/hh394145.aspx*](http://msdn.microsoft.com/en-us/magazine/hh394145.aspx)*)*. All the components would have been coupled to each other (albeit potentially just in the same application). Avoiding the 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. (same article)*

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.*[?]

During the requirements gathering process it became apparent that in order for to design to be loosely coupled, the registrations, billing company and scheduling components of the system needed to be isolated from each other. **The final architecture G2’s prototype design can be seen in appendix B**. The implemented design has multiple applications working together to accomplish the business requirements. The business requirements were divided into multiple application domains. These domains are called Core-, Sub- and Generic Sub domains. The different domains within the implemented design break the requirements into separate areas of responsibilities that deal with domain data in different ways. This domain architecture was adopted from Martin Fowler’s article [?]:

*In short, the Microservice architectural style*[*[1]*](http://martinfowler.com/articles/microservices.html#footnote-etymology)*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.*

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 separately on its own but work 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 is 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.

### Core Domain

The Core Domain needs to define what the primary business offering is. After analysing the system requirements it was identified that the main business objective is to be able to collate and create statements for customers from multiple external e-billing providers. Any changes to the business should flow out of this domain.

The architecture diagram shows the elements contained in the core domain. The component within the core domain that contains the business logic is called the Application Services. These services provide the functionality to retrieve, validate and store customer statement data obtained via the third party scraper.

### Sub Domains

The sub domains can be identified as important business elements that are not core to the business, but supports the core business strategy. The business needs identified belonging to this domain is the customer registration/maintenance operation and the external Billing Company maintenance. Figure x.x in appendix B shows the component layout of the sub-domain. The Application Services within the sub-domain provides all functionality required to operate and maintain the Customer and Billing Company operations and data.

### Generic Sub Domains

The final domain in our architecture is called Generic Sub Domain. Generic sub domains contain elements that are not directly developed internally but are needed by the system to provide the required functionality. An example of an element that falls into this domain is the 3rd party scraper.

A set of rules and standards used by the Generic Sub Domains to communicate with other business domains could be identified and implemented. This would enable the business to easily exchange the specific elements contained by the Generic Sub Domain with others. This could enable the business to make use of a different third party scraper without the need to make changes within the other domains.

### Integrating the different domains

The three different types of domains discussed in the previous sub-sections work independently from each other. Although each domain executes and maintains its own data, it provides little to no 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 provide the business objective.

The different domains communicate with each other through an integration mechanism called the Event Integration Service. This service provides cross-domain logging, as well as a common language between the different domains. The service works by passing messages between the various domains’ application services. These messages contain information regarding an event that has taken place within one of the domains. The event Integration Service resides within every domain as illustrated in figure x.x in Appendix 2.

The architecture diagram shows a *DTO\_Query* element inside the core domain. This element is responsible for the retrieval and translation of non-event data from the all domains to the core domain. This means that whenever an Application Service within the core domain requires data from another domain, it would use the DTO\_Query to retrieve that data.

Proposal for sections in Section 5:

Explain how Event Aggregator works to provide internal domain communication.

Explain how Intergration Event service works with Application Services that registers with it to receive events.

Have appendixes that shows all internal Events in core domain and sub domain

Have appendixes that shows all external Events and queries

Have appendixes that shows all queries, and discuss alternatives such as nServiceBus or MassTransit

Add something about how repository was implemented? Might need to add that to section 4?

The data persistence of all domain elements in the prototype’s design is also generic by means of the repository pattern [msdn ref]. *The repository mediates between the data source layer and the business layers of the application. Expand.* I would talk about how we could use messaging architectures and an enterprise service bus, such as nServiceBus or MassTransit. These options are still feasible if our solution grew, and we could replace our eventIntegrationService with the messaging infrastructure, quite easily by swapping out the implementation in the Dependency container.

the trade off here was simplicity, logging, event versions and making the consumer responsible for knowing when/where they were in terms of messages

or events. adding a message bus and / or queueing mechanism was an unnecessary complexity

**Stuff moved out of section 4 that should move into section 5:**

These Objects communicate with each other internally through an *eventAggregator* messaging service. Internal commination means that the messages passed between the different Application Services stay within the domain and does not leave the bounded context of the domain. The messaging service allows a single application service within the core to simultaneously publish and receive messages from other application services within the core domain without directly coupling them. Each application service in the core domain has its own repository to persist data created by it.

This is represented by the Repository and domain Data objects in figure x.x in appendix B.

Due to the fact that this is only a prototype aimed at providing a proof of concept, both the customer and external billing company sub domains have been integrated into a single domain to keep the implementation as simple as possible. It is however important to note that the customers and billing Companies are both scalable and act autonomously. This is a huge business benefit as there would be a very high probability that additional external companies would be added throughout the life-cycle of the project. It is suggested that a future production release should have these two subdomains run as different applications on possibly different servers.

The different application services in the sub domain could internally communicate with each other the through the event Aggregator in the same manner as the core domain does. Each of the Application services in the sub-domain also makes use of a repository and domain data storage to persist data generated by that specific service.

A list and description of all application service’s and internal events within the sub domain can be found in Appendix ?. <Need to add appendix>

The Event Integration Service works by sending event notifications between the different domains.. An Application Service could subscribe to any event it is interested in listing for with the Event Integration Service. The Event Integration service for that specific domain then keeps a list of all currently registered events. As soon as the Event Integration service receives an event it immediately stores it in the Event Store. The Event Integration service then checks if there are any listeners for that specific event in that specific domain. If there is at least one listener, it broadcasts the event in that domain via the *eventAggregator*. If there are no listeners it does not broadcast the event until at least one listener in the domain subscribes.

The Event Integration service could be swapped out for a different one, but due to the fact that is exists on the edge of every bounded context, it would require modification to all domains.

A list of external Events and the Application services that use them can be found in Appendix ?.

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

External events discussed in the previous section are generated only when a service in one domain performs an operation that a different service in another domain is interested in. This provides some of the cross-domain functionality needed but not all. During a typical scraping operation the 3rd party scraper interacting with the core domain requires the URL address for the external billing company and the user login credentials for that user for that specific account. The URL is stored in the Repository of the billing Company sub domain and the user’s login details for that billing account is stored in the customer repository also in the sub domain.

Discuss an Alternative design?

There is no way that the core domain could directly access the repositories of the sub domain, although it needs the data to perform the required actions. A possible design option could be to make use of event passing discussed in the previous section to enable the core domain to access the sub domain repositories. Whenever the core domain needs the URL it could publish a requestURL external event. The Billing Company sub domain could then listen for this event and when it receives the request it could retrieve the URL from the repository and publish an external event with the URL and some identifier to link it to the response message to the original message.

The design option discussed in the previous paragraph could work theoretically but will cause a lot of messages being passed between the different domains unnecessarily. It would also violate the single responsibility principle on the Event Integrations Service as it would now also be responsible to send and track data specific events above normal action events. It would also increase the complexity of the EventIntegration Service and make future maintainability more difficult.

An alternative design that was implemented is to use queries together with the DTO pattern to distribute data between different domains. Figure x.x in appendix B shows QueriesReturningDTOs on the edge of the core domain drawing. The application service in the core domain will use a specific Query to retrieve the required data form another domain. The Query will access the required Repository and retrieve the requested data. It would however not return the Repository data directly to the core domain, it would first convert it into a DTO that the core domain understand and then return the DTO.  The DTO provided only contains the specific data requested by the core domain and will not provide other unnecessary data. Extra queries and DTO could easily be added in future releases of the system.

A list of all Integration Queries together with the DTOs they would return is shown in appendix X.

When discussing the implementation of the

# 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] | J. Lewis, “The SOLID Principles,” [Online]. Available: http://joshilewis.github.io/ELEN7045/elen7045-solid.pdf. [Accessed 10 06 2014]. |
| [10] | Microsoft, “The Service Locator Pattern,” [Online]. Available: http://msdn.microsoft.com/en-us/library/ff648968.aspx. [Accessed 10 06 2014]. |
| [11] | Autofac, “Autofac,” Autofac, [Online]. Available: http://autofac.org/. [Accessed 10 06 2014]. |
| [12] | 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]. |
| [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]. |
| [14] | R. C. Martin, “Object Mentor,” [Online]. Available: http://objectmentor.com/resources/articles/dip.pdf. [Accessed 10 06 2014]. |

# Appendix

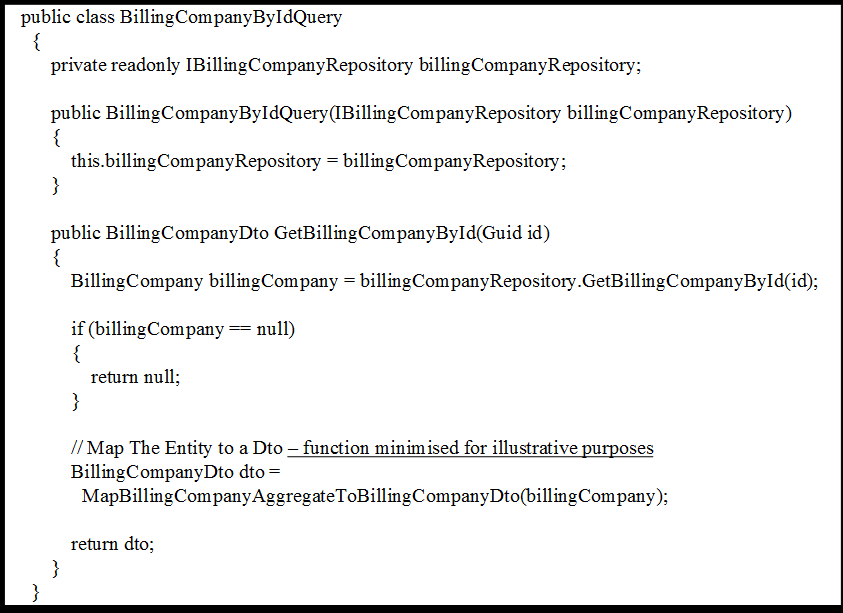
## A - Definition of terms or concepts used within the APS system:

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

