DDD Concepts and Patterns – Bounded Context

March 15, 2019 (http://blog.opus.ch/2019/03/ddd-concepts-and-patterns-bounded-context/) — Leave a comment (http://blog.opus.ch/2019/03/ddd-concepts-and-patterns-bounded-context/#respond)

DDD (http://blog.opus.ch/category/ddd/) Patterns (http://blog.opus.ch/category/patterns/)

The next parts of this series about domain driven design (DDD) will be about bounded contexts, how to keep their integrity and integrate them with each other. In this post, I would like to introduce the bounded context pattern.

When modeling a domain, it is important to define a common language for it. In DDD this is called the ubiquitous language. Every term in the ubiquitous language has a well defined, unique meaning within the boundaries of the (sub)-system under design. This boundary is called “bounded context”.

Why

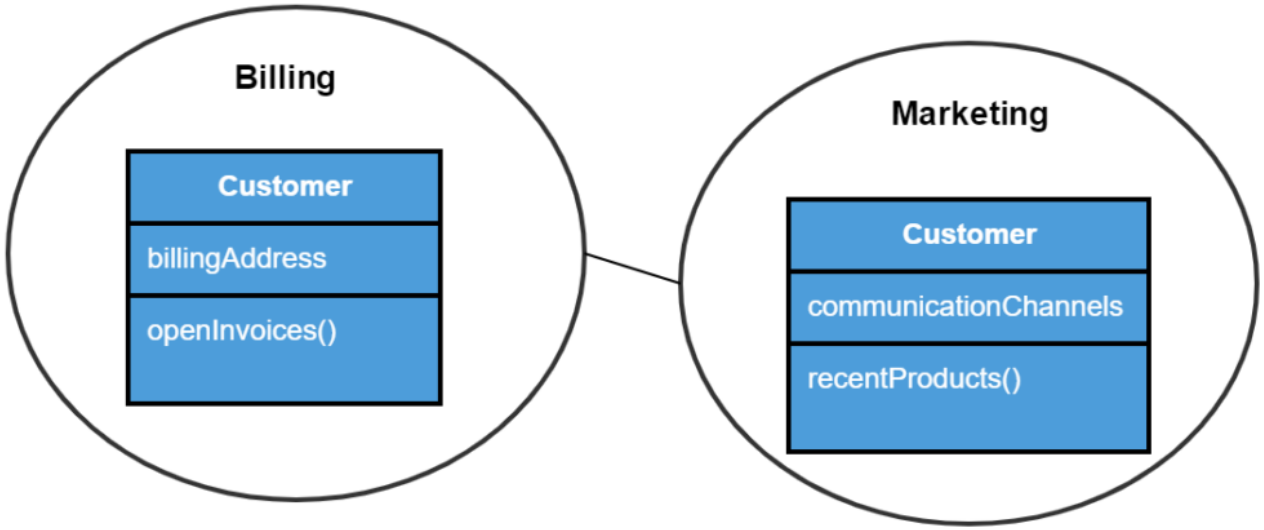
When we start designing our software from a database-centric view, we tend to build big database entities that contain all possible attributes. We use these entities in multiple features of the software. For example, we use the customer entity in the context of billing as well as marketing. It is easy to build such big entities, but they tend to be hard to use.

In DDD we design two different bounded contexts for billing and marketing. Customers are represented differently in each context, have different attributes and functionality. The advantage of this approach is simpler entities with fewer attributes and functionality which also leads to fewer concurrency issues.

A drawback is the need to synchronize contexts which may be complex to achieve. For example, the billing and marketing contexts would like to share customer information. There are multiple solutions for synchronization available both from DDD itself in the form of integration patterns as well as related techniques such as event sourcing and eventual consistency.

Example

The following diagram shows the customer entity in the bounded contexts of billing and marketing.



As you can see the customer entity needs very different attributes depending on the context. Marketing wouldn't care about open invoices probably. Moreover, billing needs only one communication channel, the billing address.

Organization

A single team should be assigned to develop a bounded context. When multiple teams work together on one bounded context, it is difficult to keep the ubiquitous language clean. Also, we can prevent very big bounded contexts

with this rule. When one team doesn't seem to be enough to build a bounded context, it is maybe too big, and we should split it into two specific ones.

On the other hand, one team may very well develop multiple bounded contexts. Many domain models have different meanings and language depending on the context. It can be useful to split the system into two contexts even if the same team maintains them.

Continuous Integration

Even when only a single team works on a bounded context, it is possible that different views of the model exist within team members. To address those and keep a unified understanding of the model and its ubiquitous language, DDD encourages the use of continuous integration.

The term "continuous integration" is used nowadays mainly in the context of DevOps, meaning to merge, build and test the code continuously. When using DDD, we also discuss the domain and ubiquitous language regularly. Frequent interaction between team members ensures that every code is written in sync with the model and no sub-contexts emerge with a separate language accidentally.

Wrap Up / Final Thoughts

Bounded contexts organize model elements on the level of domain boundaries. They help us understand that one concept can have very different meanings depending on the context. If we are aware of the contexts that exist in our software system, we can prevent misunderstandings and therefore build the right thing.

Implementing the same concept in different contexts comes at the cost of synchronization through. Therefore these contexts should have a reasonable size to prevent this overhead.

The next post will be about how to model and connect multiple bounded contexts. The model integration patterns that DDD provides offer many different possibilities.

Posted in **DDD (http://blog.opus.ch/category/ddd/)**, **Patterns (http://blog.opus.ch/category/patterns/)**. By **Michi Hausheer (http://blog.opus.ch/author/haeuslich/)**

G+ (HTTPS://PLUS.GOOGLE.COM/SHARE?

URL=HTTP%3A%2F%2FBLOG.OPUS.CH%2F2019%2F03%2FDDD-CONCEPTS-AND-

PATTERNS-BOUNDED-CONTEXT%2F)

(HTTPS://TWITTER.COM/INTENT/TWEET?

URL=HTTP%3A%2F%2FBLOG.OPUS.CH%2F2019%2F03%2FDDD-CONCEPTS-AND-PATTERNS-

BOUNDED-

CONTEXT%2F&TEXT=DDD+CONCEPTS+AND+PATTERNS+%E2%80%93+BOUNDED+CONTEXT)

(HTTP://PINTEREST.COM/PIN/CREATE/BUTTON/?

URL=HTTP%3A%2F%2FBLOG.OPUS.CH%2F2019%2F03%2FDDD-CONCEPTS-AND-

PATTERNS-BOUNDED-CONTEXT%2F&MEDIA=HTTP://BLOG.OPUS.CH/WP-

CONTENT/UPLOADS/2019/03/BLOGBOUNDEDCONTEXTFEATURE-

150X150.PNG&DESCRIPTION=DDD%20CONCEPTS%20AND%20PATTERNS%20–

%20BOUNDED%20CONTEXT)

in (HTTP://WWW.LINKEDIN.COM/SHAREARTICLE?

MINI=TRUE&URL=HTTP%3A%2F%2FBLOG.OPUS.CH%2F2019%2F03%2FDDD-CONCEPTS-

AND-PATTERNS-BOUNDED-

CONTEXT%2F&TITLE=DDD%20CONCEPTS%20AND%20PATTERNS%20–

%20BOUNDED%20CONTEXT)

(MAILTO:?SUBJECT=DDD CONCEPTS AND PATTERNS – BOUNDED

CONTEXT&BODY=HTTP://BLOG.OPUS.CH/2019/03/DDD-CONCEPTS-AND-PATTERNS-

BOUNDED-CONTEXT/)

WRITTEN BY
MICHI HAUSHEER (HTTP://BLOG.OPUS.CH/AUTHOR/HAEUSLICH/)

Software engineer with some Java experience. Please see the about section for more details.

WEBSITE (HTTP://WWW.OPUS.CH/)
LINKEDIN (HTTPS://WWW.LINKEDIN.COM/IN/MICHI-HAUSHEER-59846AA9/)

PREVIOUS ARTICLE

DDD Concepts and Patterns – Aggregate and Module (http://blog.opus.ch/2019/02/ddd-concepts-and-patterns-aggregate-and-module/)

NEXT ARTICLE

DDD Concepts and Patterns – Context Map (http://blog.opus.ch/2019/04/ddd-concepts-and-patterns-context-map/)

LEAVE A REPLY

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

POST COMMENT

RECENT POSTS

DDD Concepts and Patterns – Final (http://blog.opus.ch/2019/11/ddd-concepts-and-patterns-final/)
DDD Concepts and Patterns – Large-Scale Structure (http://blog.opus.ch/2019/09/ddd-concepts-and-patterns-large-scale-structure/)
DDD Concepts and Patterns – Distillation of the core domain (http://blog.opus.ch/2019/08/ddd-concepts-and-patterns-distillation-of-the-core-domain/)
DDD Concepts and Patterns – Supple Design 2 (http://blog.opus.ch/2019/07/ddd-concepts-and-patterns-supple-design-2/)
DDD Concepts and Patterns – Supple Design (http://blog.opus.ch/2019/05/ddd-concepts-and-patterns-supple-design/)
DDD Concepts and Patterns – Context Map (http://blog.opus.ch/2019/04/ddd-concepts-and-patterns-context-map/)
DDD Concepts and Patterns – Bounded Context (http://blog.opus.ch/2019/03/ddd-concepts-and-patterns-bounded-context/)
DDD Concepts and Patterns – Aggregate and Module (http://blog.opus.ch/2019/02/ddd-concepts-and-patterns-aggregate-and-module/)

Search … SEARCH

MORE FROM BLOG.OPUS.CH?

Email address:

Your email address

SIGN UP