# Why Domain Driven Design?

Domain Driven Design is a tool to tackle business logic complexity. It is an organization structure and practices that allow our domain to be isolated, reused, richly modeled, and closely match the world our business experts reside in.

**Software Design**

Prior to Domain Driven Design, there was a large tendency for domain logic to be spread out, and not contained. The domain logic would not be given the priority it deserved. Often UIs or database layers would be presented with equal importance, when in fact they are just implementation details. By completely isolating the domain layer, complex business operations can live uncorrupted. There are specific organization structures that DDD suggests to help manage the complexity of this layer. Without these types of practices, our domain can reach a complexity level that is too much for an individual to understand. At this point maintenance costs begin to rise, and the maintainability of the application tends to decrease until the application becomes to costly.

“Projects that have no domain model at all, but just write code to fulfill one function after another, gain few of the advantages of knowledge crunching and communication…. A complex domain will swamp them. On the other hand, many complex projects do attempt some sort of domain model, but they don’t maintain a tight connection between the model and the code. The model they develop, possibly useful as an exploratory tool at the outset, becomes increasingly irrelevant and even misleading.” **[Evans2014]** p.47

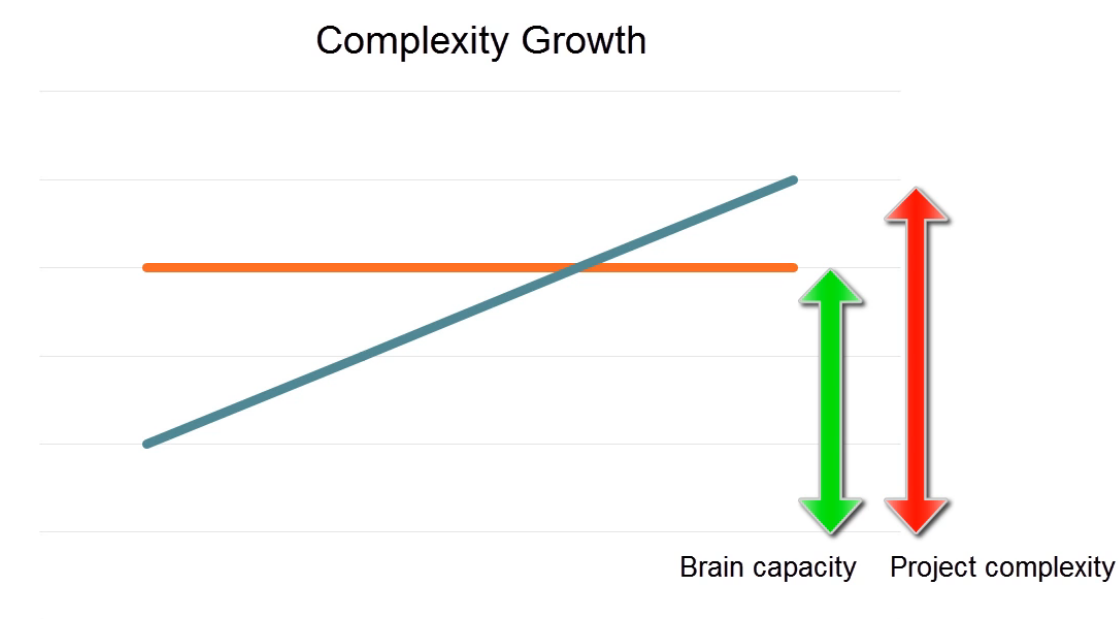


Image Courtesy of Vladimir Khorikov’s Domain Driven Design In Practice Pluralsight Course

At the time DDD was written, it was very common for software to be modeled twice, once using an *analysis model*which represented the problem domain in the real world to business users, and a *design model* which was used for development purposes.

While this approach is largely unused today, Evans suggests having one model work for both purposes. Your design model should be written in such a way that it accurately reflects the business domain as understood by the business experts. It should be understandable to a degree by your business experts.

**Communication**

Often developers were completely isolated and did not have any relationship with or understanding of the business experts. As a result, the software often did not meet the needs of the users. Communication and understanding between the business experts and the developers is given an extremely high priority in DDD. It states that a developer should put forth their best effort to understand the problem domain and become a expert themselves.

If you would like to use DDD, make certain that there is enough business logic complexity to justify it.

DDD is often represented with the onion architecture, with the domain at the center.

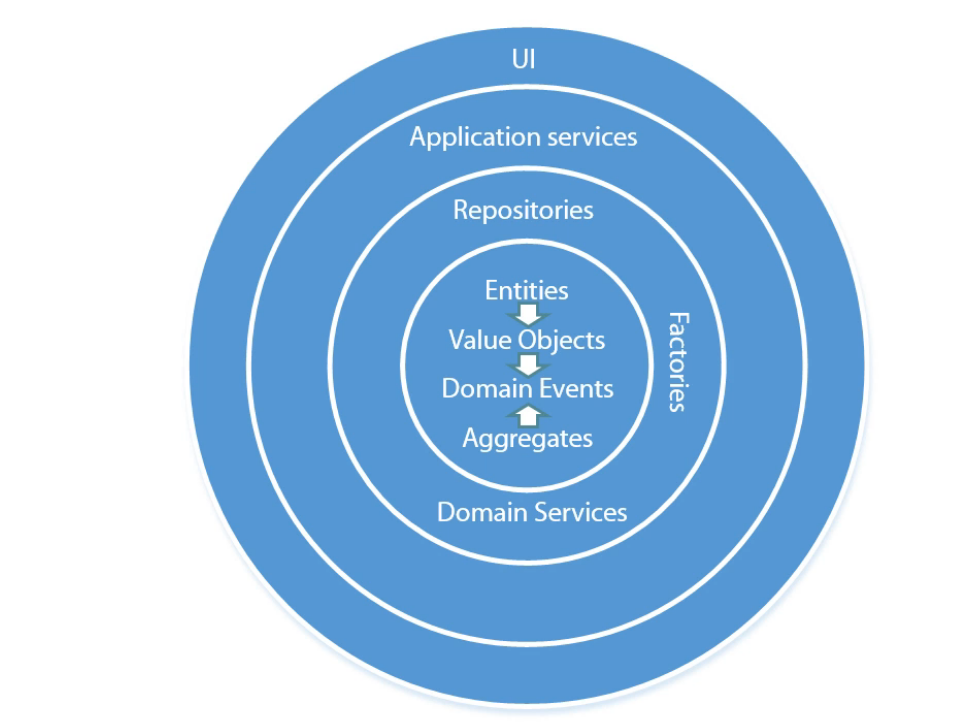


Image Courtesy of Vladimir Khorikov’s Domain Driven Design In Practice Pluralsight Course

You can certainly still model this with the old school N tier design, however I feel it doesn’t convey the same message and significance of the application core.

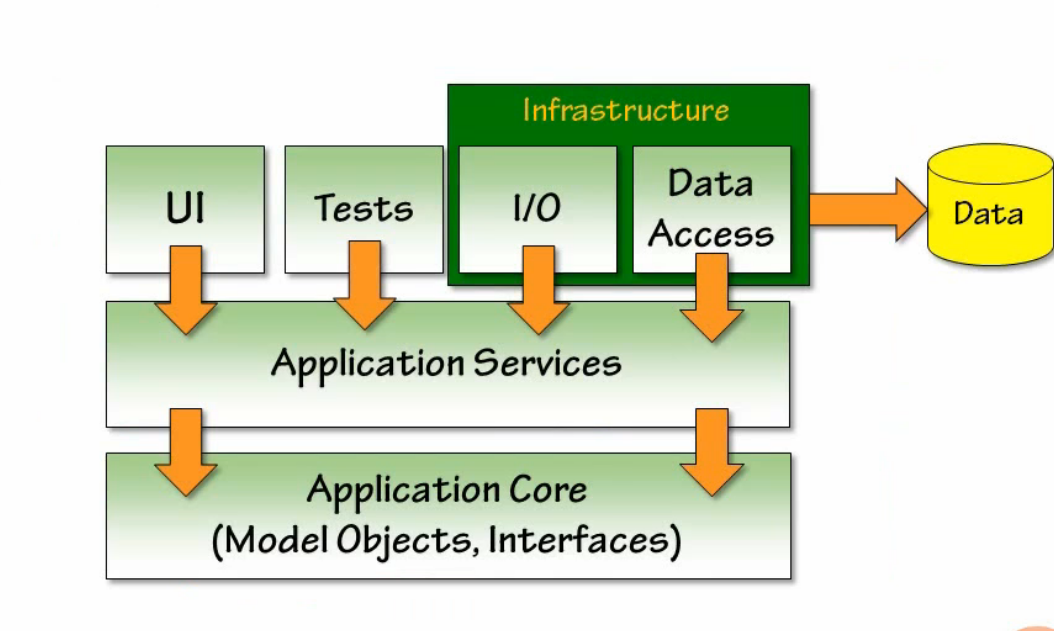


Image Courtesy if Steve Smith’s Creating N Tier Applications in C# Part 1 Pluralsight Course

# DDD Concepts

**Ubiquitous Language**

Ubiquitous Language in DDD is about establishing one common language that everyone associated with the project uses. This includes your business experts, your business analysts, and your developers. Everyone.

As developers, we should seek out any ambiguities and eliminate them with clear terminology. If there are duplicate terms, we should eliminate one. If there multiple terms that represent variations on a concept, we should take note of that and make certain that the variation is present in our code through inheritance or composition.

This can add tremendous value and greatly simplify communication and reduce misunderstandings. It also greatly increases the maintainability in your code. If you business experts, requirements, and code use these terms, it will be extremely easy to find and alter behavior within our existing code. If I have a business rule which needs to be altered, I am guaranteed that the business rule will be present as a class, method, or property somewhere within the Application Core.

“A project faces serious problems when its language is fractured. Domain experts use their jargon while technical team members have their own language tuned for discussing the domain in terms of design. The terminology of day-to-day discussions is disconnected from the terminology embedded in the code (ultimately the most important product of a software project). And even the same person uses different language in speech and writing, so that the most incisive expressions of the domain often emerge in a transient form that is never captured in the code or even in writing. Translation blunts communication and makes knowledge crunching anemic. Yet none of these dialects can be a common language because none serve all needs.” **[Evans2014]** p.25

**Ubiquitous Language Common Problem Areas**

Queries – use specification pattern

Validation – use specification pattern

Exceptions – Prefer custom exceptions with specific names of general exceptions

“Use the model as the backbone of a language. Commit the team to exercising that language relentlessly in all communication within the team and the code. Use the same language in diagrams, writing, and especially speech.

Iron out difficulties by experimenting with alternative expressions, which reflect alternative models. Then refactor the code, renaming classes, methods, and modules to confirm to the new model. Resolve confision over terms in conversation, in just the way we come to agree on the meaning of ordinary words. Recognize that a change in the **UBIQUITOUS LANGUAGE** is a change to the model. Domain experts should object to terms or structures that are awkward or inadequate to convey domain understanding; developers should watch for ambiguity or inconsistency that will trip up design.”

**[Evans2014]** p.27

**Associations**

“It is important to constrain relationships as much as possible. A bidirectional association means that both objects can be understood only together. When application requirements do not call for traversal in both directions, adding a traversal direction reduces dependence and simplifies design.” **[Evans2014]** p. 83

In addition to what Eric mentions above, I would like to take this a step further and state that it is a good idea to eliminate traversals altogether if possible. You should only traverse to other entities if you have a very good reason to. Unidirectional and **especially** bidirectional mappings couple classes together and can be **VERY** difficult to undo once an application has matured.

All in all, while simple in theory, when taken into context with a very complex domain and the complexities that come along with OOP, this can be very challenging to accomplish.

**Entities**

An entity is an object that conceptual identity is determined based upon a key, and not based upon property values.

“In a software system for tracking accounts due, that modest ‘Customer’ object may have a more colorful side. It accumulates status by prompt payment or is turned over to a bill-collection agency for failure to pay. It may lead a doulbe life in anoher system altogether when the sales force extracts customer data into its contract management software. In any case, it is unceremoniously squashed flat into a database table. When new business stops flowing from that source, the customer object will be retired to an archive, a shadow of its former self.

Each of these forms of the customer is a different implementation based on a different programming language  and technology. But when a phone call comes in with an order, it is important to know: Is this the customer who has the delinquent account? Is the the cusomer that Jack has been working with for weeks? Is this a completely new cusotmer?” **[Evans2014]** p.90

Entities can be matched across multiple systems, even if their attributes and properties do not match. We must be able to tell them apart if they do match. As long as the identifier matches, they are one conceptual entity. Mistaken identity can lead to data corruption.

**Value Objects**

Value objects have no conceptual identity, and instead are identified and compared exclusively using their attributes and properties.

“When you care only about the attributes of an element of the model, classify it as a value object. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the VALUE OBJECT as immutable. Don’t give it any identity and avoid the design complexities necessary to maintain entities.”

It is a good idea to treat value objects as immutable, because they are often passed around between different objects and manipulated. If I pass a money value object to class A and class B, but class B decides to manipulate the value of the Money Value Object, class A may be affected in a negative way. In complex systems these types of bugs can be extremely difficult to track down. It is a good idea to enforce immutability and encapsulation as a starting point.

In my experience, value objects often exist as members of entities, or as intermediate values in calculations or functions. I may flatten out the value object to be stored as part of an entity in a table within a relational database. When that database entry returns to memory in the application, it is a value object navigation property on an entity. Common examples of value objects are addresses, and money.

**Aggregates**

Aggregates are a conceptual grouping of related classes that should work together as a cohesive unit. These units may have certain “Invariants” which must be maintained between them, in order to be considered valid. For example, class A may have a field that when changed, requires class B to contain a certain value. Guaranteeing the validity of this operation can be difficult or expensive unless specific design considerations are made.

“It is difficult to guarantee the consistency of changes to objects in a model with complex associations. Invariants need to be maintained that apply to closely related groups of objects, not just discrete objects. Yet cautious locking schemes cause multiple users to interfere pointlessly with each other and make a system unusable.” **[Evans2014]** p.126

Instead of cautious locking schemes, we can use an abstraction which DDD refers to as an aggregate. The aggregate is comprised of several members, one of which is an “Aggregate Root.” There is only one aggregate root per aggregate. This aggregate route is responsible for maintaining the consistency and validity of its aggregate. In order to access members of the aggregate, users must speak with the aggregate root. No properties or methods that change properties should be accessible outside of this. If properties or methods become accessible outside of the aggregate root, you have no guarantee invariants are maintained, since that is the aggregate root’s responsibility.

**The Bounded Context**

As the domain continues to grow and grow, we may reach a point where it makes sense to split into separate sections. For example, if the application deals with Sales and delivery, it may make sense to split the two into separate contexts. Contexts are also often created when multiple teams are working on separate parts of the application.

“Multiple models are in play in a large project. Yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confused. It is often unclear in what context a model should *not* be applied.” **[Evans2014]** p.336

Creating bounded contexts also allows you to create highly specific models for each use case. For example, a customer to a salesperson and a customer to a delivery person are very different things, and are treated very differently. They may have an entirely different set of properties and methods. Forcing these two separate definitions of a customer to exist within one class or entity would not be a maintainable approach.

Bounded contexts, similar to microservices, are expected to own their own data and to have separate databases.

**Shared Kernel**

**Services**

In the onion architecture there are many layers that may contain services that have specific responsibilities. The blue book gives examples of three such services:

**Domain Services**

**Application Services**

**Infrastructure Services**

It also provides the following examples:

**Funds Transfer Domain Service**

* Interacts with necessary Account and Ledger objects, making appropriate debit and credits.
* Supplies confirmation of result (transfer allowed or not, and so on).

**Funds Transfer Application Service**

* Digests input.
* Sends message to domain service for fulfillment.
* Listens for confirmation.
* Decides to send notification using infrastructure service.

**Send Notification Infrastructure Service**

* Sends emails, letters, and other communications as directed by the application.

“It can be harder to distinguish application services from domain services. The application layer is responsible for ordering the notification. The domain layer is responsible for determine if a threshold was met. – though this task probably does not call for a service, because it would fit the responsibility of an ‘account’ object.“

This distinction is generally made based upon whether the given service will require business rules that seem like they should fall within the core domain. Domain services generally contain business logic that does not fit nicely into any given entity or aggregate. Application services will not contain business logic, and simply orchestrate between the domain and infrastructure layer. A common pattern for an application service is:

1. Receive Command.
2. Access Db Based on command for appropriate domain entities.
3. Make appropriate call to domain entities.
4. Make appropriate calls to infrastructure services based on the result of the domain entities operations.

There are many helpful articles on this topic that go into deeper detail by:

**Vladimor Khorikhov:**

<https://enterprisecraftsmanship.com/2016/08/25/what-is-domain-logic/>  
<https://enterprisecraftsmanship.com/2016/09/08/domain-services-vs-application-services/>

In the above post, Vladimir points toward application services being strictly for orchestration.

“The method above is part of the application services layer. Here, the decision regarding how much to charge (if charge at all) is made by the *Atm* domain class: it provides means for doing so via its *CanTakeMoney* and *CaluculateAmountWithCommission*methods. The application service then just orchestrates that decision.”

“The main difference between them is that **domain services hold domain logic whereas application services don’t**.”

If you find yourself introducing something that could be construed as domain logic in an application service, that is where a domain service could come into play. Vladimir gives the example of charging a credit card, but failing. We would not want to decide what to do based upon that failure in the application service. The decision of what to do when a charge fails is a business logic decision, and should be placed into a domain service.

However Vladimir does make a concession that there is a grey area, and just urges caution:

“I wouldn’t say that in this particular example these two benefits play a significant part. It’s mostly fine to keep a little bit of logic that doesn’t fit an entity in an application service itself and not introduce a separate domain service every single time. Make sure, however, that this logic is not duplicated and that it is not too complex.”

**Jimmy Bogard:**

<https://lostechies.com/jimmybogard/2008/08/21/services-in-domain-driven-design/>

Jimmy introduces a concept here that was previous not present:

“Services are always exposed as an interface, not for ‘swappability’, testability or the like, but to expose a set of cohesive operations in the form of a contract. On a sidenote, it always bothered me when people say that an interface with one implementation is a design smell.  No, an interface is used to expose a contract.  Interfaces communicate design intent, far better than a class might.”

He speaks about domain services:

“Domain services are the coordinators, allowing higher level functionality between many different smaller parts.  These would include things like OrderProcessor, ProductFinder, FundsTransferService, and so on.“

Application Services:

“Application Services could map outside messages to internal operations and processes, communicating with services in the Domain and Infrastructure layers to provide cohesive operations for outside clients.  Messaging patterns tend to rule Application Services, as the other service layers don’t have a reference back out to the Application Services.  Business rules are not allowed in an Application Service, those belong in the Domain layer.”

**Lev Gorodinki**

<http://gorodinski.com/blog/2012/04/14/services-in-domain-driven-design-ddd/>

Lev probably describes these services in the most unique way of the three:

“Domain services are different from infrastructural services because they embed and operate upon domain concepts and are part of the ubiquitous language. Infrastructural services are instead focused encapsulating the “plumbing” requirements of an application; usually [IO](http://en.wikipedia.org/wiki/Input/output) concerns such as file system access, database access, email, etc. “

“An application service has an important and distinguishing role - it provides a hosting environment for the execution of domain logic. As such, it is a convenient point to inject various [gateways](http://martinfowler.com/eaaCatalog/gateway.html) such as a [repository](http://martinfowler.com/eaaCatalog/repository.html) or wrappers for external services.”

As you can see, there are some general rules of thumb that apply to all of these services and what their responsibilities should be. If you choose to go the route of using these types of services, make certain that whatever you route you go, that you give each layer a clearly defined responsibility within your application. Make certain that your team understands exactly what specific responsibility the domain, application, and infrastructure services play within your application. Consistency is key.

**Domain Events**

What are domain events:

<https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/domain-events-design-implementation>

<https://martinfowler.com/eaaDev/DomainEvent.html>

Domain Event Implementations:

<http://udidahan.com/2009/06/14/domain-events-salvation/>

<https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/>

<https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/>

MediatR

<https://github.com/jbogard/MediatR>

<https://ardalis.com/using-mediatr-in-aspnet-core-apps>

<https://lostechies.com/jimmybogard/2015/05/05/cqrs-with-mediatr-and-automapper/>

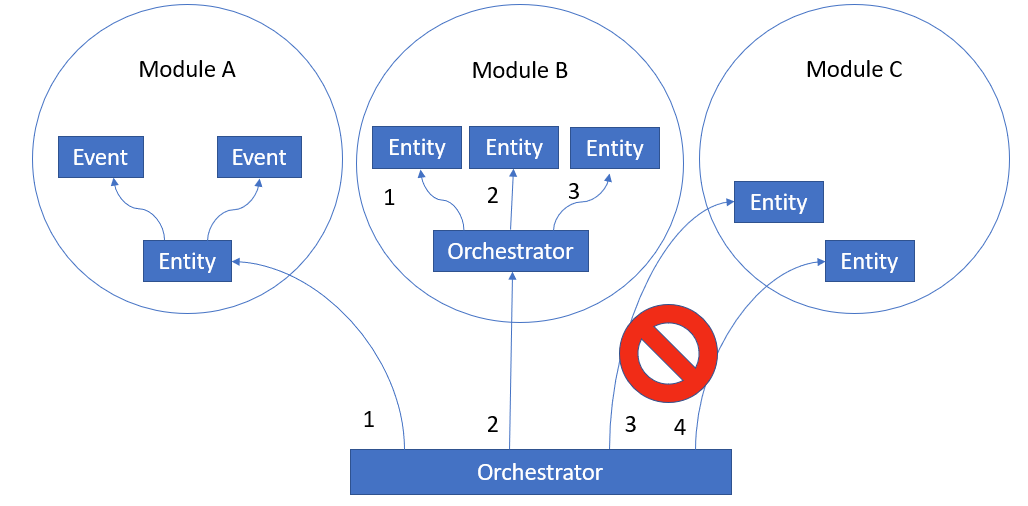
**Trade Offs**

There is a clear trade off between domain events and a more service oriented approach. Domain events allow you to complete follow the [open-closed principle](https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle), and provides excellent decoupling. However, we lose all visibility into the orchestration of our application. If I want to have a clear idea of what happens when an order is placed, I have to chase domain events, and in systems with a lot of domain events, this can be burdensome.

On the other hand, if new functionality is required to be added, and application services are being used, I not only have to add the desired classes to implement the functionality, but I also must violate the open-closed principle and dig into the internals of a domain or application service.

**Event Chaining and Orchestration**

While domain events can be an excellent way to decouple your application, there is a point where is can become very cumbersome. It can be beneficial to combine events with orchestration.



However, make certain to

**References**

**[Evans2014]** Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2014.