Domain-driven design

Domain-driven design (**DDD**) is an approach to software development for complex needs by connecting the implementation to an evolving model.[1] The premise of domain-driven design is the following:

- placing the project's primary focus on the core domain and domain 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 was coined by Eric Evans in his book of the same title.[2]

Contents

Concepts

Concepts of the model include:

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 is the domain of the software;

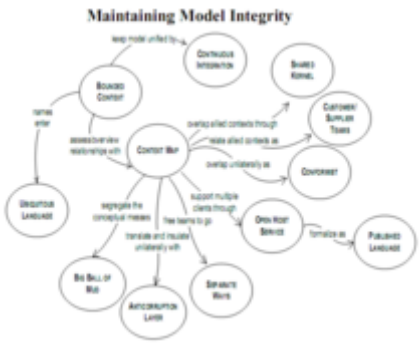
Model
: A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain;

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 domain-driven design

Ideally, it would be preferable to have a single, unified model. While this is a noble goal, in reality it typically fragments into multipl models. It is useful to recognize this fact of life and work with it.

Strategic Design is a set of principles for maintaining model integrity, distillation of the Domain Model and working with multiple models.



Patterns in strategic domain-driven design and the relationships between them

Bounded context

Multiple models are in play on any large project. Yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confusing. It is often unclear in what context a model should not be applied.

Therefore: Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside.

Continuous integration

When a number of people are working in the same bounded context, there is a strong tendency for the model to fragment. The bigger the team, the bigger the problem, but as few as three or four people can encounter serious problems. Yet breaking down the system into ever-smaller contexts eventually loses a valuable level of integration and coherency

Therefore: Institute a process of merging all code and other implementation artifacts frequently, with automated tests to flag fragmentation quickly. Relentlessly exercise the ubiquitous language to hammer out a shared view of the model as the concepts evolve in different people's heads.

Context map

An individual bounded context leaves some problems in the absence of a global view. The context of other models may still be vague and in flux.

People on other teams won't be very aware of the context bounds and will unknowingly make changes that blur the edges or complicate the interconnections. When connections must be made between different contexts, they tend to bleed into each other

Therefore: Identify each model in play on the project and define its bounded context. This includes the implicit models of non-object-oriented subsystems. Name each bounded context, and make the names part of the ubiquitous language. Describe the points of contact between the models, outlining explicit translation for any communication and highlighting any sharing. Map the existing terrain.

Building blocks

In the book *Domain-Driven Design*,[2] a number of high-level concepts and practices are articulated, such as *ubiquitous language* meaning that the domain model should form a *common language* given by domain experts for describing system requirements, that works equally well for the business users or sponsors and for the software developers. The book is very focused on describing the domain layer as one of the common layers in an object-oriented system with a multilayered architecture. In DDD, there are artifacts to express, create, and retrieve domain models:

Entity

An object that is not defined by its attributes, but rather by a thread of continuity and its identity.
Example: Most airlines distinguish each seat uniquely on every flight. Each seat is an entity in this context. However, Southwest Airlines, EasyJet and Ryanair do not distinguish

between every seat; all seats are the same. In this context, a seat is actually a value object.

Value object
: An object that contains attributes but has no conceptual identity. They should be treated as immutable.
Example: When people exchange business cards, they generally do not distinguish between each unique card; they only are concerned about the information printed on the card. In this context, business cards are value objects.

Aggregate
: A collection of objects that are bound together by a root entity, otherwise known as an aggregate root. The aggregate root guarantees the consistency of changes being made within the aggregate by forbidding external objects from holding references to its members.
Example: When you drive a car, you do not have to worry about moving the wheels forward, making the engine combust with spark and fuel, etc.; you are simply driving the car. In this context, the car is an aggregate of several other objects and serves as the aggregate root to all of the other systems.

Domain Event
: A domain object that defines an event (something that happens). A domain event is an event that domain experts care about.

Service
: When an operation does not conceptually belong to any object. Following the natural contours of the problem, you can implement these operations in services. See also Service (systems architecture).

Repository
: Methods for retrieving domain objects should delegate to a specialized Repository object such that alternative storage implementations may be easily interchanged.

Factory
: Methods for creating domain objects should delegate to a specialized Factory object such that alternative implementations may be easily interchanged.

Disadvantages

In order to help maintain the model as a pure and helpful language construct, the team must typically implement a great deal of isolation and encapsulation within the domain model. Consequently, a system based on domain-driven design can come at a relatively high cost. While domain-driven design provides many technical benefits, such as maintainability, Microsoft recommends that it be applied only to complex domains where the model and the linguistic processes provide clear benefits in the communication of complex information, and in the formulation of a common understanding of the domain.[3]

Relationship to other ideas

Object-oriented analysis and design
: Although, in theory, the general idea of DDD need not be restricted to object-oriented approaches, in practice DDD seeks to exploit the advantages that object-oriented techniques make possible. These include entities/aggregate roots as receivers of commands/method invocations and the encapsulation of state within foremost aggregate roots and on a higher architectural level, bounded contexts.

Model-driven engineering (MDE) and Model-driven architecture (MDA)
: While DDD is compatible with MDA/MDE (where MDE can be regarded as a superset of MDA) the intent of the two concepts is somewhat different. MDA is concerned more with the means of translating a model into code for different technology platforms than with the practice of defining better domain models. : The techniques provided by MDE (to model domains, to create DSLs to facilitate the communication between domain experts and developers,...) facilitate the application of DDD in practice and help DDD practitioners to get more out of their models. Thanks to the model transformation and code generation techniques of MDE, the domain model can be used not only to represent the domain but also to generate the actual software system that will be used to manage it. This picture shows a possible representation of DDD and MDE combined.

POJOs and POCOs

POJOs and POCOs are technical implementation concepts, specific to Java and the .NET framework respectively. However, the emergence of the terms POJO and POCO reflect a growing view that, within the context of either of those technical platforms, domain objects should be defined purely to implement the business behaviour of the corresponding domain concept, rather than be defined by the requirements of a more specific technology framework.

The naked objects pattern

Based on the premise that if you have a good enough domain model, the user interface can simply be a reflection of this domain model; and that if you require the user interface to be a direct reflection of the domain model then this will force the design of a better domain model.[4]

Domain-specific modeling (DSM)

DSM is DDD applied through the use of Domain-specific languages.

Domain-specific language (DSL)

DDD does not specifically require the use of a DSL, though it could be used to help define a DSL and support methods like domain-specific multimodeling.

Aspect-oriented programming (AOP)

AOP makes it easy to factor out technical concerns (such as security, transaction management, logging) from a domain model, and as such makes it easier to design and implement domain models that focus purely on the business logic.

Command Query Responsibility Segregation (CQRS)

CQRS is an architectural pattern for separation of reads from writes, where the former is a Query and the latter is a Command. Commands mutate state and are hence approximately equivalent to method invocation on aggregate roots/entities. Queries query state but do not mutate it. CQRS is a derivative architectural pattern from the design pattern called Command and Query Separation (CQS) which was coined by Bertrand Meyer. While CQRS does not require DDD, domain-driven design makes the distinction between commands and queries, explicit, around the concept of an aggregate root. The idea is that a given aggregate root has a method that corresponds to a command and a command handler invokes the method on the aggregate root. The aggregate root is responsible for performing the logic of the operation and yielding either a number of events or a failure (exception or execution result enumeration/number) response OR (if Event Sourcing (ES) is not used) just mutating its state for a persister implementation such as an ORM to write to a data store, while the command handler is responsible for pulling in infrastructure concerns related to the saving of the aggregate root's state or events and creating the needed contexts (e.g. transactions).

Event Sourcing (ES)

An architectural pattern which warrants that your entities (as per Eric Evans' definition) do not track their internal state by means of direct serialization or O/R mapping, but by means of reading and committing events to an event store. Where ES is combined with CQRS and DDD, aggregate roots are responsible for thoroughly validating and applying commands (often by means having their instance methods invoked from a Command Handler), and then publishing a single or a set of events which is also the foundation upon which the aggregate roots base their logic for dealing with method invocations. Hence, the input is a command and the output is one or many events which are transactionally (single commit) saved to an event store, and then often published on a message broker for the benefit of those interested (often the views are interested; they are then queried using Query-messages). When modeling your aggregate roots to output events, you can isolate the internal state event further than would be possible when projecting read-data from your entities, as is done in standard n-tier data-passing architectures. One significant benefit from this is that tooling such as axiomatic theorem provers (e.g. Microsoft Contracts or CHESS) are easier to apply, as the aggregate root comprehensively hides its internal state. Events are often persisted based on the version of the aggregate root instance, which yields a domain model that synchronizes in distributed systems around the concept of optimistic concurrency.

Tools

Practicing DDD does not depend upon the use of any particular software tool or framework. Nonetheless, there is a growing number of open-source tools and frameworks that provide support to the specific patterns advocated in Evans' book or the general approach of DDD. Among these are:

- Actifsource is a plug-in for Eclipse which enables software development combining DDD with model-driven engineering and code generation.
- Apache Isis is a Java framework for developing domain-driven and RESTful applications using the Naked Objects pattern.
- ECO (Domain Driven Design) Framework with database, class, code and state machine generation from UML diagrams by CapableObjects.
- OpenMDX: Open source, Java based, MDA Framework supporting Java SE, Java EE, and .NET. OpenMDX differs from typical MDA frameworks in that *"use models to directly drive the runtime behavior of operational systems"*.
- OpenXava: Generates an AJAX application from JPA entities. You only need to write the domain classes to obtain a ready to use application.
- Restful Objects is a standard for a Restful API onto a domain object model (where the domain objects may represent entities, view models, or services). Two open source frameworks (one for Java, one for .NET) can create a Restful Objects API from a domain model automatically, using reflection.
- CubicWeb is an open source semantic web framework entirely driven by a data model. High-level directives allow to refine the data model iteratively release after release. Defining the data model is enough to get a functioning web application. Further work is required to define how the data is displayed when the default views are not sufficient.
- ENode: is a C# framework which support to develop DDD+CQRS+Event Sourcing architecture style applications.

See also

- Domain of a function
- Event Storming
- Knowledge representation
- Ontology (information science)
- Semantic analysis (knowledge representation)
- Semantic networks
- Semantics

References

1. *Domain driven design* (http://www.domaindrivendesign.org/)
2. Evans, Eric (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software* (http://dddcommunity.org/book/evans_2003/). Addison-Wesley. ISBN 978-032-112521-7. Retrieved August 12, 2012..
3. Microsoft Application Architecture Guide, 2nd Edition (http://msdn.microsoft.com/en-us/library/ee658117.aspx#DomainModelStyle)
4. Haywood, D (2009), *Domain-Driven Design using Naked Objects* (http://www.pragprog.com/titles/dhnako/domain-driven-design-using-naked-objects) Pragmatic Programmers

External links

- *Domain Driven Design, Definitions and Pattern Summaries* (PDF), Eric Evans, 2015
- Implementing Aggregate root in C# language
- *An Introduction to Domain Driven Design*, Methods & tools
- Haywood, Dan, *Domain Driven Design using Naked Objects* (interview), InfoQ
- *How To Define Bounded Contexts*