Domänengetriebenes Design (DDD)
Domain-Driven Design (DDD) schlägt Techniken und Muster vor, um die intrinsische Komplexität in der Softwareentwicklung sowohl organisatorisch als auch technisch zu bewältigen. Wichtige DDD-Themen sind Business-Orientierung, Domänenmodellierung und Wiederverwendung von Wissen sowohl auf strategischer Ebene (d. H. Auf lange Sicht, projektübergreifend) als auch auf der taktischen Ebene.

Concept/Topic: *Strategic Domain-Driven Design (DDD)*

Context

Tactic DDD deals with implementing domain layer components; strategic DDD deals with integrating these components and managing complexity in end-to-end application landscapes for the long run. In small projects and businesses such long-time perspective might not be needed, but think about the many software development projects and development organizations at large software-intensive firms such as Amazon, ABB or Microsoft: the interfaces between system and teams have to be managed somehow, either centrally (this would be the task of enterprise architecture efforts) or decentrally (for instance, via Scrums of Scrums in the Scaled Agile Framework, SAFe). Strategic DDD provided patterns and a simple diagram type to support such efforts.

Definition(s)

Two Patterns for Model Partitioning: Subdomain (Functional), Bounded Context (Organizational, Technical)

The following definition and characteristics of *Bounded Context* (dt. "beschränkter Kontext", Gültigskeitsbereich) were given in the lecture:

- "A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable" (Evans (2003)).
- "A bounded context is an explicit boundary within which a domain model exists. Inside the boundary, all terms and phrases of the Ubiquitous Language have specific meaning, and the model reflects the language with exactness" (Vaughn (2013)).
- M. Fowler empshasizes that a bounded context is explicit about its interrelationships in his bliki article on Bounded Contexts[1].

According to an InfoQ article on strategic DDD[2] that develops an example in several steps, there are four particular reasons to bother about model partitioning and model/context boundaries:

- Same term, different meaning (homonym)
- Same concept, different use (polyseme)
- External system differences (heterogeneity)
- Scaling up the organization (multiple teams)

The pattern synopsis is shown in the Figure "Subdomain and Bounded Context patterns".

Subdomain partitioning is a top-down approach; Bounded Context partitioning a bottom up one. More differences between these concepts are listed in the Figure "Subdomain vs. Bounded Context Comparison".

Context Map and Relationship Types

Context Mapping is the technique in strategic DDD to identify and connect bounded contexts. The map metaphor expresses that details are left out to allow a quick orientation in the team/system landscape.[3] The Figure "Context Map pattern" shows the pattern synopsis.

[1] http://martinfowler.com/bliki/BoundedContext.html
[2] https://www.infoq.com/articles/ddd-contextmapping
[3] S. Brown uses the map metaphor as well when motivating the need for models and architecture en route to his C4 model covered previously.

Context

A large and complex set of requirements has to be dealt with. Parts of these have already been implemented, possibly multiple times. Both analysis-level domain model and design-level domain models are large and complex as well, and hard to agree upon universally/globally (for instance, organization-widely).

Problem

How can a large set of candidate aggregates (comprising entities and value objects and services) be grouped so that problem and solution space remain manageable and semantic ambiguities are isolated (or at least identified)?

umfassen

unklarheiten

Solution

Organize the problem space (i.e., analysis-level domain model) into multiple *subdomains*; distinguish the *core domain* from *supporting* and *generic subdomains*. Separate the solution space (i.e., design-level domain model) into multiple *bounded contexts* and apply tactic DDD to each context-specific model.

trennen

Figure 1: *Subdomain and Bounded Context patterns*

Subomains vs Bounded Context

	Subdomain	Bounded Context (BC)
Purpose	Partition problem space, focus and prioritize work	Partition solution space, identify communication and integration needs
Stakeholder concern	Complexity and size (of requirements); special skills needed for certain design tasks	Complexity and size (of realization); deal with terminology conflicts (homonyms, polysemes)
Viewpoint	Scenario, Logical	Implementation, Process
Types/Variants	Core Domain Supporting Subdomain Generic Subdomain	Team BC, subsystem BC
Units of decomposition	Groups of features (e.g. epic in agile, use case model/category in UML), classes in OOA domain model	Subdomains from OOA, one single OOD domain model
Relationship to Tactic DDD	Identifies need for Entities, Value Objects, Factories, Repositories; suggests initial set of Aggregates	Identifies need for implementation of Aggregates, groups them (and their content), leads to refactoring of Aggregate landscape
Mutual relationship	1:1 (in ideal world), n:m (in practice)	

vorschlagen

Figure 2: *Subdomain vs. Bounded Context Comparison*

Context

A rich, complex domain model has been decomposed into multiple analysis-level subdomains (aufgeteilt) to be realized/organized in several design-level bounded contexts.

Problem

How can end-to-end functionality be implemented in a controlled manner (Art und Weise) when dealing with a partitioned domain model that is organized into functional subdomains and bounded contexts that represent (sub-)systems or teams?

Solution

Map subdomains to one or more bounded contexts. Make the permitted organizational relationships between bounded contexts explicit and visualize them in a *context map*. Assign one or more relationship types to these relations that regulate the visibility of the related parties, their right to influence each other (coupling), as well as the (a)symmetry of the relation and remote connectivity.

Figure 3: *Context Map pattern*

An example is presented in the InfoQ article[4] already cited above:

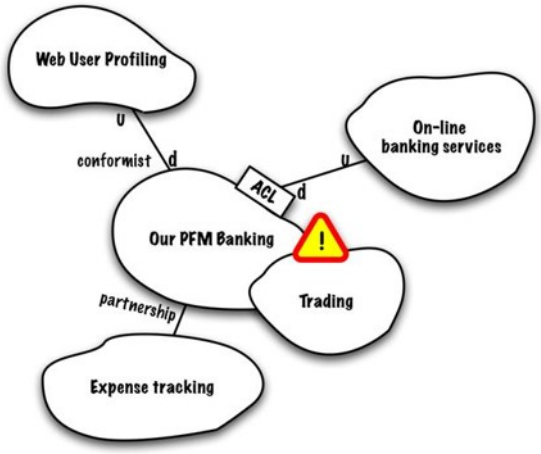

Figure 4: *Sample Context Map*

The original DDD book (Evans (2003)) defined an initial set of relations between contexts appearing in a map, e.g., the *Conformist* pattern. Later on, a few additional types were added, for instance *Partnership* and *Big Ball of Mud*. Summaries of the patterns from the original DDD book as well as a few extensions are available for free download in the book "Domain-Driven Design Reference"[5] also by E. Evans:

- *Published Language (PL)*: "The interacting bounded contexts agree on a common a language (for example a bunch of XML schemas over an enterprise service bus) by which they can interact with each other."
- *Shared Kernel*: "Two bounded contexts use a common kernel of code (for example a library) as a common

[4] https://www.infoq.com/articles/ddd-contextmapping
[5] http://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf

lingua-franca, but otherwise do their other stuff in their own specific way."

- *Open Host Service (OHS)*: "A Bounded Context specifies a protocol by which any other bounded context can use its services (e.g. a RESTful HTTP service or a SOAP Web service). This protocol exposes the Published Language."
- *Customer/Supplier* (a.k.a. Customer/Supplier Teams/Development): "One bounded context uses the services of another and is a stakeholder (customer) of that other bounded context. As such it can influence the services provided by that bounded context."
- *Conformist*: "One bounded context uses the services of another but is not a stakeholder to that other bounded context. As such it uses"as-is" (conforms to) the protocols or APIs provided by that bounded context."
- *Anti-Corruption Layer (ACL)*: "One bounded context uses the services of another and is not a stakeholder, but aims to minimize impact from changes in the bounded context it depends on by introducing a set of adapters – an anti-corruption layer."
- An additional one (not featured in the lecture because it merely describes "no relationship" and therefore cannot appear in a context map) is *Separate Ways*.

These context relationships differ regarding the following design concerns:

- Topology: Local vs. remote?
- Visibility (of collaboration/communication partners to each other)?
- (A)symmetry of relationship?
- Amount of control and influence for client/consumer?

When drawing a context map, you basically have to answers these questions to end up at the right pattern for any given relationship.

The relationship types are organizational patterns (not technical ones); they can drive the architectural decision making in API design. Follow-on decisions pertain the integration style technology and style (covered in VSS and the weeks to come). The relationship types do not exclude, but complement each other by default.

Critique and Pattern Combinations

The pattern naming is somewhat inconsistent: a name such as "Conformist" refers to the downstream end of a relation, whereas "Customer/Supplier" includes both ends.

The explanation of OHS is rather dense and it is not very clear what "open" means in this setting; in one interpretation (the one followed in the IDDD book Vaughn (2013)), open means that the upstream provider might not know is downstream consumers, and might not be willing to let individual known ones influence the common interfaces that is used by multiple consumers (both known and unknown ones).

A Published Language is not only exposed by OHSs (otherwise, there would not be a need for two patterns in the language).

The ACL pattern is the DDD version of Enterprise Application Integration (EAI) middleware and the Enterprise Integration Pattern (EIP) category Message Transformation (Hohpe and Woolf (2003)); such pattern appears in several pattern languages. The DDD books could make this connection more explicit. The responsibilities and collaborations of an ACL are specified in the Figure "CRC Card for ACL Pattern". An ACL is commonly applied on the downstream, conformist side of an asymmetric remote collaboration/communication relationship (with OHS and Published Language appearing on the other side, the upstream side).

The figure "Context Mapping Best Practices" explains some more pattern relationships:

***Component:* Anti-Corruption Layer (ACL)**	
Responsibilities: Verantwortung • Acts as an intermediary between client and server • Protects downstream client from platform- and model-specifics with model transformations • Makes upstream client independent of change management policy of upstream provider (versioning etc.) • Maps external transfer syntax (JSON, XML) to object notation; maps from model to model when crossing context boundary (including error message mapping) • Cache requests and responses (optional), support conditional requests • Observe rate limits and other quality-related constraints	*Collaborators (Interfaces to/from):* • Downstream client • Upstream data/service provider
Candidate implementations technologies (and known uses): • Many EAI products and open source assets such as Mule ESB, Talend, Pentaho DI • Java Connector Architecture (in previous editions of JEE) • AdCubum SYRIUS; Core Banking SOA (from lesson 1) and most enterprise applications vary this pattern (typically under different names: Gateway, Adapter, Façade, Wrapper)	

Figure 5: *CRC Card for ACL Pattern*

Context

The context map of an integrated system of systems (or a single complex system decomposed into services) shows relationships of many types: published language, shared kernel, open host service, customer/supplier, conformist, ACL.

Problem

How can upstream and downstream contexts be protected from each other (to support loose coupling), but still benefit from each other/invoke each other?

Solution

Keep the published language minimal, especially when defining an OHS. Avoid shared kernels between supporting subdomains and in customer-supplier relations. Add an ACL on the conformist side of an OHS-conformist relation. Consider introduction of ACLs also for customer-supplier relations.

Figure 6: *Context Mapping Best Practices*

Notation

DDD does not mandate any particular syntax such as UML or pattern icons or pictograms; hence, you can also visualize the shared kernel differently (as some of the samples in the lecture show). The point here is the pattern selection and visualization (not notation correctness): identification of the mentioned bounded contexts with their relations and selection of suited types of these relations.

Summary

The figure "Strategic DDD Patterns and their Relationships" summarizes our coverage of strategic DDD in this lecture (in an UML class diagram serving as DDD metamodel):

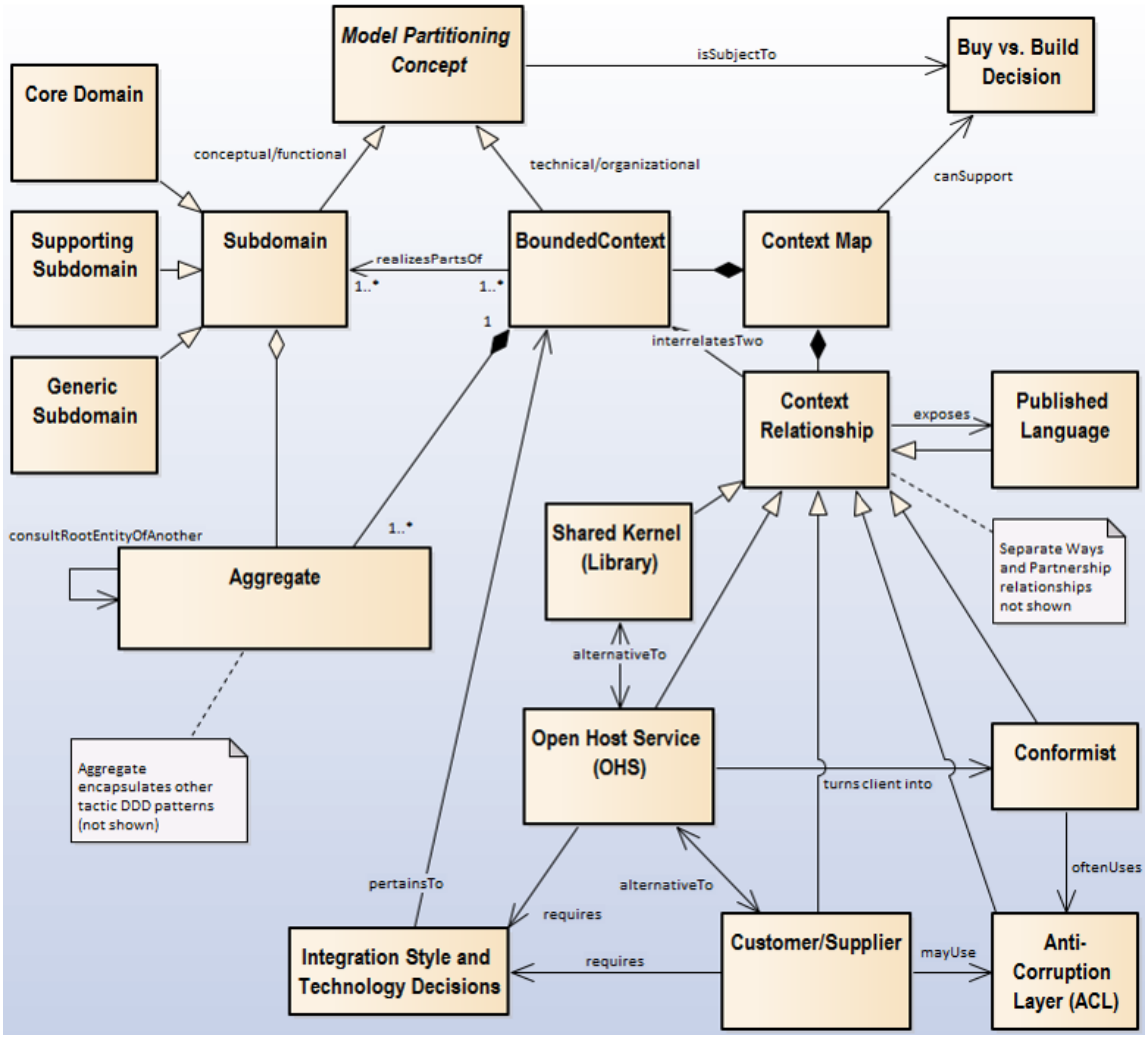

Figure 7: *Strategic DDD Patterns and their Relationships*

Application of the Concept in Products and Projects

Usage of Concept/Topic

There is an active DDD user community[6] gathering architects, developers, and microservices integration specialists; the topic is frequently represented at practitioner conferences such as QCon, OOP, and SATURN.

[6] http://dddcommunity.org/

Regrettably, the canonical Cargo Tracking DDD Sample Application[7] only features tactic DDD, not strategic DDD (at least not explicitly).

There are three main use cases/usage scenarios for DDD:

- DDD in Commercially-Off-The-Shelf (COTS) software evaluation and in support of buy-vs.-build decisions
- "OOAD/OOP done right"
- Implementing SOA and designing (micro-)services

DDD in COTS Evaluation and in Support of Buy-vs.-Build Decisions. As discussed in the lecture slides, step 3 of exercise 8 and the corresponding Repetition Questions (RQs): DDD can help to focus the evaluation effort on the Core Domain, and getting information about the functional coverage and the domain model of the candidates. See this article (from the same Statoil architects): "Using domain-driven design to evaluate commercial off-the-shelf software"[8] for further details.

OOAD/OOP Done Right. See fact sheet o Tactical DDD.

Implementing SOA, Designing (Micro-)Services. This topic will be covered to some extend in the weeks to come. Vaughn (2013) covers it to some extend, but does not go into detail.

General Observations and Discussion on DDD

Many of the DDD patterns qualify as organizational patterns rather than technical ones, which makes them powerful but also harder to grasp. An Example is Ubiquitous Language; counter examples are Entity and Value Object (from tactic DDD, see separate fact sheet). This is a key difference to pattern languages such as GoF Design Patterns, POSA, PoEAA, and EIP (which makes DDD complementary to these languages).

You might ask yourself whether some of the DDD patterns a bit shallow (or even trivial). Possibly an for experienced practitioner, but they still serve well as common vocabulary and design philosophy; thy become powerful when applied in combination.

DDD overlaps with OOAD methods and other patterns books to some extend, but that is ok as long one is aware of it (and any semantic differences).

The original "light blue DDD book" (Evans (2003)) might come across as a bit too abstract and vague, for instance due to the usage of rather short and generic pattern names such as Aggregate. This is actually not the case, at least not everywhere in the book (e.g., Repository pattern); multiple reading passes are required (note that the book uses the "Alexandrian" pattern style, unlike GoF and PoEAA). Furthermore, second-generation books such as IDDD and experience reports such as those used in the exercise can fill any gaps (e.g., Aggregate pattern implementation, Subdomain vs. Bounded Context).

More Information

- "Implementing Domain-Driven Design" by V. Vaughn[9]; the sample application featured in the book is available on GitHub
- "Patterns, Principles, and Practices of Domain-Driven Design" by S. Millett[10] has .NET source code

[7] https://github.com/citerus/dddsample-core
[8] http://dddcommunity.org/wp-content/uploads/files/practitioner_reports/landre_einar_2006_part2.pdf
[9] https://vaughnvernon.co/?page_id=168
[10] http://www.wrox.com/WileyCDA/WroxTitle/productCd-1118714709.html

- DDD when modernizing legacy systems: Bubble Context article by E. Evans[11]
- Subdomains (vs. Aggregates vs. Bounded Contexts), Ubiquitous Language vs. Published Language are featured in this article[12]

Related Topics and Concepts

Component Modeling and C4; Buy vs. Build, Tactic DDD, SOA and Microservices

Miscellaneous Links:

- An IFS website on DDD[13] that has additional pointers.

References

Evans, Eric. 2003. *Domain-Driven Design: Tacking Complexity in the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Hohpe, Gregor, and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Vaughn, Vernon. 2013. *Implementing Domain-Driven Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[11] http://domainlanguage.com/ddd/strategy/GettingStartedWithDDDWhenSurroundedByLegacySystemsV1.pdf
[12] http://gorodinski.com/blog/2013/04/29/sub-domains-and-bounded-contexts-in-domain-driven-design-ddd/
[13] https://www.ifs.hsr.ch/index.php?id=15666&L=4