Domain-Driven Design and Spring

Oliver Gierke – 2020-10-25 12:48:32 +0100

Back to lectures

1. Introduction
2. Domain-Driven Design
    2.1. Building blocks
        2.1.1. Bounded context
        2.1.2. Value objects
        2.1.3. Entities
        2.1.4. Aggregate roots
        2.1.5. Repositories
        2.1.6. Domain services
    2.2. Domain-Driven Design in a Spring application
3. Further reading

Appendix
    Appendix A: Bibliography
    Appendix B: License

1. Introduction

This script is supposed to give a brief overview about the fundamental building blocks and concepts of Domain-Driven Design focussing on their application to Java (and mostly Spring) based web applications as well as how these building blocks translate into different groups of classes with different traits.

The second part focuses on the concepts of entities, aggregate roots and repositories and how they map to Spring based Java applications.

2. Domain-Driven Design

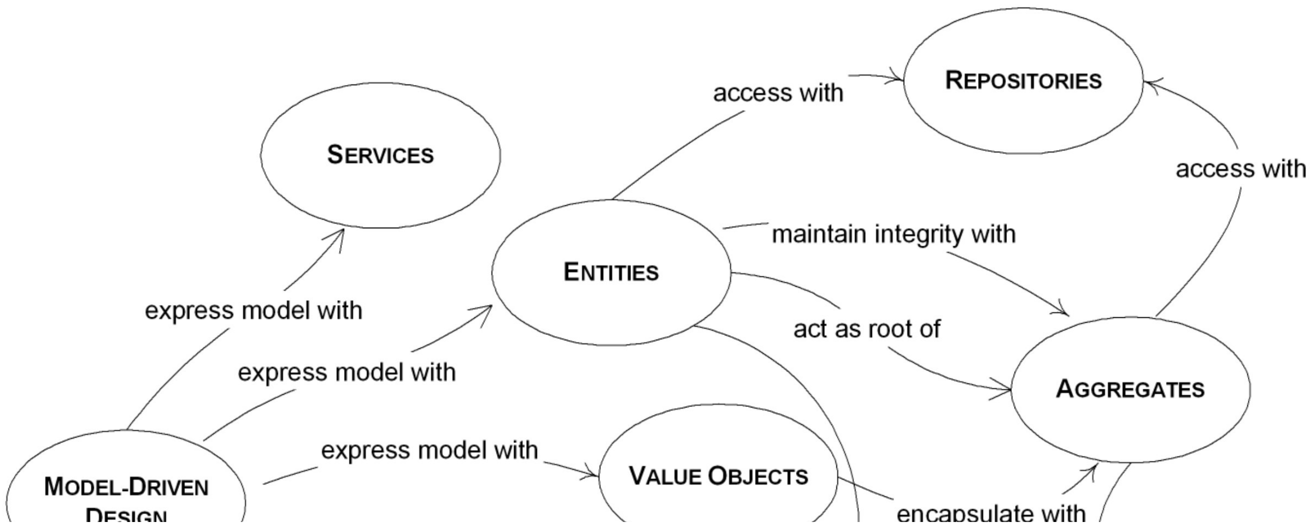
Domain-Driven Design is a book by Eric Evans and is undoubtedly one of the most important books on software design. A more compact version of the book is available as Domain-Driven Design Quickly on InfoQ.

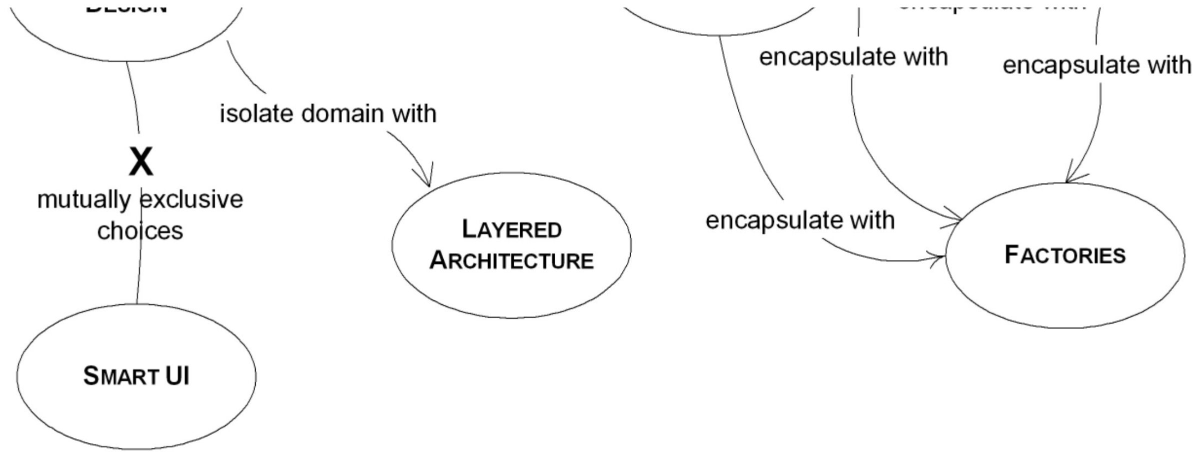
The book's core focus is on the domain a piece of software is supposed to deal with and how that domain actually connects to the software: starting with an emphasis on the domain's ubiquitous language, approaches to distilling a model from that language and eventually turning this into working software reflecting that model.

While software design patterns[1] are usually defined through a technical context and are used to describe and capture relationships of classes, DDD is a more wholistic approach to find a common language between programmers and business and ultimately connect the code to be written to the domain. In other words, while you might be familiar with more technical patterns already, in business code they're mostly perceived as technical noise[2]. Also, the original authors of the GoF book have shared incisive insights[3] about what they'd do differently if the book ever got an update, including which patterns to deemphasize, how to slightly change the organization of the patterns etc.

2.1. Building blocks

The building blocks define common terms that are used to describe model elements and assign certain traits to them. Let's have a look at the most fundamental ones here.





2.1.1. Bounded context

When modeling a domain any attempt to model it as a whole is doomed to fail as the sheer number of stakeholders and their different views on the domain might be very different and trying to craft a single, unique model to satisfy all of the needs might be either completely impossible or the source of great complexity in the best case. Let’s have a look at a sample sketch that outlines identified concepts in a a Point of Sales (POS) domain.

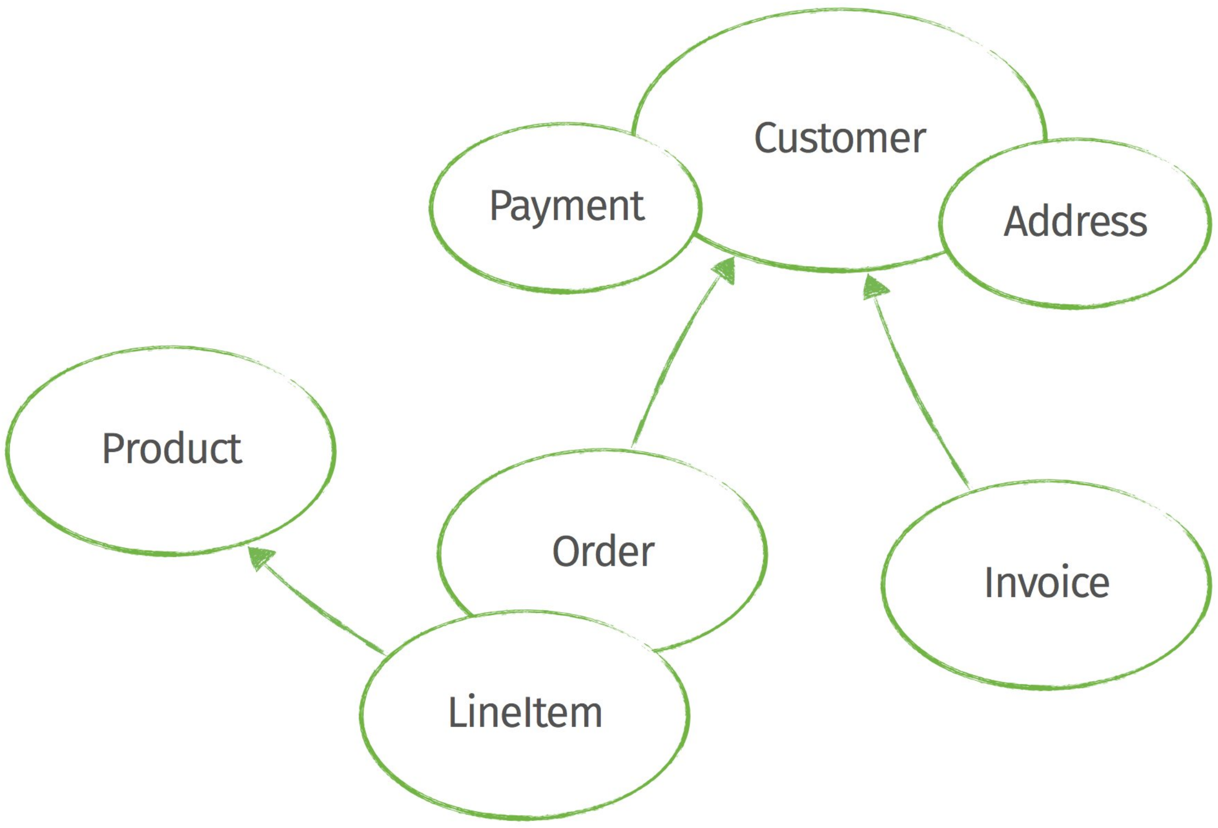


Figure 1. Model elements in a point of sales application

As you can see we already grouped the model elements slightly to differentiate from more remote ones. Looking at the core concepts of customer and order here we can still identify different contexts in which the model elements just discovered might play a role in.

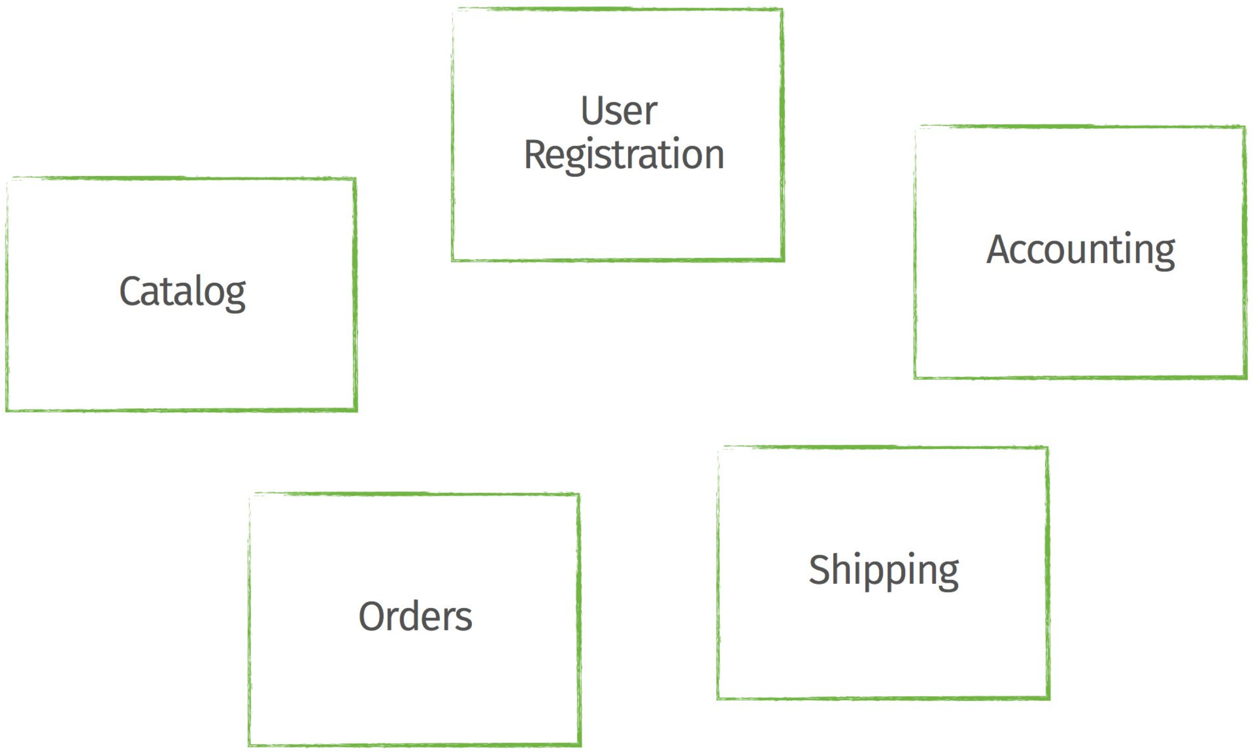


Figure 2. Identified bounded contexts

Here we identified core high-level parts of the system that might all deal with the concept of a customer or an order but are usually interested in different aspects of them. The accounting context is usually interested in billing information of a customer and different payment options while the shipping context's sole interest is usually shipping addresses and tracking an order. The order context might know about a product through its line items but actually only refers to what is essentially maintained by the catalog.

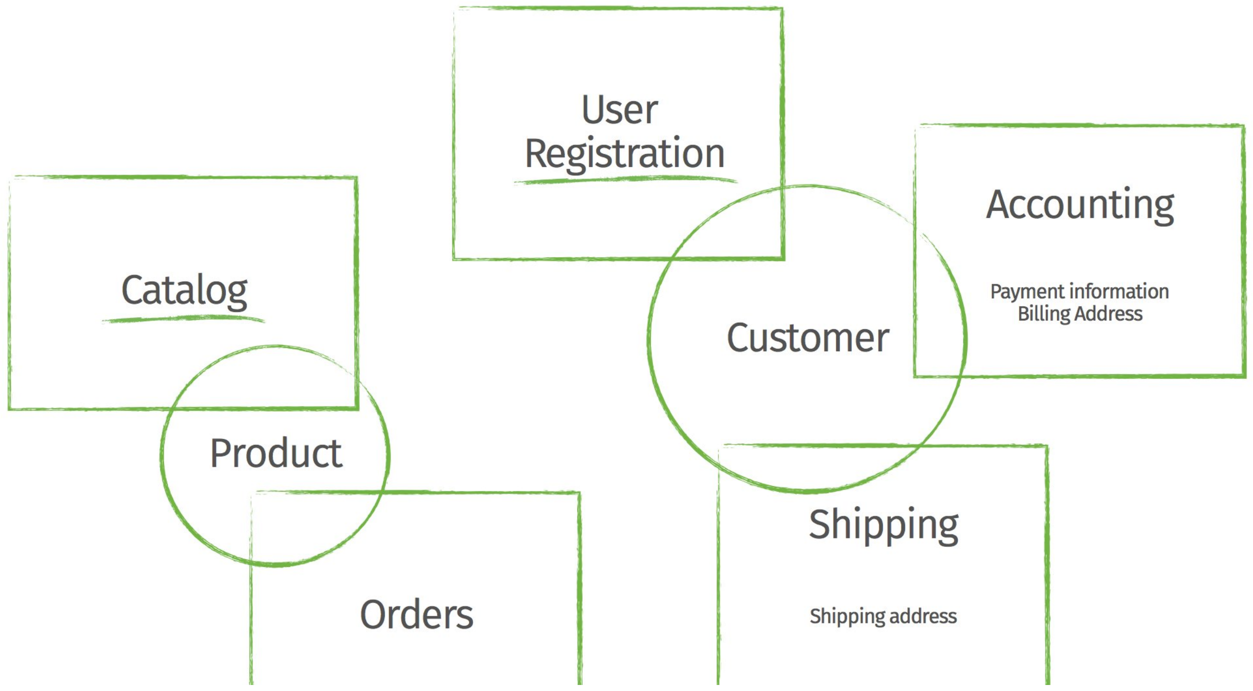


Figure 3. Model elements in bounded contexts

This boils down to model elements which might appear as a singular concept at first glance needing to be reflected in different way in different aspects of the system. Modern software architecture even sometimes takes this a step further and even aligns system boundaries with such a bounded context, actively accepts redundancy and eventual consistency to reduce the complexity of the individual systems, fosters resilience and increases development productivity.[4].

2.1.2. Value objects

Represent a concept from the domain that — as the name suggests — is considered a value. In terms of classes that means that individual instances don't have identity, no life cycle. A value object needs to be immutable to ensure integrity of the instances as they can be shared amongst different consumers. In the Java language, an instance of `String` is a good example of these traits, although it's not a good example for a domain concept as it's rather generic. A credit card number, an email address are great candidates to be modeled as value objects. Value objects are usually part of other model elements like entities or services. Implementing model elements as value objects also has a great impact on legibility and comprehensibility of the code base as Dan Bergh Johnsson demonstrates in his talk Power Use of Value Objects in DDD.

Value objects are often undervalued when it comes to the translation of model into code as the implementation of a value object in plain Java is quite cumbersome due to the need to implement accessors, `equals(…)` and `hashCode()`. This can be countered with a tiny annotation processor called Lombok[5].

2.1.3. Entities

In comparison to value objects, an entity's core trait is it's identity. Two customers named Michael Müller might not constitue the very same instance, so that usually a dedicated property is introduced to capture the identity. Another core trait of entities is that they're usually subject to a certain lifecycle within the problem domain. They get created, they undergo certain state changes usually driven by domain events and might reach an end state (i.e. might be deleted, although this doesn't necessary mean that the information is removed from the system). Entities usually relate to other entities and contain properties that are value objects or primitives (the former preferred).

Entities VS. Value objects

It’s not always clear whether to model a domain concept as value object or entity. In fact the concept of an address can — depending on the context — even be modeled as both within the same application. While the address of a store might be a value object might just be part of the domain concept store, it might as well be an entity in the context of modeling a customer as shipping and billing addresses might have to be created, edited, deleted etc.

2.1.4. Aggregate roots

Within the set of entities of a system, some usually play a special role in their relationship to others. Consider an order consisting of line items. The order might expose a total which is calculated from the prices of the individual line items. It might only be in a valid state if it is more than a certain minimum total. The root entities of such a construct are usually conceptually elevated to a so called aggregate root and thus create certain implications:

- The aggregate root is responsible to assert invariants on the entire aggregate. State changes on the aggregate always result in a valid result state or trigger an exception.
- To fulfil this responsibility, only aggregate roots can be accessed by repositories (see Repositories). State changes involving non-root entities need to be applied by obtaining the aggregate root and triggering it on the root.
- As an aggregate forms a natural consistency boundaries, references to other aggregates should be implemented in a by-id way.

2.1.5. Repositories

Conceptually a repository simulates a collection of aggregate roots and allows accessing subsets or individual items. They’re usually backed by some kind of persistence mechanism but shouldn’t expose it to client code. Repositories refer to entities, not the other way round.

2.1.6. Domain services

Domain services implement functionality that cannot uniquely be assigned to an entity or value object or need to orchestrate logic between them and repositories. Business logic should be implemented in entities and value objects as much as possible as it can be tested more easily within them.

2.2. Domain-Driven Design in a Spring application

The mapping of a domain concept to a DDD concept has quite a few important implications for the way these concepts are reflected in the code. To work effectively with Spring based Java applications, it's important to distinguish between that category of newables and injectables.

As the names suggest, the differentiating line is drawn between the ways a developer gets hold of an instance of a class for that particular model element. A newable can be simply instantiated using the `new` operator, although even that should be limited to as few places as possible. The factory pattern can help here, too. Entities and value objects are newables. An injectable is usually a Spring component, which means that the latter controls its lifecycle, creates instances and destroys them. This allows the container to equip the service instance with technical services like transactions or security. Clients obtain instances by using dependency injection (hence the name injectable). Repositories and services are injectables.

This distinction between these two groups of classes naturally defines a preferred dependency direction from injectables to newables. Generally speaking

- *Value object* - JPA `@Embeddable` + corresponding `equals(…)` and `hashCode()` (Lombok's `@Value` helps here). Can depend on other value objects and entities.
- *Entity* - JPA `@Entity` + corresponding `equals(…)` and `hashCode()` implementations. Can depend on other entities and value objects.
- *Repository* - Spring component, usually a Spring Data repository interface. Can depend on entities and value objects, are centered around entities that are aggregate roots.
- *Domain services* - Usually a Spring component, a class annotated with `@Component` or a stereotype annotation. Can also be modeled as newables in some cases in case they don't require technical services to be applied (e.g. security, transactions).

Not all classes of a Spring application can be assigned to these DDD categories. These other classes can usually be grouped into the following, more technical ones:

- *Application configuration* - Classes to configure components.
- *Technical adapters* - Business logic implemented in a Spring application is usually exposed to clients through some remoting technology. In a web application these technologies are HTTP, HTML and JavaScript. With Spring MVC, controller classes serve the purpose of translating the concepts of the remoting technology (e.g. the notion of a request, request parameters, a payload, a response etc.) into the domain concepts and invoke services with value objects and entities.

3. Further reading

1. Check out the Guestbook[6] and Videoshop[7] and make sure you understand which classes implement which DDD concepts.
2. Make sure you understand how the different traits of these concepts (identity for entities, immutability for value objects, dependency injection for services) are implemented.

Appendix

Appendix A: Bibliography

- [ddd-book] - Eric Evans — Domain-Driven Design: Tackling Complexity in the Heart of Software- Addison Wesley. 2003.
- [ddd-quickly] - Abel Avram, Floyd Marinescu — Domain-Driven Design Quickly. InfoQ. 2006.
- [power-of-value-objects] - Dan Bergh Johnsson — Power Use of Value Objects in DDD. InfoQ. 2009.

Appendix B: License

---

1. Software Design Patterns - Wikipedia
2. Patterns as technical noise - Slide deck, slide 54
3. Design Patterns 15 Years Later: An Interview with Erich Gamma, Richard Helm, and Ralph Johnson, Interview
4. Microservices - Wikipedia
5. Project Lombok - Project website
6. Guestbook - Sample application on GitHub
7. Videoshop - Sample application on GitHub

Last updated 2020-10-25 12:48:32 +0100