**An Introduction to Domain-Driven Design - Part 1: The Strategic Tools**

In this introductory article architects & engineers will gain an understanding of what Domain-Driven Design (DDD) is and how it's strategic tools can be leveraged to improve software engineering practice

**Introduction**

Domain-Driven Design (DDD) is the concept that the structure and language of your software components should match the business domain. DDD connects the software implementation to the evolving model of the business and is predicated on the following goals [1]:

* Placing the project's primary focus on the core business domain and its logic
* Basing complex designs on a model of the domain
* Initiating a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems

The term, first coined by Eric Evans [2], combines the following concepts into a methodology [1]:

* **Context**: The setting in which a word or statement appears that determines its meaning
* **Domain**: A sphere of knowledge (ontology), influence, or activity. The subject area to which the user applies a program/system is the domain of the software
* **Model**: A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain
* **Ubiquitous Language**: A language structured around the domain model and used by all team members to connect all the activities of the team with the software

**Strategic versus Tactical patterns**

Domain-Driven Design proposes strategic as well as tactical design patterns. Strategic design patterns (domain, subdomain, context map, anti-corruption layer, bounded context) are used to guide the use of tactical design patterns1 throughout a project and help answer questions around build versus buy as well as assisting with questions over team composition (permanent versus contract, senior versus more junior).

While the strategic patterns focus on the big picture (e.g interaction between sub-domains and contexts, the tactical patterns focus on the domain model within a context. Unfortunately, many teams that adopt DDD first jump right in to the tactical design patterns without a thought for the strategic ones, thereby missing out on key benefits that DDD provides.

1 Tactical design patterns, which are the subject of another of my ABC Technology Academy (DTA) articles, include: entities, aggregates, value objects, domain (events, stores, services, registries, repositories), application services, infrastructure services, et al

**Design and Agile**

Evans' initial ideas have been extended in recent years by the community that has grown up around the practice with notable enhancements concerning architecture (e.g. hexagonal instead of layered), domain events and patterns for concrete implementation found in sources such as [4] [5] [3].

Domain-Driven Design maps directly to the concepts of Agile and NoSQL. There is nothing in Agile to suggest that one should skip design. Instead agile suggests that design should be evolutionary and iterative. DDD encourages an iterative process, first at the strategic level to divide the work and focus on what is important to the business and then at a tactical level to understand the details of each context [6].

**Strategic patterns**

This section provides a brief overview of some of the strategic patterns so that content presented in later articles and documents can be more readily understood.

**Organizational patterns**

A number of DDD organizational patterns exist which include [2]:

* **Partnership**: "When teams in two Contexts will succeed or fail together, a cooperative relationship needs to emerge. The teams institute a process for coordinated planning of development and joint management of integration. The teams must cooperate on the evolution of their interfaces to accommodate the development of both systems. Interdependent features should be scheduled so that they are completed for the same release" [3].
* **Shared Kernel**: "Sharing part of the model and associated code forms a very intimate interdependency, which can leverage design work or undermine it. Designate with an explicit boundary some subset of the domain model that the teams agree to share. Keep the kernel small. This explicit shared stuff has special status and shouldn't be changed without consultation with the other team. Define a continuous integration process that will keep the model tight and align the Ubiquitous Language of the teams" [3].
* **Customer-Supplier Development**: "When two teams are in an upstream-downstream relationship, where the upstream team may succeed interdependently of the fate of the downstream team, the needs of the downstream team come to be addressed in a variety of ways with a wide range of consequences. Downstream priorities factor into upstream planning. Negotiate and budget tasks for downstream requirements so that everyone understands the commitment and schedule" [3].
* **Conformist**: "When two development teams have an upstream/downstream relationship in which the upstream team has no motivation to provide for the downstream team's needs, the downstream team is helpless. Altruism may motivate upstream developers to make promises but they are unlikely to be fulfilled. The downstream team eliminates the complexity of translation between bounded contexts by slavishly adhering to the model of the upstream team" [3].

**Domains**

In DDD a **Core Domain** is a part of the business domain that is of primary importance to the success of the organization, the part that the business must excel in to win in the marketplace. Projects in the Core Domain should get the highest priority and be staffed with domain experts with deep knowledge of that subdomain, the best architects & developers and as much leeway and leverage as possible to give the close-knit team an unobstructed success path [3].

DDD also introduces Supporting Subdomain and Generic Subdomain. If a domain models a part of the business that is essential and somewhat specialised, yet not core, then it is referred to as a **Supporting Subdomain**. If the subdomain captures nothing special to the business, yet is required as part of the overall business solution, it is a **Generic Subdomain** [3]. From the business' perspective there will be little competitive advantage to developing in-house solutions for a Generic Subdomain and hence vendor solutions may be bought instead. One situation where the business may prefer to develop an in-house solution is when the vendor licensing and/or operating costs are considered too expensive over the long term and hence it is considered cheaper to build instead of buy.

For the purposes of the Legacy-to-EPP data migration, Garry Smith (the author of this article), also identifies the Legacy subdomains; legacy systems that either support EPP by providing the source of truth for an ongoing data ingestion exercise, or are supported by EPP as the CPCD becomes the source of truth and the flow of data synchronization is reversed. In addition, External Subdomains are identified which are external to ABC and may be partner organisations, service providers or tenant customer systems

**Gaining the right focus**

Being supporting or generic does not imply that these domains are unimportant to the business, they are important, but there is no need for the business to excel in these areas. It is the Core Domain that requires excellence in implementation since it will provide distinct advantages to the business [3].

The EPP initiative is clearly a strategic investment for ABC aiming to provide not only best in class payment services, but also a flexible product that can be offered to the market and which internally is intended to underpin the ability to rapidly respond to market opportunities. Therefore, read the current subdomain designations in the Context Maps below from the point of view that tomorrow some of the non-core domains could themselves become core as the business innovates in new directions.

In order to help determine where best to employ development effort (build versus buy), staffing (permanent versus contract - for optimal knowledge retention; senior versus junior) and when to use DDD tactical modeling patterns versus other more traditional approaches (e.g. CRUD), it is important to understand the business context, not only the high level interactions between the various subdomains, but also the classification of those subdomains (core versus supporting versus generic).

**Bounded Contexts**

A bounded context is an explicit boundary within which a domain model exists and is an essential solution space modelling tool for DDD. Inside the boundary all terms and phrases of the Ubiquitous Language have specific meaning within the model2 which exactly reflects the language. Thus a Bounded Context is primarily a linguistic boundary [3], but can also be formed around organisational structure (see Conway's Law [7]), intentionally or otherwise!

2 Although the model can be depicted by diagrams it is physically embodied in source code and not just the underlying data storage schema

DDD best practice is to ensure that there is a one-to-one mapping between a subdomain and a bounded context, but acknowledges this is often not possible outside of greenfield development. A situation that often occurs is where multiple bounded contexts are modelled within a single sub-domain. In this instance, because the contexts do not overlap subdomains, the end result may be acceptable.

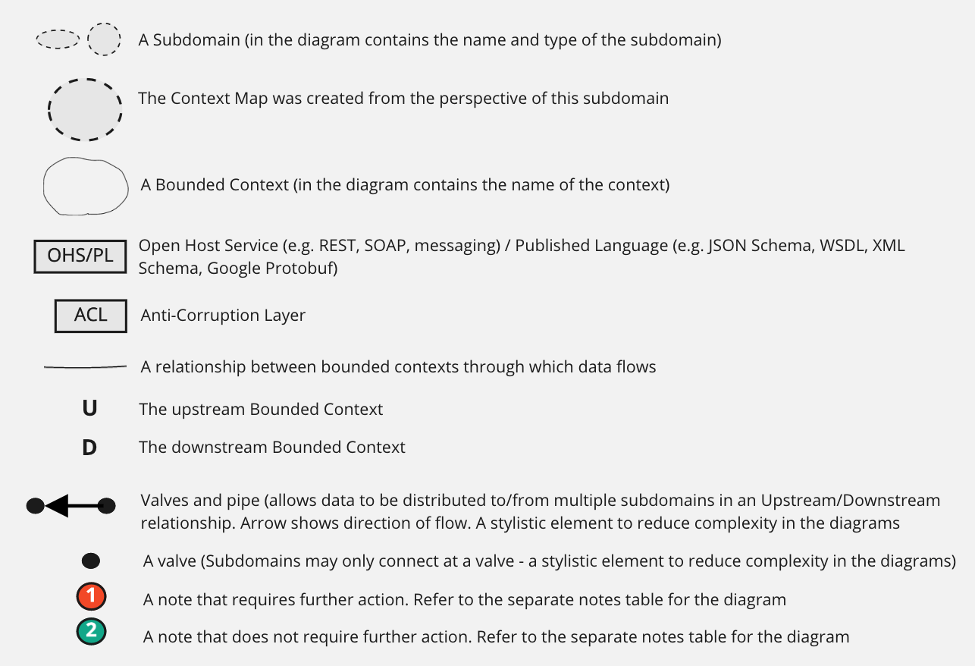
At the other end of the scale DDD identifies a single bounded context covering multiple subdomains as an anti-pattern; teams often find themselves needing to address this undesirable situation when dealing with legacy systems or when they have failed to first properly apply strategic design. The problem manifests because now the model needs to cater for the nuances of different subdomains, meaning becomes subjective and vague, with elements optional in some subdomains and conditional or mandatory in others.

A key tenet of DDD is that a single team (business and development) is responsible for all aspects of a given bounded context3; It is considered an anti-pattern to allow multiple teams to make changes as the precise nature of the language and implementation of the model can become corrupted/blurred. Direction given by Amir Arooni, ABC's Chief Information Officer, to establish product-based teams as part of the architectural runway program established in 2020 aligns with this DDD philosophy.

3 This is not to say that development work could not be sub-contracted out to near shore teams to increase development velocity as long as the product team remain closely involved and additional levels of product owners are not brought in. Communication should be kept direct so focus, vision and approach are not lost.

**Context Maps**

Context Maps are a DDD strategic modelling pattern that visualize domain, subdomain, bounded contexts, interactions between subdomains and the basis of that interaction denoting upstream and downstream relationships as well as concepts such Open Host Services (OHS), Published Languages (PL) and Anti-Corruption Layers (ACL) [3].



*Figure 1: Context Map Key*

Open Host Services could be implemented as a REST API, a message-based system or both. Published Language refers to a published data format represented in for example, JSON Schema, XML Schema or Google Protocol Buffers [3]. From the CPCD perspective the Published Language contains the canonical model that is used across the EPP Foundations platform and which aims to ensure consistency in naming and types of literals across the bounded contexts present in the Payment Services domain; the Payment Services data dictionary provides governance over naming and style [8].

A client can specify its preferred Published Language (the json, protobuf, etc that convey a context's ubiquitous language) which may help to decrease the overhead associated with transferring and parsing large amounts of data.

Anti-Corruption Layers can be implemented within Domain Services in downstream contexts or placed behind a downstream Repository interface. The synchronized state is the limited, minimal attributes of the remote models that are needed by the local model so that the inbound data aligns correctly with the concepts present in the downstream model.

**DDD & EPP Customer Profile & Configuration Data (CPCD)**

For the CPCD integration, Smith is proposing teams to cooperate using the Customer-Supplier model. If any team feels that they are being forced into a conformist model then Smith should be notified immediately as that sort of relationship tends to be rather negative and would need to be addressed. The Customer-Supplier model does not imply that requesting teams will always get everything that they ask for because architectural constraints (e.g. platform security, performance and stability) come into play, but the relationship is collaborative and solutions are actively sought to support the business use cases.

The teams will employ Open Host Service, Published Language, and, perhaps at first glance rather surprising given the comment about collaborative Customer-Supplier teams, Anti-Corruption Layer integration patterns. This is not a contradiction because even though the teams are establishing open standards between the Bounded Contexts, there is a need to ensure that downstream models are not polluted with foreign concepts, relationships and unnecessary information [3] [4].

Given the nature of the CPCD is to underpin the entire EPP platform, more data will typically be provided to clients (in a party profile for example) than they need and so their Anti-Corruption Layer will perform the filtering suitable for their own model thereby realizing the benefits of isolation but with less complexity than when dealing with a Big Ball of Mud. Indeed, given the collaborative nature of the Customer-Supplier relationship, downstream Anti-Corruption Layers will be simple, elegant and a thing of beauty!

An ideal in DDD is to ensure that downstream subdomains are not highly dependent on upstream subdomains, i.e. not reaching out to the upstream bounded context to request information every single time it is needed [3]. Instead, the ideal is for downstream bounded contexts to be autonomous, strategically pulling in data, for example just the recent changes that have occurred and which are in effective for a given Settlement window.

**Payment Services EPP Context Maps**

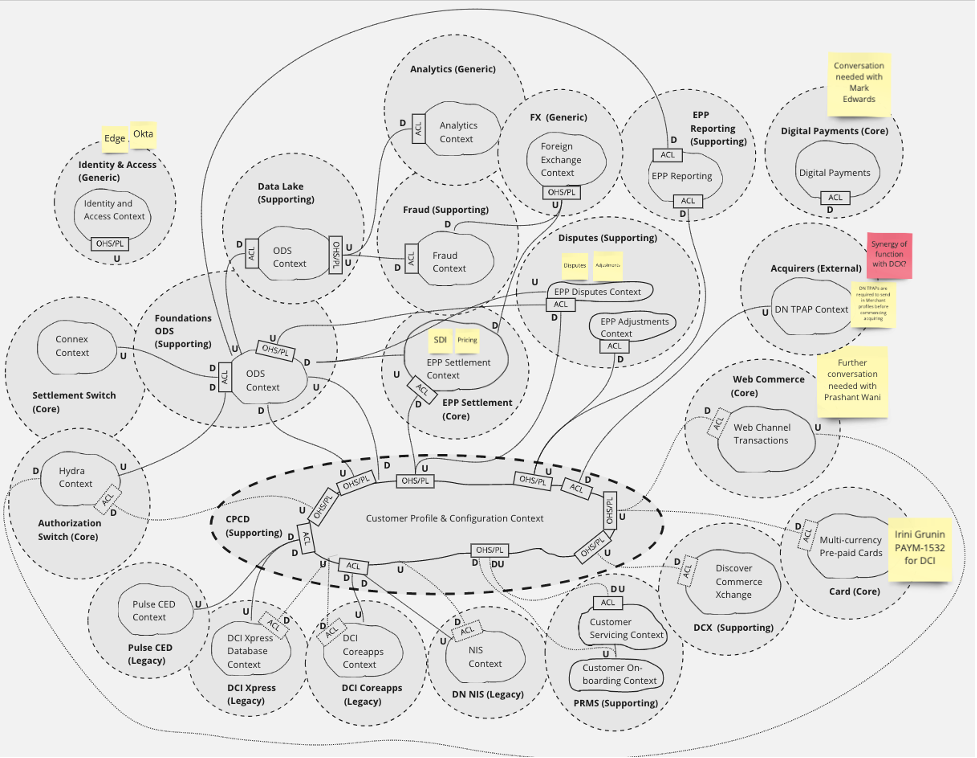
This section contains Contexts Maps that represent the Payments Services Domain from the point of view of the CPCD and related contexts (i.e. EPP Merchant and EPP Reference data) - please refer to Figure 2 for the notation used.

Zoomed Context maps that provide greater detail for each of the subdomain use cases will be covered in the article **An Introduction to Domain-Driven Design - Part 3 - Anti Corruption Layers and Zoomed Context Maps**

**Note:** The following context maps were created during the spring/summer of 2020 and hence represent a snapshot in time.

**CPCD is not a single sub-domain or bounded context**

The traditional view of the CPCD is it comprises of i. a database that contains all of the data (party, merchant and reference) needed to support transaction processing, ii. REST based services that act as gateways to the data, and iii. event publishing capabilities. Working on that definition, one may naively create a first cut of a context map that lumps all of the CPCD capability into a single sub-domain, as shown in Figure 2.



*Figure 2: Payment services context map (traditional view)4*

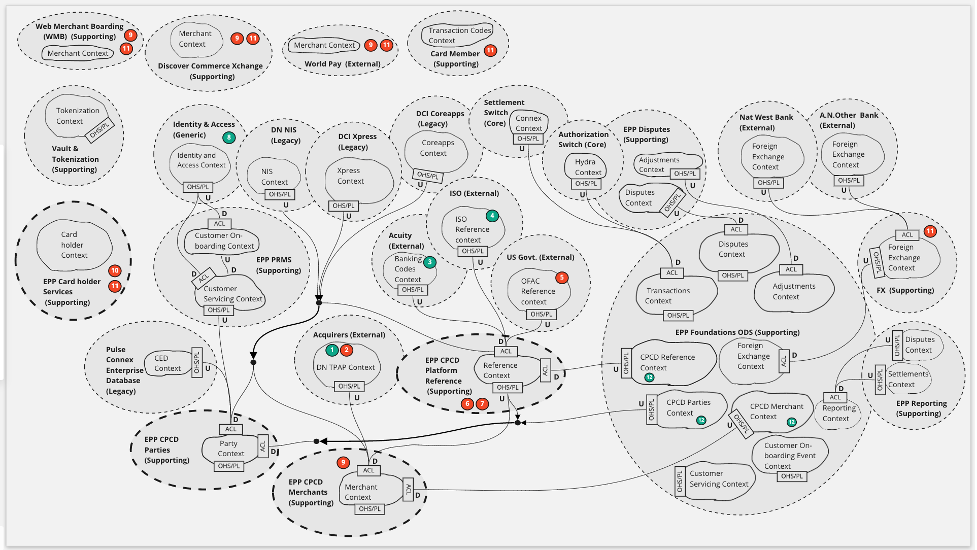
However, the more one considers the characteristics of the domain in terms of business processes, transactional consistency requirements, life-cycle management, et al, one may realize that the first and 'obvious' approach may not be the best! Event Storming (the subject of another article) combined with Context Map analysis proves to be a very powerful combination for getting the right balance when it comes to determining sub-domains and bounded contexts.

4 Showing subdomains, bounded contexts and interactions as originally envisaged through mindset purely focused on technology verticals. The direction of data flows are indicated by U (upstream) and D (downstream) between components. Note this diagram does not respect gravity as some upstream subdomains are pushing data uphill to reach a downstream consumer! Not very clear to say the least!

**'CPCD' segregated into multiple subdomains**

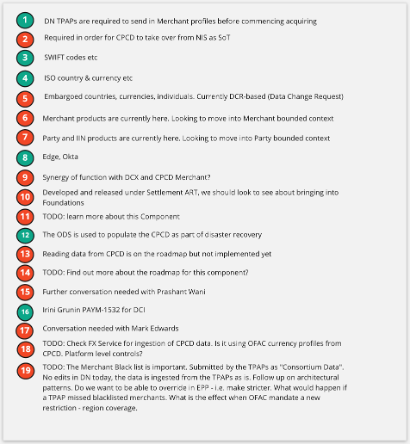
Domain-driven Design advocates the use of separate bounded contexts where the vocabulary of the various stages of an entity's life-cycle differ. For example, Vaughn [3] presents the use case for a book publishing company and points out that there is not a single way to properly model a book throughout the publishing process which ranges from conceptualization and proposal; contracting with authors; managing the editorial process and authorship; designing the book layout; translating into different languages; producing physical and electronic editions, marketing, selling to resellers and shipping.

This section contains Context Maps showing the segregation of the original 'CPCD' sub-domain into three sub-domains based around i. party-related data, ii. merchant data and iii. reference data. The approach provides greater clarity by allowing us to reason about the different bounded contexts explicitly. The notes associated with each of the diagrams are shown in Figure 4.



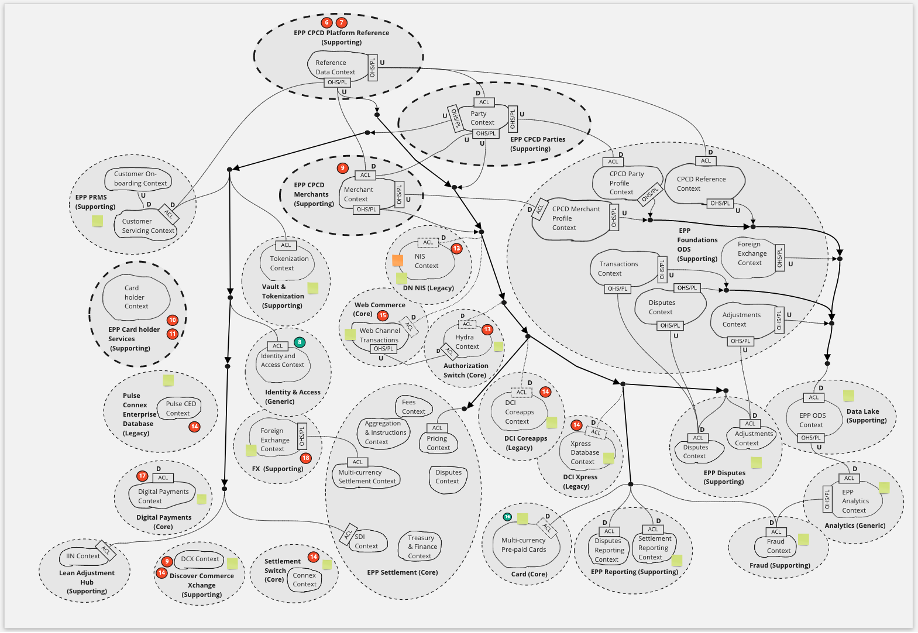
*Figure 3: Context Map focused on the flow of data from upstream into the party, merchant and reference data subdomains5*

5 This figure respects gravity, so upstream subdomains are shown above downstream consumers. Note: The PRMS subdomain is still external to the corresponding 'CPCD' subdomains but could/should become incorporated under a Product-based design



*Figure 4: A key to the notes used to annotate the Context Maps shown in Figure 3 & Figure 56*

6 from a documentation point of view, tracking open questions and clarifications in note form with the diagrams is helpful when identifying opportunities and gaps in current process/thinking



*Figure 5: Context Map focused on the flow of data from the CPCD into downstream subdomains7*

7 This figure respects gravity, so upstream subdomains are shown above downstream consumers. Although some downstream consumers of ODS data are depicted, the context is from the point of view of the CPCD subdomains (bold dashed ellipses), hence notably the flow of transaction events into the Settlement subdomain is not shown

**Context Maps are not enterprise diagrams**

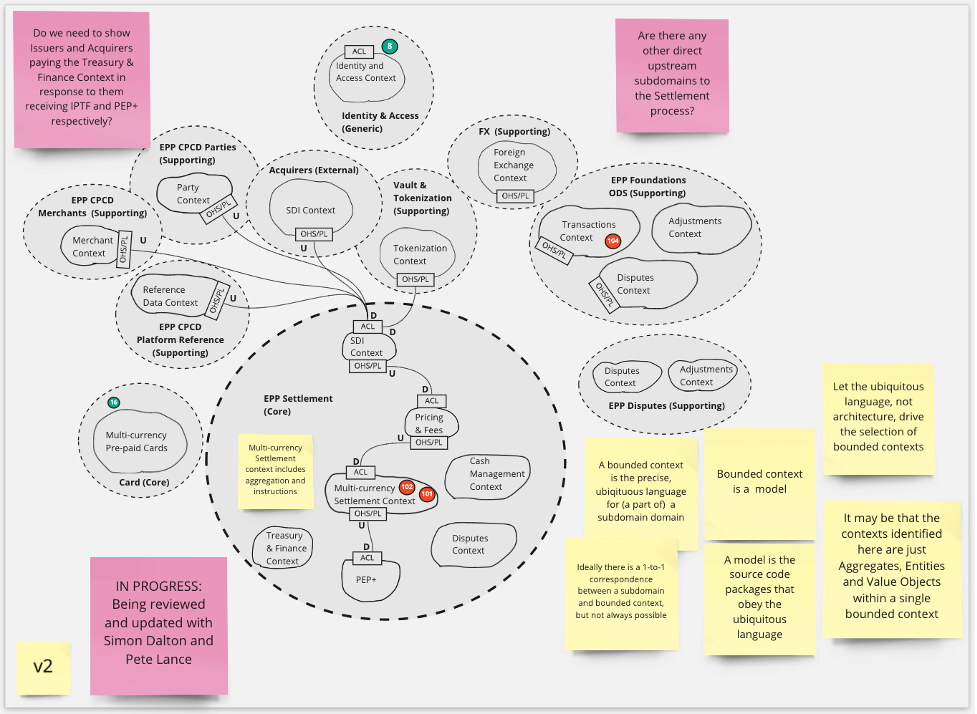
As previously noted, Context Maps are created from the perspective of a particular domain and as such a single diagram should not map out the entire enterprise. A strength of Context Maps is that because they are created from a particular perspective, sub-domain architects can come together to reason about the interactions between different bounded contexts and identify contradictions. This is especially true when designing/reviewing anti-corruption layer functionality as part of tactical design work.

From the perspective of a 'CPCD' architect working with colleagues in different domains, collaboratively creating/reviewing Context Maps and tactical designs helps enormously with shared understanding which usually leads to a far superior solution than would otherwise have been achieved.

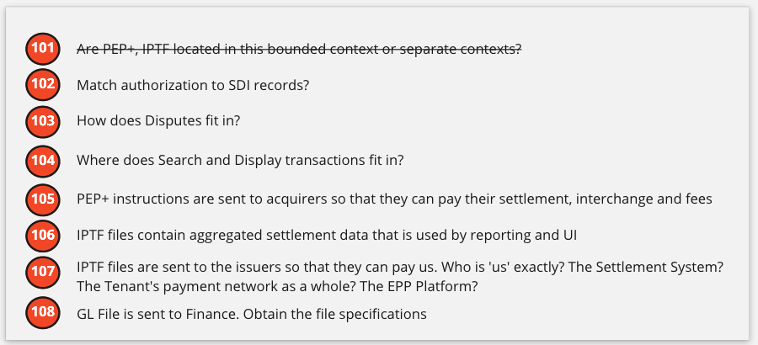
**Example Context Maps from the EPP Settlement and EPP PRMS UI perspectives**

The Context Maps in this section were created during the first part of 2020 and represent initial work focused on CPCD, EPP Settlement and User Interface (PRMS) integration.

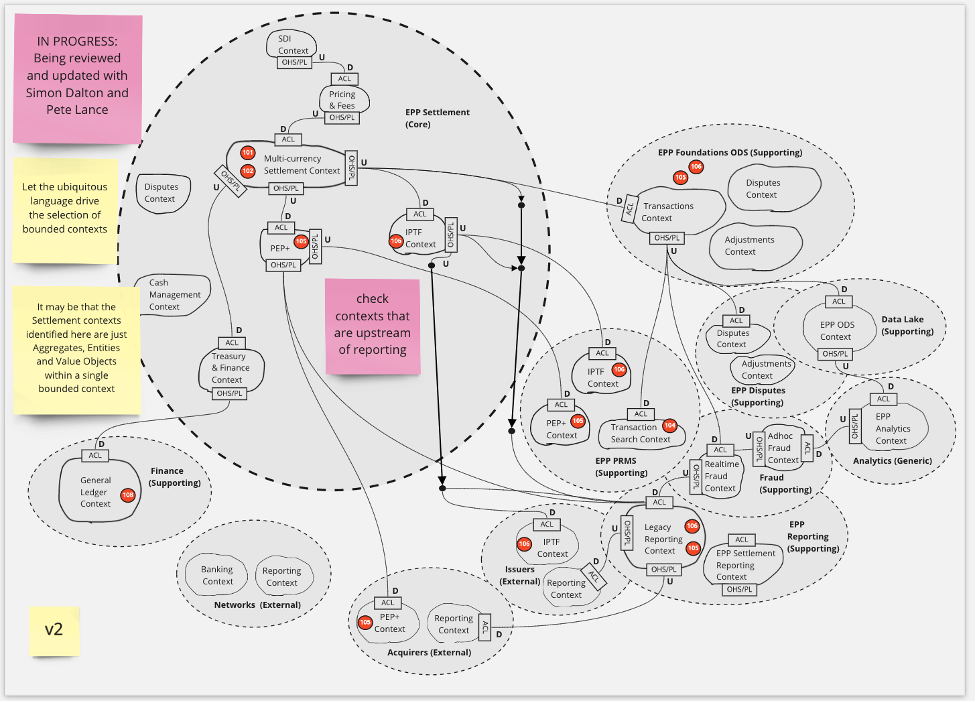
**EPP Settlement**



*Figure 5: : Context Map showing the flow of data from upstream into the Settlement subdomain*



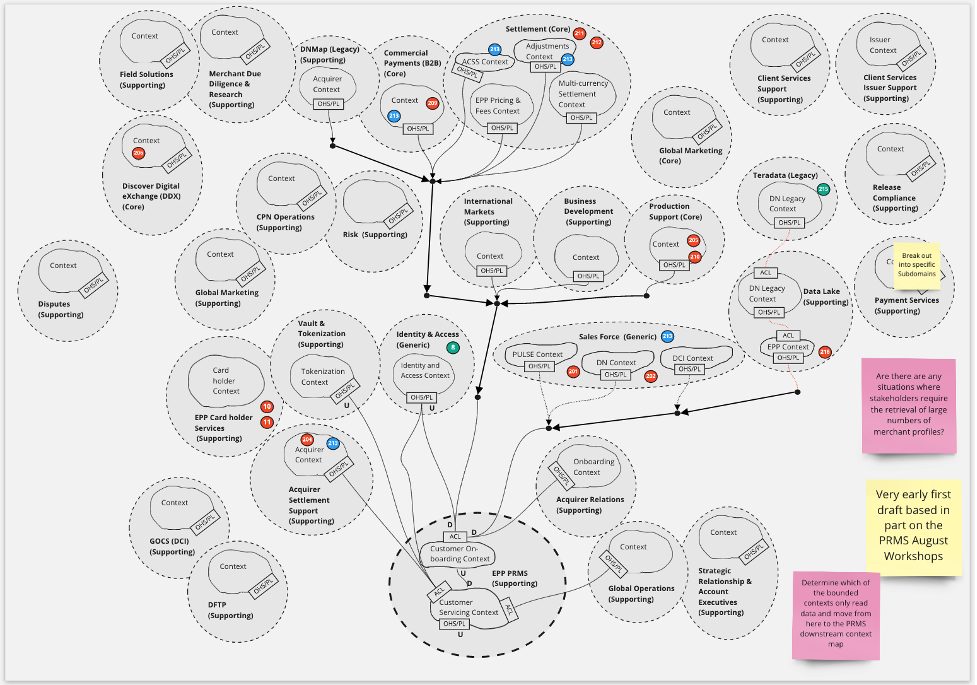
*Figure 6: A key to the Settlement-specific notes used to annotate the Context Maps in Figures 5 & 7*



*Figure 7: Context Map showing the flow of data from Settlement to downstream*

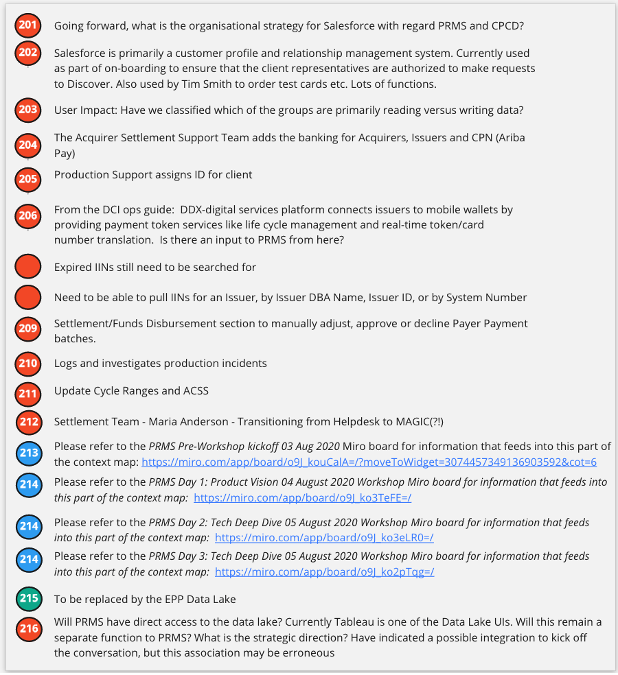
**EPP Party and Relationship Management System (PRMS)**

The EPP Party and Relationship Management System (PRMS) is focused on the onboarding and servicing of tenant customer data - for example issuer & acquirer profiles, banking information, netting group configuration, ISO reference data, issuer identification numbers and reference data to support e.g. dispute processing (there are many more examples). Once account executives have captured the requisite data and it has been approved, the data is released into the CPCD and used by components performing transaction processing and reporting, amongst others.



*Figure 8: Context Map showing the flow of data from upstream into the PRMS (Party Management & Servicing) EPP User Interface8*

8 First cut and incomplete (likely inaccurate in places) as per early 2020. Useful to identify gaps in shared understanding at the time



*Figure 9: A key to the PRMS-specific notes used to annotate the Context Map in Figure 8*

**Summary**

This article gave an overview of the Domain-Driven Design strategic tools and patterns and touched on how DDD, Agile and No-SQL complement each other. Organizational patterns, domains and bounded contexts were presented and their importance for ensuring the right software is built was explained. Context Maps were introduced and examples from the EPP CPCD space were shown which hopefully helps to put some flesh on the bones of the DDD theory. DDD fits naturally into product philosophy and an example was given where preconceived ideas about CPCD bounded contexts were corrected using DDD strategies. To become accomplished in DDD, one does not simply read the odd article and watch a Udemy video or two; some understanding may be gained but in the author's opinion it is normally very 2-dimensional and lacks the depth of understanding gained by thoroughly engaging with the material in the literature and practicing the craft. Recommended reading in order of priority is [2][3][4][5].

**References**

[1] Wikipedia, "Domain-driven design," [Online]. Available: <https://en.wikipedia.org/wiki/Domain-driven_design>. [Accessed 29 July 2020].

[2] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.

[3] V. Vernon, Implementing Domain-Driven Design, Addison Wesley, 2013.

[4] E. Evans, Domain-Driven Design Reference: Definitions and Pattern Summaries, Dog Ear Publishing LLC, 2014.

[5] V. Vernon, Domain-Driven Design Distilled Paperback, Addison-Wesley Professional, 2016.

[6] Hackolade, "Data Modelling is Dead ... Long Live Schema Design!," 3 June 2018. [Online]. Available: <https://hackolade.blogspot.com/2018/06/data-modeling-is-dead-long-live-schema.html>.

[7] Wikipedia, "Conway's Law," 28 07 2020. [Online]. Available: <https://en.wikipedia.org/wiki/Conway%27s_law>.

[8] Payment Services Data Design, EPP, BT, ABC LLC, "Payment Services Data Design," 28 July 2020. [Online]. Available: [https://github.ABCfinancial.com/pages/pymtsrv-org/PS-Data-Design/](https://github.discoverfinancial.com/pages/pymtsrv-org/PS-Data-Design/).