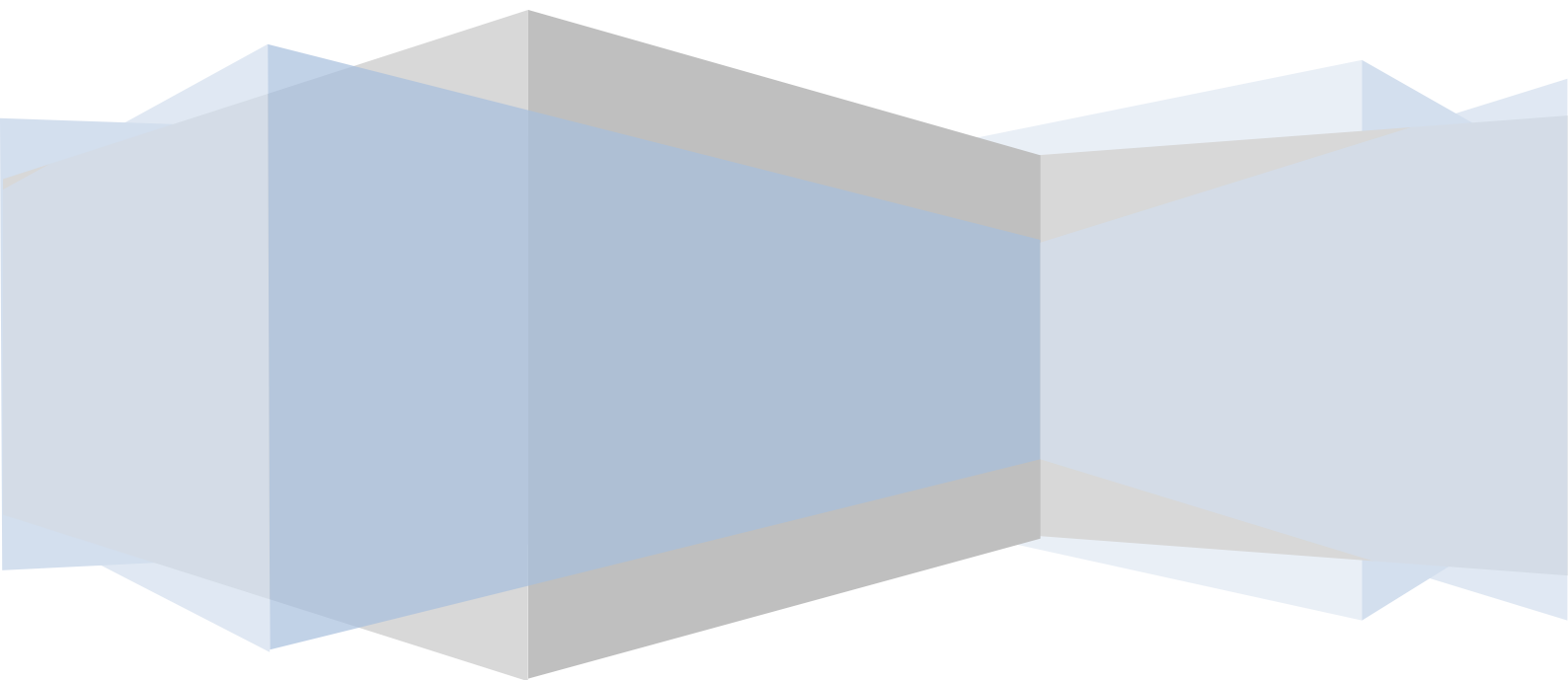


<http://DDDStepByStep.com>

Domain Driven Design

Step by Step Guide

Casey Charlton 2009 (casey@goinsane.co.uk)



Contents

Foreword.....	5
The Ubiquitous Language	6
In Conclusion.....	6
Bounded Contexts.....	8
Example Ecommerce Contexts.....	8
Bounded Context or Context?	9
Surely Bounded Context's Must Interact?	9
In Conclusion.....	9
There Is No Database	10
What is Persistence Ignorance and Why Does it Matter?	10
Where Does Persistence Ignorance Appear in DDD?	10
So Why "There Is No Database"?.....	11
In Conclusion.....	11
Command Query Separation as an Architectural Concept	12
What Does CQS Mean at an Architectural Level.....	12
Does This Mean We Have More Than One Set of Data?	12
Duplication of Data	13
In Conclusion.....	13
Entities and Value Objects	14
Entities	14
Value Objects	14
Hey, I've Heard of These, We Have Them in .Net!	14
So Entities and Value Objects Just Store Data?	15
And We Put These Things Into Repositories Right?	15
In Conclusion.....	15
Where is the Code? A Brief Interlude	16
Aggregates and Aggregate Roots.....	17
A Simple Example.....	17
So What is the Point of an Aggregate?	17
Restrictions on Aggregates and Aggregate Roots.....	18

In Conclusion.....	18
Services	19
So ... Domain Services Are ... What?	19
Shouldn't Logic Be on the Entities Directly?	19
So What Are the Characteristics of a Good Domain Service?.....	19
Some Examples of Services.....	19
Services in the Ubiquitous Language	20
In Conclusion.....	20
DDD: What Kind of Applications Is It Suited To?	21
Where is Domain Driven Design a Good Fit?.....	21
Internal Applications.....	22
Outsourced Applications.....	22
ISV Applications.....	22
Data Centric Applications.....	22
The Bad News.....	22
The Good News.....	22
In Conclusion.....	23
DDD: The Repository Pattern	24
Repository Is Just A Collection	24
Repository Is Just A Facade.....	24
Repository Is Not A Data Access Layer	24
Repository Is Persistence Ignorance	24
Repository Is A Collection ... Of Aggregate Roots.....	24
What Sits Behind A Repository	24
In Conclusion.....	25
DDD: Living In The Enterprise	26
What is SOA Anyway?	26
Our Ecommerce Example	26
SOA Services as a Facade to Bounded Contexts	27
In Conclusion.....	27
The Specification Pattern.....	28
Some Code – A First for the Series.....	28
Removing the Dependencies	29
What Options Do We Have?	30

What Is Double Dispatch?.....	31
Chaining Specifications	31
In Conclusion.....	31
Validation, Consistency and Immutability	32
Validity.....	32
Consistency.....	32
Immutability.....	33
Reference	34
Fundamental Patterns	34
Good Online Reading	34
Books.....	34

Foreword

There is a lot of interest in DDD recently, both in the book, and in the methodology, and in the buzzword.

As a book and methodology, DDD is an excellent way to approach complex software problems, and make them far more understandable and manageable. As a buzzword, DDD is in danger of being corrupted like many other good software practices.

To try and clear up some of the confusion around DDD, I am intending to start a series of short blog posts, covering aspects of DDD and trying to demystify it.

Domain Driven Design is actually pretty simple. It really isn't that hard. That said, developers seem to have a hard time grasping it. I put this down to a great deal of inexperience, with many people who have just read the book in a cursory way saying "we are doing domain driven design" – these people then confuse the issue for others.

Very few people could claim to have done full-on-balls-to-the-wall DDD, due to the relative "newness" of the book, and the time the ideas it contain talk to percolate down. In addition I work in the .Net space, where DDD has taken longer to penetrate than the Java space where the book was formed. I have practiced DDD in some way on three live projects, to a lesser degree in some aspects, and closer to the "one true way" in others. So, like almost everyone doing DDD, I am no expert, beware anyone that claims they are.

That said, some of the "community" have done more with DDD than others. Their wisdom is well worth picking up along the way – with no specific favouritism or deliberate omission, I heartily recommend you read stuff by [Jimmy Nilsson](#), [Greg Young](#), [Colin Jack](#), [Udi Dahan](#) and of course [Eric Evans](#).

Some take the book, Domain-Driven Design: Tackling Complexity in the Heart of Software, to be "the one true bible", but like everything in software, it is only a good starting point. That said, if you are stepping into DDD with more than a gentle dip in the water, this book will prove to be a very solid basis for going forward without drowning – I recommend it strongly.

For a quicker introduction, I recommend ([and have done so in the past](#)), downloading the [InfoQ eBook Domain Driven Design Quickly](#). This distillation of Eric's work provides a really strongly overview of what DDD is, and how it can help you.

When you remember that DDD is really just "OO software done right", it becomes more obvious that strong OO experience will also stand you in good stead when approaching DDD. To do my part, I am intending to start a series of blog posts covering various aspects of DDD, to try and make DDD seem somewhat more approachable, and hopefully to try and show that DDD is a whole lot simpler than many make it appear.

The Ubiquitous Language

Britain and America are two nations divided by a common language
(*George Bernard Shaw / Oscar Wilde*)

It could be said that the business and developers are two nations divided by a common language. If you listen to conversations between developers and the business guys they are writing code for, they seem to speak the same language, but often have entirely different meanings.

This disjoint is what the concept of the **Ubiquitous Language (or UL)** is intended to eliminate in Domain Driven Design.

The concept is simple, that developers and the business should share a common language, that both understand to mean the same things, and more importantly, that is set in business terminology, not technical terminology.

Developers have historically had a hard time with talking to the business – developers are by nature technical, and have a bad habit of letting their technical “geek” terms leak into their conversations. We are all guilty of it at the best of times.

The Ubiquitous Language in DDD is a negotiated language between the Domain Experts and the technical guys – but the rule is that the Domain Experts lead and the technical people follow. As the UL evolves, and this may take weeks or many months, the language becomes more and more refined.

Eric Evans has some good examples in his book of how the UL is drawn out of the conversations between the domain experts and the development team, so I won't duplicate them here – suffice to say you should be listening to how your users and domain experts refer to things, and try and fit in with their language.

One thing to bear in mind here is that although the UL is drawn in the language of the business, it can also include things that might seem like technical terms.

A recent conversation on a DDD list discussed things like Paging and whether these belonged in the domain. Some people then decided that these were technical terms, and on their (IMO misunderstanding) reading of the book, they stated that words like Paging are technical terms and shouldn't be in the UL

That assumption has two basic problems:

- 1) Users have adopted terms like Paging as their own. They know what Paging means ... it is present in 90% of the applications and web pages they use. So Domain Experts adopt this language when expressing their requirements. They probably won't use a term like Paging in regard to the domain, but they may well use it in their user stories, use cases or requirement documents. Whether this concept carries through to the domain is an implementation issue.
- 2) There are many technical terms that exist in the UL already, Eric listed them in the book. These are common pattern names, that the technical team introduce to try and help the Domain Experts formalise their concepts. So terms like Specification and Event may well appear in the UL.

In Conclusion

The Ubiquitous Language is the foundation of Domain Driven Design, it is the basis on which your technical team can become part of the business, and work in their interests, rather than

being see as “the geeks who sit in the corner and deliver buggy software” – don’t be two departments divided by a common language

Bounded Contexts

It was brought up indirectly in a comment on my last post ... the idea of a **Ubiquitous Language (UL)** seems like it is too all encompassing. Indeed it is – unless you are working on a very simple application, one language just isn't enough. Not only that, but a single “mammoth model” will also be too large, unwieldy, and eventually turn into a [Big Ball of Mud](#). That is where the idea of **Bounded Contexts** come into the game.

Bounded Contexts in DDD is one of the “advanced” topics, it doesn't even appear in the book until well over halfway. However, I am going to introduce it now, so that later in this series, the terms don't become an obstacle... in the book you could flick a few hundred pages ahead, here you don't have that luxury. Feel free to skip this post until later if it doesn't make much sense now.

Suffice to say you could model your entire domain as a single thing, with a language to cover everything, but if you look at the business you are trying to model, it is unlikely that is how they see things.

An example of **Context** here would be within an ecommerce system. Although you could model this as a single system, it would also warrant splitting into separate **Contexts**. Each of these areas within the application have their own **Ubiquitous Language**, their own **Model**, and a way to talk to other **Bounded Contexts** to obtain the information they need.

Example Ecommerce Contexts

Well the obvious starting point would be the customer facing **Context** – let's call this the **Shopping Context** – this is the part of the application that lets a customer browse the catalogue, search for items, pick items, and put them in their shopping basket.

You may be thinking this is it for a simple ecommerce system – and for a simple system it probably is. However on larger systems there are other **Contexts** that should probably come into play

When the customer clicks “checkout” then we need to process that order, this is going to involve the **Ordering Context** to take the shopping basket, process the payment and to create the relevant instructions for picking, the **Inventory Context** (which may have been involved earlier to display “items in stock”) to reserve the items in the warehouse, and the list may go on depending on the scale of the entire application.

Each of these **Bounded Contexts** has a specific responsibility, and can operate in a semi-autonomous fashion. By splitting these apart it becomes more obvious where logic should sit, and we can avoid that [BBOM](#).

Bounded Context or Context?

The term **Context** is a general description of a grouping of concepts, the term **Bounded Context** is more specific – a **Bounded Context** is an area of your application which has explicitly defined borders, has it's own model, has its own **Model**, and maintains it's own code. Within the **Bounded Context** everything should be strictly consistent.

Usually, we can use the terms **Context** and **Bounded Context** interchangeably, though I tend to talk in terms of **Context** about the business side of things, and the term **Bounded Context** about the technical implementation.

In essence, a **Bounded Context** is an application in its own right. By splitting your large application into many **Contexts**, you gain all the advantages of modularity, separation of concerns and loose coupling – but on a much larger scale than many developers often think.

Surely Bounded Context's Must Interact?

Indeed they do need to. Within DDD this is done using a **Context Map**. Again, I will skip the details for now, but suffice to say, a **Context Map** is a clearly defined way of talking between **Bounded Contexts**. The book has a number of examples of how this could be implemented, and many more will become apparent as you progress. The **Context Map** is the contract your **Bounded Context** provides to the outside world, other **Bounded Contexts**

In Conclusion

We skipped a fair way ahead here, so don't get too stuck up on Contexts, just bear in mind they exist, and they resolve issues around conflicting Ubiquitous Language terms, allow Models to focus on specific problems, and provide a loosely coupled application architecture.

There Is No Database

Do not try to bend the spoon; that's impossible. Instead only try to realize the truth:
There is no spoon.
The Matrix

Again prompted by discussion on the DDD Yahoo list, this post is intended to explain what on the face of it is a pretty dumb assertion – there is no database in DDD!

I'm sure right at this point, [jdn](#) is reaching for his keyboard to point out that I am dangerously close to repeating [The Tao of Domain Driven Design](#). Yes, I love my mystical analogies!

But if you think about this apparently stupid statement, in the context of Domain Driven Design, it has a subtle and important point – DDD is all about the domain, not about the database, and **Persistence Ignorance (PI)** is a very important aspect of DDD.

What is Persistence Ignorance and Why Does it Matter?

This relatively new principle says we should try and keep our code free of anything that refers to, relates to, or propagates, aspects of the mechanism we intend to use for persisting our data – traditionally the good old relational database.

Early versions of this were focused on the imaginary requirement that we should be able to change from SQL Server to Oracle with relatively little effort. Changing connection strings wasn't good enough, so various frameworks and adapters appeared to try and make this happen. Frequently however, knowledge of not only what specific database would be used, but even the fact that the data would be stored in a database leaked into our objects.

With persistence ignorance, we try and eliminate all knowledge from our business objects of how, where, why or even if they will be stored somewhere.

This is a major step towards decoupling our real code and logic from infrastructure concerns like data access, it therefore benefits us hugely in terms of testing, maintenance and change.

Where Does Persistence Ignorance Appear in DDD?

Pretty much everything in DDD is designed around **PI**, the basic pattern that is used for accessing **Entities** and **Value Objects** (the core building blocks of DDD which I will cover in my next post) is the **Repository**.

I won't cover **Repository** in detail at this point in the series, but the primary purpose of a **Repository** is to represent itself as a collection, albeit a collection with fairly specific and advanced querying. The **Repository** provides a simple and powerful mechanism for making your Domain totally unaware of the actual persistence framework you are using behind it.

As a **Repository** is a collection, anything that uses it is completely unaware of where the **Entity** or **Value Object** may or may not have been stored.

So Why “There Is No Database”?

Domain Driven Design states specifically, in the name, why – we are designing our applications from the point of view of the **Domain**, and the **Domain** is drawn from the **Ubiquitous Language** we negotiate with our **Domain Experts**.

If we were to start with a database the it would be Database Driven Design. If we were to consciously or subconsciously bring elements from the database into our **Domain**, then we would risk compromising our **Domain** model purely to support the persistence layer we had chosen.

In Conclusion

Start from the premise “there is no database”, try to evolve the **UL**, the **Domain**, and everything that entails, without concern to the database you may or may not use in the end.

Eric covers in his book points where you may need to compromise one side or other, and good reasons for doing so, but if you start from a pure point of view, at least you will be making informed and explicit decisions about your compromises.

Command Query Separation as an Architectural Concept

There has been some confusion recently around a recent evolution of DDD, the idea of Command Query Separation (CQS) at the architectural level.

This post is yet again jumping ahead into the “advanced” stuff, so go ahead and skip this if you just want to get the basics down first... those will come soon I promise.

Many of you will be familiar with **CQS** in its original form, Bertrand Meyer presented it as:

“every method should either be a command that performs an action, or a query that returns data to the caller, but not both. In other words, asking a question should not change the answer”
[wikipedia](#)

On the method level it is an excellent principle to follow, but [Greg Young introduced](#) (at least I heard it from him first) the idea of Command Query Separation at the architectural level. This was also [discussed a long time back](#) by such luminaries as Eric Evans and Martin Fowler.

What Does CQS Mean at an Architectural Level

In DDD, as presented by Eric Evans originally, the **Repository** concept is responsible for abstracting your **Entities** and **Value Objects** away from the way they are persisted. To retrieve an **Entity**, or a set of **Entities**, you would use a **Repository** method like `.GetAllOutstandingInvoices` or pass a **Specification** into a generic query method.

This is a perfectly viable and valid way of querying your **Entities**, but it does have some issues around some aspects of an application, these are generally referred to as **Reporting** issues. Reporting issues may well be reports in the classic database style, or may be such things as user interfaces that require searching, sorting, paging and filtering of data.

The concept of applying CQS at an architectural level says, our **Domain** and transactional operations will use the **Repositories**, but for **Reporting** operations, we will use a separate mechanism. The Command is the **Domain** operations, the Query is the **Reporting** operations.

Does This Mean We Have More Than One Set of Data?

Perhaps. For the vast majority of systems this shouldn't be an issue, whatever sits behind the **Repositories** can also provide the data for the **Reporting** side of things. This after all is just an issue of **Bounded Context** again – we have a separate **Bounded Context** for **Reporting**, and as **Bounded Contexts** have their own code and data access, this **Context** can encapsulate all read requests, and just read them off the same store that your **Repositories** in your main **Domain** do.

For example, your main **Domain** may be using an ORM like NHibernate sitting behind your **Repositories**. It will provide a rich mapping from your **Entities** into the database behind.

Your **Reporting Context** may well use NHibernate too, but with a simplified model designed for read purposes, or you may even just have an ADO DataReader sitting in the **Reporting Context**, reading directly from the database that NHibernate is writing to elsewhere.

It is certainly viable to have more than one persistence mechanism however and larger more complex systems may well duplicate data. I can almost hear a bunch of you shouting “Heresy!”.

Duplication of Data

“We spend a great deal of effort maintaining data, maintaining integrity, maintaining consistency, and some fool wants to duplicate this problem all over, now we have two lots of data to manage and synchronise!”

Well, no. When you think about it, this is just data for read purposes. And that means that this data can be a subset or a combination of data from our **Domain**. It could be created by events being published from the **Domain**, or it could be created by taking a snapshot of the **Domain** data.

The only issue that really exists is that this data could be stale or inconsistent – it may be 5 seconds out of date, or 10 seconds, or maybe just 1 second – but this data may not be up to date.

Well, of course it may not be up to date... but is any of the data in your system really up to date? Even if you just requested it from your **Domain**, and it appeared on screen, before you hit any key on your keyboard, that data is already stale – by the time you press “update” someone or something else may have modified the data.

Eventually the data may be up to date and consistent, it just may not be the instant you request it.

So yes, the data may be stale, but is that really an issue? (hint: the answer is no)

In Conclusion

It may pay to split your **Reporting** concerns from your **Domain** concerns, keeping your **Domain** clear of ad-hoc reporting concerns, and UI concerns. It may also be far more effort than it is worth if the system does not have either massive numbers of these kinds of queries, or just is not that complex.

However, bear in mind that this is a good option for allowing the Domain to evolve without concerns from other parts of the wider application leaking into it, and is a good tool to have ready when you face this issue.

Entities and Value Objects

Finally, after 5 posts in the series, we get to the beginning point, the basis of all things... **Entities** and **Value Objects**. OK, maybe a small exaggeration, but **Entities** and **Value Objects** (VO) form the core building blocks of Domain Driven applications.

Why has it taken this long to get to something so fundamental? Well, mostly I got distracted by shiny things along the way – topics came up and I felt the need to “get them down on paper” before I forgot where I was. I am at that age where, if I don’t do something when I think of it, I rarely get around to it. Oh yeah, **Entities** and **Value Objects** – almost forgot!

Back in the good old days we used to have things called business objects, these were classes that held some data, had some methods, and we threw into a database.

DDD has refined this concept a little, by splitting the idea of these business objects into two distinct types, **Entities** and **Value Objects**

Entities

“this is my Entity, there are many like it, but this one is mine”

The key defining characteristic of an **Entity** is that it has an Identity – it is unique within the system, and no other **Entity**, no matter how similar is the same **Entity** unless it has the same **Identity**.

Identity can be represented in many ways on an **Entity** – it could be a numeric identifier (the classic CustomerID), it could be a Guid (the classic ... oh never mind), or it could be a natural key (like the CustomerNumber your Customer was given by your CRM system when they first bought from you).

Examples of common Entities are: Customer, Product, Container, Vehicle

Whichever way you choose to represent it, an **Entity** is defined by having an **Identity**.

Value Objects

The key defining characteristic of a **Value Objects** is that it has no Identity. Ok, perhaps a little simplistic, but the intention of a **Value Object** is to represent something by its attributes only. Two **VOs** may have identical attributes, in which case they are identical. They don’t however have any value other than by virtue of their attributes.

Another aspect common to **VOs** is that they should probably be immutable, once created they cannot be changed or altered. You can create a new one, and as they have no identity, that is just the same as changing another one.

Examples of common Value Objects: Money, Address, ProductCode

Hey, I’ve Heard of These, We Have Them in .Net!

Don’t confuse **Entities** and **Value Objects** with Reference Types and Value Types, they sound similar but bear only a passable resemblance. Both **Entities** and **Value Objects** would be usually be implemented in .Net as Reference Types (Class not Struct).

So Entities and Value Objects Just Store Data?

Definitely not! The point of DDD is to create a rich **Domain** – if you only stored data in your **Entities** and **Value Objects**, you would have an anaemic domain model – one of the primary anti-patterns to DDD.

Entities should have methods on them that logically belong there – so a **Customer** might have a `.UpgradeToPreferredStatus` and a **VO** representing **Money** may have a methods to `.Add`, `.Subtract`, and so on. These methods would obviously come from the **Ubiquitous Language** (“we can **Upgrade** a **Customer** to **Preferred Status**”, “we can **Add** two amounts of **Money** together”)

In fact, in DDD there is less concern about what or how the your **Entities** and **VOs** store data, and more concern about how they represent that data and how they manipulate that data.

Some would go so far as to remove Setters and Getters from them entirely – though my personal preference is to remove Setters only, and make Getters representative of the outside “shape” of the **Entity**.

For **VOs** create only constructors, after all they are meant to be immutable.

For **Entities** use only constructors or strong methods to make any changes. Directly changing properties can have nasty side effects, and leave **Entities** in a temporarily invalid state.

And We Put These Things Into Repositories Right?

Not quite, at least not all of them directly, we will cover that in the next part of the series, the concept of Aggregate Roots.

In Conclusion

The “things” in your business will usually be represented by **Entities** and **Value Objects** in DDD. They form the shape of your Domain, and would generally be the things you draw up on that class diagram on the wall.

Where is the Code? A Brief Interlude

Right about now I can hear murmurs, "I haven't seen any code yet"

That is because I view Domain Driven Design firstly as a design methodology, secondly as an architectural style, and lastly as some great software patterns.

I don't believe I am alone in that view, after all it is a significant way into the book before anything resembling UML appears, and even further before anything code-like is introduced. Now you could pickup the book, extract some of the key terms and half a dozen patterns, and undoubtedly you would be writing better software ... but I suspect you may have missed the "wood for the trees" or put another way "You spent so much time focusing on the details, and missed the bigger picture"

DDD is a better way of thinking about software design, it helps you translate what users and businesses need into software that meets those needs. The patterns may make your software more stable or more maintainable, but it is the methodology that guides you to deliver something fit for purpose.

Of course, eventually you are going to need some code, as a comment on my last post indicated. I am going to postpone that for a while I am afraid, I apologise in advance if anyone is eager to get typing. We have a few more fundamentals of DDD to explore first, then I promise I will start to delve a little into how to best implement these things in C# code.

Aggregates and Aggregate Roots

We are family
I got all my sisters with me
Sister Sledge

Some things belong together, like Apple Pie and Ice Cream, or Sonny and Cher. And so it is with **Entities** and **Value Objects (VOs)** – some of them belong together.

Aggregates are groups of things that belong together. An **Aggregate Root** is the thing that holds them all together.

I will warn in advance, as I proof read this post, it was pretty complicated – while I have tried to simplify the concepts, I am not certain I have totally succeeded. Hopefully **Aggregates** will become more clear later in the series when I start exploring them in code.

A Simple Example

In all Object Oriented programming we are used to dealing with objects that have references to other objects, and in DDD it is no different. For example, our **Customer** may have a reference to the customer's **Orders**.

An **Aggregate** is slightly different, where as **Customers** and **Orders** can exist in the system independently, some **Entities** and **Value Objects** make absolutely no sense without their parent. The obvious example following on is **Orders** and **OrderLines**.

OrderLines have no reason to exist without their parent **Order**, nor can they belong to any other **Order**. In this case, **Order** and **OrderLines** would probably be an **Aggregate**, and the **Order** would be the **Aggregate Root**

The rule of **Cascading Delete** is sometimes cited as a good way to tell if you have a group of **Entities** or **VOs** that should be an **Aggregate** – if the parent, in this case the **Order**, was deleted all other parts of that **Aggregate** below **Order** would be deleted too. So if it doesn't make sense that a parent being deleted would also delete all children, then you don't have an **Aggregate**, you just have a good old fashioned reference.

So What is the Point of an Aggregate?

Well the first and most obvious point of an **Aggregate** is that it operates to a large degree as one "thing". Notably the **Aggregate Root** is the single **Entity** that controls access to the children.

Where you may have a **Customer** with operations like `.UpdateToPreferredStatus` and you may have an **Order** with `.GetSumTotal` – the **OrderLines** of an **Order** would not have any logic exposed outside of the **Order Entity** – in other words to add a new **OrderLine**, or to change an **OrderLine**, you would tell the **Order** to make the changes – `Order.AddNewItem` for example.

In this respect, **Aggregates** provide a clean pattern to keep logic where it really belongs.

Another aspect of Aggregate Roots is that they are the Entities that are dealt with by Repositories.

In our examples above, we would have a **Customer Repository**, and an **Order Repository**, but there would not be an **OrderLine Repository**. It is the responsibility of the **Aggregate Root Repository** to deal with persistence of all the children of that **Aggregate**.

Restrictions on Aggregates and Aggregate Roots

The main, and possibly obvious restriction on **Aggregate Roots** is, they must be **Entities**, and cannot be **Value Objects**. Back to the previous post, you will remember that **Entities** have **Identity**, and **Value Objects** do not – you could not ask a **Repository** to retrieve an **Aggregate Root** if it had no **Identity**.

Within an **Aggregate**, the other players can be **Entities** or **VOs** as the domain dictates. For example, expanding our **Order** example further, the **Aggregate** may comprise the **Order** (**Aggregate Root**), the **Order** may have an **OrderNumber** (**Value Object**), some **OrderLines** (**Entities**), and a **Shipping Address** and **Billing Address** (**Value Objects**)

Entities can hold references to any **Aggregate Root**, but never to any other **Entity** or **VO** within the **Aggregate**. To access any other part of the **Aggregate**, you must navigate from the **Aggregate Root**.

How the component parts of an **Aggregate** are persisted is a matter for the implementation behind **Repository**, but if you were using an ORM like NHibernate for example, the changes are that the **Value Objects** would be NHibernate Components nested in their parent entity record and the **Entities** would be old fashioned mappings, each with their own table.

In Conclusion

Aggregates provide a logical grouping of **Entities** and **Value Objects** that belong together at all times. An **Aggregate Root** is the gatekeeper to the **Aggregate**. Each **Aggregate** is treated as a single unit for persistence purposes.

By logically grouping **Entities** and **VOs** in this way, we provide a mechanism to strictly manage a grouping of objects, and a way to allow us to treat a number of different **Entities** and **VOs** as one.

Services

There can be no word more common in development, and no word used for such a multitude of different things as “service”

It was therefore unfortunate that Eric Evans introduced yet another concept of **Service** in DDD, one which has since been referred to by some as a **Domain Service**. However, people often use the term **Service** in isolation, as Eric did in his book – so if it is in relation to the **Domain**, it is probably what I prefer to call a **Domain Service** – otherwise it is likely to be one of those “other” services.

So ... Domain Services Are ... What?

Well, if **Entities** and **Value Objects** are the “things” in our **Domain**, the **Services** are a way of dealing with actions, operations and activities.

Shouldn't Logic Be on the Entities Directly?

Yes, it really should. We should be modelling our Entities with the logic that relates to them and their children. But sometimes that logic either doesn't fit on the Entity, or it would make the Entity bloated and unwieldy.

That is when Services come into the picture. They help us split logic that deals with multiple Entities, or that deals with complex operations or external responsibilities, into a separate structure more suited to the task.

So What Are the Characteristics of a Good Domain Service?

Oddly, they are much the same as the properties of any other good services... but to reiterate Eric Evans directly, they are:

- The operation should be something that relates to a concept that does not naturally fit on an Entity or VO
- The interface is defined in terms of the elements of the Domain Model
- The operation is stateless

The first two of those are pretty obvious and easy to apply, the last is in general a good definition of a Service, and although some would argue there is no such thing as stateless, don't express state explicitly, and presume instance of your Services should not affect each other.

Some Examples of Services

So far we have been using an ecommerce system as a basis for the examples, so I'll continue here.

When a customer adds items to their shopping basket, they will expect to see the items totalled, and things like shipping costs calculated.

These things might conceivably sit on the **Shopping Basket Entity** – but a bit of deeper thought shows that this is probably not a perfect fit. For a simple system the basket might be able to contain all the logic required, but expanding our system and moving it to a DDD focus would probably extrapolate Services to deal with these issues – **PricingService** and **ShippingCostingService** could well be two of the players in this game.

After all, apart from needing to deal with the items in the Shopping Basket, these operations may well require communication with other services, calculation and pricing engines, external suppliers ... in other words, they have a lot of responsibilities that we do not want to bring into our Entity.

Now instead of our ShoppingBasket having .GetTotalPrice and .GetShippingCost methods, we can use a Domain Service to get this information, and keep our Entity focused on the real tasks, managing the items in the ShoppingBasket

Services in the Ubiquitous Language

Entities and Value Objects live comfortably in the UL, they are the “things” that our Domain Experts and Users talk about.

Services also live in the UL, they are the actions that our DEs and Users talk about... where you see a noun in the conversation, it is likely to be an Entity or VO. When you see a verb, it may well be a Service

In Conclusion

Services provide a way of expressing actions and operations within our domain. Where as Entities and Value Objects, the “things”, provide the building blocks, Services provide the plumbing between them and allow us to express more abstract concepts.

Just beware of creating an anaemic domain model; Services are there to support Entities and VOs, not to strip their logic from them.

DDD: What Kind of Applications Is It Suited To?

In many conversations, and in many comments here, you hear phrases like “well that’s not really suited to DDD” or “DDD isn’t the best fit for that problem”. You even see those kind of comments on my blog, and often they are posted by me.

This obviously leaves a number of people confused, after all DDD is the wonder cure for every software illness, surely there is nothing it cannot do!!!!

[Download the eBook of the Series so far ...](#)

Where is Domain Driven Design a Good Fit?

DDD is a software methodology suited to a particular kind of application – one where there is **significant complexity**, and there is a significant focus on a **well defined business model**.

Complexity

The subtitle of Eric Evans book says it all - “Tackling Complexity in the Heart of Software” – DDD is a way of making complex systems more manageable, better understanding their intricacies, and simplifying a subject that could otherwise be overwhelming – i.e. avoiding a classic big ball of mud.

We often end up with a Big Ball of Mud when we started out with a fairly sketchy idea of the requirements we had, and built our software in such a way that rigidity set in early on. Over time as new requirements evolve, we bolt them on to what we had before, and piece by piece we end up with our Mud Ball ... or as I like to imagine it – a Borg Cube – where all your code has been assimilated, and is now part of the collective.



By applying DDD we can break that complexity down into methods of extracting better models from our users and domain experts, and apply software patterns that let us add and evolve to the model, without it becoming a rigid monolith.

Focused Business Model

From within the text of the work by Eric Evans, you will soon find that there is a heavy emphasis upon the business side of the equation – there is constant reference to Domain Experts. These Domain Experts represent the business, and should be intimately familiar with it’s inner workings. Not the workings of the software, or the way the business analysts see the software – but the business itself.

Therefore, you will find it hard if not impossible, to gain full benefit out of DDD if you do not have Domain Experts up close and personal for the duration of the project – this means that doing DDD on an ISV style application will be hard, you just don’t have the access to the business model of your customers in the way that you should.

It will also be hard to fit DDD to your project if your requirements are too flexible or generic, as again a lot of ISV products are by their nature. DDD is very geared towards providing a highly bespoke solution for an individual business.

What DDD is good at is drilling deep into a business model, and turning that business into a software model.

Internal Applications

Obviously from what I have said so far, DDD is best suited to large projects, where the development team is in-house with the business they are providing a solution for. This ensures that you can gain maximum benefit from what DDD has to offer.

Outsourced Applications

If your development team is outsourced, but writing a bespoke application for a single client, then DDD could be made to work. The key to doing this successfully would be ensuring you had constant access to domain experts, and that their time was not in high demand elsewhere. Constant verification with the Domain Experts would be an important aspect of making this work, and obviously this would be easier with the team on the client's site.

ISV Applications

By their nature, ISV applications have two qualities that are not a good fit for DDD ... they have very generic requirements, and they have highly user customisable models. These kind of applications may well benefit from a lot of the common sense in DDD, and some of the patterns, but DDD will be no magic bullet here.

Data Centric Applications

Some applications are fundamentally just data manipulation programs. Obviously all applications deal with moving data around, but some live for the data and some live for the logic. DDD is very firmly in the logic camp – so is not suited to applications where the primary focus is on retrieving data, modifying that data, and then putting it back into a database.

For data centric operations you would probably be better off using something like an Active Record pattern, or even a DAL over stored procedures. You may find some benefits in some of the more cursory aspects of DDD, and perhaps using some of the terminology, but trying to make DDD fit here will not be a pleasant experience.

The Bad News

Probably 95% of all software applications fall into the “not so good for using DDD” categories.

Most are fundamentally Data Centric – most websites are, most desktop applications are ... fundamentally most data updating and reporting applications are data centric.

Of the remaining applications, a large percentage don't come close to being complex. They may be tricky, or big, but not complex.

The Good News

For the 5% of applications where DDD is a good fit, it is a very good fit. For those situations, DDD will help you crack a very tough nut. Here, DDD may be the silver bullet for that werewolf that your manager just pointed to your desk.

And even if your application isn't in that 5%, there is a wealth of wisdom and experience encapsulated in Domain Driven Design – use what you think applies to your situation, and you

will find your software becoming more flexible, more reactive to your audience, and easier to understand – just don't expect miracles, and beware of over complicating your code for the sake of it – sometimes simpler really is better.

In Conclusion

There are no hard and fast rules about what DDD is best suited to, and I just made those statistics up, but no software methodology is going to be the miracle cure.

Take from DDD what you think benefits you and your situation – but accept that it cannot be everything to every person.

DDD: The Repository Pattern

I seem to have taken a fairly long time to get here, and it has been mentioned in passing, but now we get to the last major part of the Domain Driven Design picture – **Repository**.

In traditional architectures, your application talks to some kind of database layer, and asks it to save or retrieve your objects. DDD is slightly different, after all, remember I said “[There is no database](#)”

Of course, there is a high likelihood you do *actually* have a database, but the **Repository** pattern ensures you need to take little to no concern of it within your domain.

Repository Is Just A Collection

The primary thing that differentiates a Repository from a traditional data access layer is that it is to all intents and purposes a Collection semantic – just like `IList<T>` in .Net

Repository Is Just A Facade

While Repository pretends to be a Collection to your domain, it is actually just a variation on the Facade pattern – it takes a complex subsystem (persistence or the database), and wraps it with a simpler interface (a Collection)

Repository Is Not A Data Access Layer

Well, at least not in the traditional sense – it doesn’t talk in terms of “data” it talks in terms of Aggregate Roots. You can tell your Repository to add an Aggregate Root into it’s collection, or you can ask it for a particular Aggregate Root. When you remember that Aggregate Roots may comprise one or many Entities and Value Objects, this makes it fairly different to a traditional DAL that returns you back a set of rows from your database tables.

Repository Is Persistence Ignorance

An important part of any properly decoupled architecture is Persistence Ignorance – your application or domain should be completely unaware of how or even if your data is persisted – it should just expect it to happen. In DDD the Repository pattern achieves this by a combination of the first three headings above – it pretends to be a collection, acts as a facade onto your actual persistence layer, and it does not act like a DAL

The Repository is the seam between your domain, and the technical implementation you use to store and retrieve your data.

Repository Is A Collection ... Of Aggregate Roots

The Repository pattern is not just a thin DAL – it is responsible for talking in Aggregate Roots only. A single Aggregate may contain 2,3 or more Entities and Value objects – so our Order entity may also contain OrderLines, there would not be a Repository for OrderLines, it would be up to the Order Repository to persist an Order and all the OrderLines it contains.

What Sits Behind A Repository

Pretty much anything you like.

Yep, you heard it right. You could have a database, or you could have many different databases. You could use relational databases, or object databases. You could have an in memory database, or a singleton containing a list of in memory items. You could have a REST layer, or a set of SOA services, or a file system, or an in memory cache...

You can have pretty much anything – your only limitation is that the Repository should be able to act like a Collection to your domain.

This flexibility is a key difference between Repository and traditional data access techniques.

In Conclusion

The **Repository** pattern is a Facade, that abstracts your persistence away from your Domain. On one side it pretends to be a collection – on the other it deals with the technical concerns of your persistence implementation.

Repository provides us with a mechanism to achieve Persistence Ignorance in our Domain.

DDD: Living In The Enterprise

No, not that Enterprise!

The other Enterprise – the big amorphous one that organisation spent a fortune putting SOA around.

Domain Driven Design appears to be at odds with large scale distributed systems, it is after all methodology to design and write application software, and in SOA we don't have applications – we have services (this is one of those times I don't mean Domain Services")



The foundation of SOA is an environment of distributed systems talking via messages, sent between services, probably over message busses – not applications like DDD

What is SOA Anyway?

Nice try – I like to argue, but you won't get me started on a discussion of what SOA is, why it exists, whether it exists, or any silly questions like that. For the sake of this discussion, let us just presume that if you are interested in SOA you probably have what I like to refer to as a VBS ... a VERY BIG SYSTEM! (you may not of course, but someone may have decided that SOA would magically solve all your IT problems, so you have it anyway)

Our Ecommerce Example

Let us return to the ecommerce scenario I have used a few times in this series. This could be a single web application, front end coded with MVC, back end written with DDD, talking to an ORM, producing orders, and sending those through to your sales guys. That doesn't sound very "complex" – and one of the criteria I mentioned earlier as being suitable for DDD were complex applications.

As it happens, an ecommerce website is a pretty poor example for a DDD application – there is little logic, lots of data retrieval and updating, and some pretty graphics. That is, unless you are Amazon.

OK, I'm going to bet you aren't Amazon if you are reading this, but let's presume you have a large sales operation, thousands of orders a day, complex pricing and discount models, departments for the warehouse and the delivery operation, billing and accounts systems to talk to ... now you have a complex application. Sure, on the front end web servers it is much the same if on a larger scale, but behind the scenes, much more is going on.

And if you have all that, then I'll bet you got all cosy with SOA at some point too. You probably have services for Billing, Credit Control, Warehouse and Packing, Stock Control, Delivery, and a lot more.

Earlier on we talked about Bounded Contexts in DDD. I didn't mention at the time, but there was probably an assumption that these were just other parts of the application. For simpler systems they may well be, but as one of the criteria for a Bounded Context is that it has all it's own code and data, the Bounded Context could be local, or remote, depending on your needs.

SOA Services as a Facade to Bounded Contexts

Separation at a local level is good for isolating change and for keeping a logical split of things that do not belong together. On a wider scale, separation allows you to distribute components of a DDD application as the enterprise requires.

An individual Bounded Context in DDD is a self contained application; it just doesn't have a user interface like a traditional application does. Instead it has a Context Map which could well be the set of services that fulfils our requirement to play nicely in the SOA Enterprise. These services could be old fashioned web services, or a REST layer, depending on the requirements; they can talk via a message bus, via synchronous or asynchronous requests, or via RPC requests.

Within the Bounded Context we have a full Domain, we have a full set of Entities, VOs, Domain Services, Repositories and a persistence layer. So we have our DDD application – and we can have as many or as few of these as we need.

In Conclusion

When you need to think of a distributed system, think of individual Bounded Contexts as miniature applications sitting behind a services layer.

The Specification Pattern

Continuing our series on Domain Driven Design, we now get to one of the more interesting patterns in DDD – the Specification.

A Specification is, in simple terms, a small piece of logic that sits on it's own and gives an answer to a simple question ... “does this match?”

With a Specification we split the logic of how a selection is made, away from the thing we are selecting.

We have a Customer, and we want to be able check if they are eligible for a discount on a Product.

If we were to put a method on the Customer entity, for example `.IsEntitledToDiscountPrice(Product)` we start to couple our entities tightly together, and as the number of questions we want to ask of our entity increases, the more polluted its interface becomes.

To avoid this we can use a Specification.

Some Code – A First for the Series

The basic implementation of Specification has a single method `.IsSatisfiedBy`, in our Discount example above we may have a `EligibleForDiscountSpecification`, the method would be `.IsSatisfiedBy(Customer c)`

The actual Specification is variable in how it is defined, but a very simple version for this scenario would be:

```
public interface ISpecification<T>
{
    bool IsSatisfiedBy(T sut);
}

public class EligibleForDiscountSpecification : ISpecification<Customer>
{
    private readonly Product _product;
    public EligibleForDiscountSpecification(Product product)
    {
        _product = product;
    }

    public bool IsSatisfiedBy(Customer customer)
    {
        return (_product.Price < 100 && customer.CreditRating >=
        _product.MinimumCreditRating);
    }
}
```

This now simplifies selection or matching, and the Customer and Product are no longer coupled – our Specification now knows how to decide if the Customer is eligible for a discount. Ignoring the fact that the following is a rotten unit test, we can use our Specification like this:

```
[Fact]
public void TestSpecification()
```

```

{
    var product = new Product() { MinimumCreditRating = 3, Price = 50 };
    var spec = new EligibleForDiscountSpecification(product);
    var goodCustomer = new Customer() { CreditRating = 3 };
    var badCustomer = new Customer() { CreditRating = 1 };

    Assert.True(spec.IsSatisfiedBy(goodCustomer));
    Assert.False(spec.IsSatisfiedBy(badCustomer));
}

```

Two things become very easy when using Specification – we can easily select from lists and we can pass dependencies without creating coupling.

To find all the Customers in a List<Customer> who are eligible for a discount, we can do (again ignoring the terrible unit test:

```

[Fact]
public void TestSpecificationCollection()
{
    var minCredit = 3;
    var product = new Product()
        { MinimumCreditRating = minCredit, Price = 50 };
    var spec = new EligibleForDiscountSpecification(product);

    var customers = new List<Customer>()
        {
            new Customer() {CreditRating = minCredit-2}, // bad credit
            new Customer() {CreditRating = minCredit-1}, // bad credit
            new Customer() {CreditRating = minCredit} // good credit
        };
    IEnumerable<Customer> eligible =
        GetAllCustomersMatching(customers, spec);
    int count = 0;
    foreach (var eligibleCustomer in eligible)
    {
        Assert.True(eligibleCustomer.CreditRating >= minCredit);
        count++;
    }
    Assert.Equal(1, count);
}

public IEnumerable<Customer> GetAllCustomersMatching
    (IList<Customer> customers,
     ISpecification<Customer> specification)
{
    foreach (var customer in customers)
    {
        if (specification.IsSatisfiedBy(customer))
            yield return customer;
    }
}

```

Removing the Dependencies

A scenario I had a while back required an Entity to only allow a method to proceed if a search against a Repository return no matches already there. This specific example turns out to be quite common, people seem to need access to Domain Services or Repositories on a frequent basis.

My first question would be, is the Entity doing too much? Is this an operation that should be in a Domain Service or perhaps a Specification?

Assuming you can legitimately answer “no” to that last question – let us assume that we want to have a Product with a ProductCode – but the ProductCode cannot be duplicated. Leaving aside the fact that this is a rather fake scenario, it means our Product now needs access to the ProductRepository, which is a bit back to front.

What Options Do We Have?

Generally in this scenario we have a few options:

1. The Entity could call the Repository<Product>.GetByProductCode
2. The Entity could call a Domain Service to check that it is a unique code
3. A domain service could hold all the logic and coordinate the Entity with the Repository.

Option (1) is not a good design – an Entity should not be aware of Repositories. It would probably require a Service Locator pattern to achieve, but becomes very messy very quickly. It could possibly be resolved with Double Dispatch, but is still a bit back to front (see 2 below).

Option (2) is a bit back to front – Entities logically sit behind Domain Services. This could work, as alluded to in a comment to a previous blog post on Domain Services, and if so will require Double Dispatch to be clean.

Option (3) is probably the best of all those options. Domain Services sit in front of Repositories, Entities, and other Domain Services.

The last option not on that list is to use Specification.

If we created a NoOtherMatchingProductCodeSpecification, we can cleanly use this to do the work.

```
public class NoMatchingProductCodeSpecification : ISpecification<Product>
{
    private readonly IRepository<Product> _repository;
    public NoMatchingProductCodeSpecification(
        IRepository<Product> repository)
    {
        _repository = repository;
    }

    public bool IsSatisfiedBy(Product product)
    {
        Product match =
            _repository.GetProductByProductCode(product.ProductCode);
        return match == null;
    }
}
```

Please bear in mind, this is a contrived scenario, I know this specific example has issues around ACID etc ... it is here to prove a point, and Kyle Baley made me write this in a hurry!!!!

What Is Double Dispatch?

So far I have mentioned it more than a few times, and even used it in the title of this blog – so it must be important. You have probably used Double Dispatch many times, even if you didn't know it had become a pattern with a proper name.

In simple terms, Double Dispatch means we pass in an object to a function, to let the function take an action or make a decision, based on the logic of the passed in object. This decouples our Entities from the actions they take.

Chaining Specifications

It also happens, another useful side effect of Specification is that they can be chained very easily, but I will leave that till a future post. A starting point [if you are interested now is on Wikipedia](#)

In Conclusion

I seem to have written more code in this post than I intended, and more than has been the style for the rest of the series. I again intend to blame Kyle for that :) Hopefully it has however given you a more concrete idea of where and how the Specification pattern can be of use.

Validation, Consistency and Immutability

There seems to be some confusion around these and similar concepts, so I thought it might be an idea to provide some clarification. Now these things aren't specific to DDD, but they certainly have a lot of relevance there, and often provoke furious debate.

In this series I first mentioned these things back when I introduced Entities and Value Objects, I said that you should try and ensure Entities are valid at all times, and that Value Objects should be immutable. The discussion around Aggregates and Aggregate Roots has also proved to be of some confusion, the definition from Eric Evans of an Aggregate says "A set of consistency rules applies within the Aggregate's boundaries".

Validity

The term "valid" has many meanings in many contexts. When it was mentioned with regards to an Entity, I suggested that you should attempt to ensure that your Entities are always in a valid state, and cannot become invalid. I followed up that this could be achieved by removing property accessors, and only using either constructors or strongly named methods.

Let us go back to our Customer example, and presume that for a Customer to be valid within our Domain, it must have a CustomerNumber, a FirstName, and a LastName. It may also have half a dozen other properties.

So we can ensure this Entity never becomes invalid by removing the property accessors for the CustomerNumber, FirstName, and LastName, and providing a single constructor that allows us to create a new Customer ... `Customer(customerNumber, firstName, lastName)`

Now there is no possibility that this Customer can become invalid - it either has these three properties or it cannot be created. If we want to be able to change the Customer's name, we can create a new method `.UpdateName(firstName, LastName)`

By taking this approach, we ensure that the entity can never be put into an invalid state, for example a developer setting a `.FirstName` property, but forgetting to set the `LastName` at the same time.

Some have taken this concept too far, and assume that as a Customer also has a Policy, that it must always have one of these or it is invalid. Of course, there are things we may want to do with our Customer that will require it to have a Policy, but that does not constitute an invalid Customer, it constitutes a rule that must be applied before we attempt to take the action.

The purpose of ensuring the Entity is always valid prevents us having to do messy things like checking an `.IsValid()` method every time we want to persist our entity, or we write a function that accepts an instance of our entity. Enforcing an "always valid" approach will mean that we do not have to code multiple guard clauses scattered across our applications.

Consistency

The term consistency has come up in this series and around DDD mailing lists. In this respect, consistency is the idea that one piece of data in the system is consistent with another. Here is the bad news:

You cannot ever achieve total consistency

OK - headline over. The point is that something in your application will always be inconsistent. The best example is that when your users have chosen to edit the Customer on their screen they have a copy of that data. By the time they have received that data it will potentially be inconsistent with the domain, and with your persistence store - after all this is why we have the concepts of optimistic and pessimistic locking. When they submit a change to the Customer, it is probably minutes out of date.

Back to our headline, when you accept that it is impossible to achieve total consistency, you can deal with the real problem - how you achieve eventual consistency. Eventually messages will make their way through our domain, eventually data will be persisted, eventually you will have consistency.

While our reporting domain may lag behind our primary domain, does it honestly matter? How important is it that they are both totally consistent? You can certainly achieve a close facsimile of total consistency, but it will come at a high price - does the business want to pay for this?

When you let go of the premise that your data is ever really consistent, it now just becomes an issue of an SLA agreement, if the business really wants data that accurate, then explain the costs and tradeoffs and go with it. If after an explanation they understand that “almost consistent” is good enough for almost all applications, then they just have to make a decision about how important this is to them.

Consistency also applies at a much more granular level, with the domain for example. Entities and Aggregates may be valid, but not consistent with each other at all times. Eventually they should become consistent, but ensuring they are all consistent at all times is very tricky to say the least.

While an Aggregate Root is responsible for ensuring consistency with the Aggregate, it is acceptable, and almost inevitable that Aggregates will be inconsistent with each other at some point in their lifetime.

Again, letting go of the myth of total consistency here will give you far more flexibility.

Immutability

I would hope this is familiar to all - but perhaps some of you saw me mention it and didn't quite get my context.

The concept of immutability is pretty much what it says - the thing cannot be mutated - or in context, an Immutable Value Object can be created, but can never be changed.

If you want to change an immutable object, then you can create a new one and replace the original.

By making objects immutable, we avoid many problems with validation and consistency, only the constructors on the object matter, and only they need to validate the parameters passed in. It is far easier to check all the parameters at a single time than to check individual properties against each other at a later stage.

Reference

<http://dddstepbystep.com>

Wiki available at: <http://dddstepbystep.com/wikis/ddd/default.aspx>

Fundamental Patterns

from: <http://domaindrivendesign.org/discussion/messageboardarchive/Glossary.html>

- [Entities](#)
- [Value Objects](#)
- [Repositories](#)
- [Bounded Context](#)
- [Context Map](#)
- [Aggregates](#)
- [Services](#)
- [Repositories](#)

Good Online Reading

- [Martin Fowler's On CQS: EagerReadDerivation](#)
- [Greg Young's Blog](#)
- [Udi Dahan's Blog](#)
- [Colin Jack's Blog](#)

Books

- [InfoQ Free eBook : Domain Driven Design Quickly](#)
<http://www.infoq.com/minibooks/domain-driven-design-quickly>
- [Domain-Driven Design: Tackling Complexity in the Heart of Software \(Eric Evans\)](#)