airbrake.io

Domain-Driven Design: What is it and how do you use it?

8-11 minutes

Expanding on our previous article that covered Object-Oriented Analysis and Design (OOAD), today's article will explore `domain-driven design` (DDD). DDD is a software development approach that uses and builds upon OOADprinciples and ideas, so it's the next logical topic for us to dive into.

Throughout this article we'll examine what `domain-driven design` is, how it is commonly implemented in modern development life cycles, and consider a few potential advantages and disadvantages of using DDD in your own projects. Let's get crackin'!

To define `domain-driven design` we should first establish what we mean by `domain` in this context (and in development in general). The common dictionary definition of `domain` is: "A sphere of knowledge or activity." Drilling down a bit from that, `domain` in the realm of software engineering commonly refers to the subject area on which the application is intended to apply. In other words, during application development, the `domain` is the "sphere of knowledge and activity around which the application logic revolves."

Another common term used during software development is the `domain layer` or `domain logic`, which may be better known to many developers as the `business logic`. The `business logic` of an application refers to the higher-level rules for how `business objects` (see: [OOAD](#)) interact with one another to create and modify modelled data.

What is Domain-Driven Design?

Initially introduced and made popular by programmer Eric Evans in his 2004 book, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, `domain-driven design` is the expansion upon and application of the `domain` concept, as it applies to the development of software. It aims to ease the creation of complex applications by connecting the related pieces of the software into an ever-evolving model. DDD focuses on three core principles:

- Focus on the core `domain` and `domain logic`.

- Base complex designs on models of the `domain`.

- Constantly collaborate with `domain experts`, in order to improve the application model and resolve any emerging `domain`-related issues.

  Evans' *Domain-Driven Design* further defines a few common terms that are useful when describing and discussing DDD practices:

- **Context**: The setting in which a word or statement appears that determines its meaning. Statements about a model can only be understood in a context.

- **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.

- **Bounded Context**: A description of a boundary (typically a subsystem, or the work of a specific team) within which a particular model is defined and applicable.

Building Blocks

`Domain-driven design` also defines a number of high-level concepts that can be used in conjunction with one another to create and modify `domain models`:

- **Entity**: An `object` that is identified by its consistent thread of continuity, as opposed to traditional `objects`, which are defined by their `attributes`.

- **Value Object**: An `immutable` (unchangeable) `object` that has `attributes`, but no distinct identity.

- **Domain Event**: An object that is used to record a discrete event related to model activity within the system. While *all* events within the system could be tracked, a `domain event` is only created for event types which the `domain experts` care about.

- **Aggregate**: A cluster of `entities` and `value objects` with defined boundaries around the group. Rather than allowing every single `entity` or `value object` to perform all actions on its own, the collective `aggregate` of items is assigned a singular `aggregate root` item. Now, external objects no longer have direct access to every individual `entity` or `value object` within

the `aggregate`, but instead only have access to the single `aggregate root` item, and use that to pass along instructions to the group as a whole. This practice correlates with many of the actual coding practices we're covering in our <u>design patterns</u> series.

- **Service**: Essentially, a `service` is an operation or form of business logic that doesn't naturally fit within the realm of `objects`. In other words, if some functionality must exist, but it cannot be related to an `entity` or `value object`, it's probably a `service`.

- **Repositories**: Not be confused with common `version control repositories`, the DDD meaning of a `repository` is a `service` that uses a global interface to provide access to all `entities` and `value objects` that are within a particular `aggregate` collection. Methods should be defined to allow for creation, modification, and deletion of objects within the `aggregate`. However, by using this `repository service` to make data queries, the goal is to remove such data query capabilities from within the business logic of `object models`.

- **Factories**: As we've discussed through a number of <u>design patterns</u> articles already, DDD suggests the use of a `factory`, which encapsulates the logic of creating complex objects and `aggregates`, ensuring that the client has no knowledge of the inner-workings of object manipulation.

`Domain-driven design` also heavily emphasizes the ever-more-popular practice of `continuous integration`, which asks the entire development team to use one shared code repository and push commits to it daily (if not multiple times a day).

An automatic process executes at the end of the work day, which checks the integrity of the entire code base, running automated unit tests, regression tests, and the like, to quickly detect any potential problems that may have been introduced in the latest commits.

Advantages of Domain-Driven Design

- **Eases Communication**: With an early emphasis on establishing a common and `ubiquitous` language related to the `domain model` of the project, teams will often find communication throughout the entire development life cycle to be much easier. Typically, DDD will require less technical jargon when discussing aspects of the application, since the `ubiquitous language` established early on will likely define simpler terms to refer to those more technical aspects.

- **Improves Flexibility**: Since DDD is so heavily based around the concepts of `object-oriented analysis and design`, nearly everything within the `domain model` will be based on an object and will, therefore, be quite modular and encapsulated. This allows for various components, or even the entire system as a whole, to be altered and improved on a regular, continuous basis.

- **Emphasizes Domain Over Interface**: Since DDD is the practice of building around the concepts of `domain` and what the `domain experts` within the project advise, DDD will often produce applications that are accurately suited for and representative of the `domain` at hand, as opposed to those applications which emphasize the UI/UX first and foremost. While an obvious balance is required, the focus on `domain` means that a DDD approach can

produce a product that resonates well with the audience associated with that `domain`.

Disadvantages of Domain-Driven Design

- **Requires Robust Domain Expertise**: Even with the most technically proficient minds working on development, it's all for naught if there isn't at least one `domain expert` on the team that knows the exact ins and outs of the subject area on which the application is intended to apply. In some cases, `domain-driven design` may require the integration of one or more outside team members who can act as `domain experts` throughout the development life cycle.

- **Encourages Iterative Practices**: While many would consider this an advantage, it cannot be denied that DDDpractices strongly rely on constant iteration and continuous integration in order to build a malleable project that can adjust itself when necessary. Some organizations may have trouble with these practices, particularly if their past experience is largely tied to less-flexible development models, such as the <u>waterfall model</u> or the like.

- **Ill-Suited for Highly Technical Projects**: While DDD is great for applications where there is a great deal of `domain` complexity (where `business logic` is rather complex and convoluted), DDD is not very well-suited for applications that have marginal `domain` complexity, but conversely have a great deal of technical complexity. Since DDD so heavily emphasizes the need for (and importance of) `domain experts` to generate the proper `ubiquitous language` and then `domain model` on which the project is based, a project that is incredibly technically complex

may be challenging for `domain experts` to grasp, causing
problems down the line, perhaps when technical requirements or
limitations were not fully understood by all members of the team.