# DDD – The Basics

**Rob** / September 18, 2009

Whenever I'm trying to learn the fundamentals of a programming methodology or principle I find it best to write short-ish notes on the subject, the way I used to to when revising for exams. That way if I need to remind myself of something I've got a relatively short reference list.

The following are my notes on the free PDF book, Domain Driven Design Quickly by Abel Avram and Floyd Marinescu. I'd really recommend you download and read this book or better still buy it.

**Domain Driven Development**

In order to create good software you need to know the software is about.

You can't create good banking software unless you know about banking itself, banking being the domain. Who knows the domain best? The bankers, they know the business. They know the details, the catches, the possible issues and the rules. This is where you start, with the domain.

The best way to make the software fit with the domain is for the software to reflect the domain, it has to model the domain it's going to be used in.

We need to create an abstraction of the domain. A domain model is a diagram which captures the essence of a domain expert's knowledge. The model is our internal representation of the target domain. Domains can contain too much information and some of it just isn't relevant for the model.

To make sure we've got the model right we need to able to demonstrate what it's all about :

- Graphical Diagrams
- Use Cases
- Drawings
- In Writing

Once we've got the model then we can look at code design. The key is to try and work out the essential concepts of the domain.

**Ubiquitous / Common Language**

A core principle of domain-driven design is to use a language based on the model. The model is the common ground where the software meets the domain. This language is to be used everywhere, in all communication and in the code, it is the Ubiquitous language.

We need to find key concepts which define the domain and the design and then find corresponding words for them and start to use them. Some are easily spotted but some are not.
How do we move transfer the domain model into code?

To tightly tie the implementation to a model usually requires software development tools and languages that support a modelling paradigm such as object-oriented programming.

Object-oriented programming is suitable for model implementation because they are both based on the same paradigm or concept.

**Layered Architecture**

It's easier to write a system where the business logic or database support code is written into the business objects or business logic is incorporated into the UI.

This makes code maintenance much more difficult though, and automated testing awkward.

A complex program must be partitioned into layers. Designs should be developed within the layers and should only have a dependency on the layer below.

***UI – Application – Domain – Infrastructure***

Code relating to the domain model should be within one later and it should be isolated from the UI, application and infrastructure.

**Layers** :

| | |
|---|---|
| UI / Presentation Layer | Responsible for presenting information to the user and interpreting user commands. |
| Application Layer | Thin layer coordinating application activity. Does not contain business logic.<br>Does not hold the states of business objects but can hold the state of an application task's progress. |
| Domain Layer | Contains information about the domain, the heart of the business software. The state of business objects is held here.<br>Persistence of the business objects and possibly their state is delegated to the infrastructure layer. |
| Infrastructure layer | Acts as a supporting library for all other layers. Provides communication between layers, implements persistence for business objects, contains supporting libraries for the user interface layer etc. |

**Entities – Objects with an identity**

Objects which seem to have an identity which remains the same throughout the state of the software, these objects are Entities.

Usually the identity is either an attribute of the object, a combination of attributes, an attribute specially created to preserve and express the identity or even a behaviour.

It's important for two objects with different identities to be easily distinguishable by the system, and two objects with the same identity to be considered the same by the system.

Define a means of distinguishing each object regardless of its form or history. Be alert to requirements that call for matching objects by attributes.

Define an operation that is guaranteed to produce a unique result for each object, possibly by attaching a symbol that is guaranteed unique. This means of identification may come from the outside or it may be an arbitrary identifier created by and for the system.

**Value Objects – Disposable, immutable objects,**

There are cases when we need to contain some attributes of a domain element. We're interested in the attributes the object has, not which object it is.

An object used to describe certain aspects of a domain, which does not have an identity, is a Value Object.

It's recommended that only objects which conform to the Entity definition are created as Entities and all other objects are created as Value Objects.

Having no identity, Value Objects can be easily created and discarded.

They should be immutable – created with a constructor and never changed. If you want a different one, create it.

If Value Objects are shareable they should be immutable, they should be kept thin and simple. They can be passed by value or a copy created.

**Services – Objects which perform operations on other objects.**

Some aspects of the domain may not easily be mappable to objects.

There may be some actions or verbs in the domain which don't seem to belong to any object. They represent important behaviour of the domain but they're not part of the Entity.

We have to use an object, we can't have a function on its own, it has to be attached to an object.

When this behaviour is found in the domain, the best practice is to declare it as a service.

Service objects do not have an internal state, their purpose is to supply a service to the domain.

Services can be significant and can group related functionality which serves the Entities and the Value Objects.

It's better to declare the service explicitly so a distinction is made in the domain.

Services act as interfaces which provide many operations.

Characteristics of a service :

1. Operation performed by the service refers to a domain concept which does not naturally belong to an Entity or Value Object.
2. The operation performed refers to other objects in the domain.
3. The operation is stateless.

**Modules – Use well defined groups to represent the domain.**

Modules are used as a method of organizing related concepts and tasks to reduce complexity.

It's easier to understand a domain model if you look at the modules it contains and then at the relationships between the modules. After interaction between modules is understood, you can start to work out the details inside a module.

Communicational cohesion is achieved when parts of the module operate on the same data.

Functional cohesion is achieved when parts of the module work together to perform a well-defined task.

Functional cohesion is considered the best type of cohesion.

Modules should have well defined interfaces which are accessed by other modules.

**Aggregates – Group objects together in a larger, managed object.**

It's natural for a domain model to contain many relationships or associations. Associations which aren't essential to the model should be removed.

If many objects satisfy a relationship then it's possible that only one will do if the right constraint is imposed on the relationship.

Transactions are normally used in databases to ensure that associated data is updated when a change is made to a database record. It's better if you can solve the problem related to data integrity in the model itself.

It's also necessary to be able to enforce invariants, rules which have to be maintained when data changes.

Aggregates are a group of associated objects which are considered as one unit with regard to data changes.

Each Aggregate has one root and the root is an Entity and it is the only object accessible from the outside. The root can hold references to any of the Aggregate objects and the other objects can hold references to each other but an outside object can only hold references to the root object. If there are other Entities inside the boundary, the identity of those Entities is local, making only sense inside of the Aggregate.

Since other objects can only hold references to the root they can't directly change other objects in the Aggregate, all they can do is change the root or ask the root to perform some actions.

If the root is deleted and removed from memory all the other objects from the Aggregate will be deleted too. When any change is done to the root affecting other objects in the variant the root will enforce the invariants.

If objects of an Aggregate are stored in a database only the root should be obtainable through queries. The other objects should be obtained through traversal associations.

**Factories – they make new, shiny things.**

Entities and Aggregates can often be large and complex, too complex to create in a constructor of the root entity.

When a client object wants to create another object it calls its constructor and possibly passes in some parameters. But when the object construction is a laborious process, creating the object involves a lot of knowledge about the internal structure of the object, relationships between the objects contained and the rules applied to them. This breaks encapsulation of the domain object and of the Aggregate. If the client belongs to the application layer a part of the domain layer has been moved outside, messing up the entire design.
Creation of an object can be a major operation in itself, but complex assembly operations are not the responsibility of created objects.

A new concept is necessary, one that encapsulates the process of complex object creation. This is called a Factory. Factories encapsulate the knowledge necessary for object creation and are especially useful for creating Aggregates.

The creation process must be atomic, the process must be complete and not leave anything in an undefined state whether this is an attribute of an object or an object within an Aggregate. If the object cannot be created properly then an exception should be raised, invalid values should not be allowed.

The responsibility of creating complex objects or Aggregates is moved to a separate object within the domain design. Provide an interface that encapsulates all complex assembly and doesn't require the client to reference a concrete class of the objects being instantiated.

A Factory Method is an object method which hides the knowledge necessary to create another object. When a client wants to create an object which belongs to an Aggregate, the solution is to add a method to the Aggregate root which takes care of the object created, enforces all invariants and returns a reference to that object or to a copy of it.

Entity Factories and Value Object Factories are different. Values are normally immutable and have everything they need when they are created. Entities can be changed and their invariants need to be respected. Entities also need an identity but Value Objects do not.

Sometimes a factory isn't needed and a constructor will suffice. Use a constructor when :

- The construction is not complicated

- The creation of an object does not involve the creation of others and all attributes needed are passed via the constructor.
- The client is interested in the implementation, perhaps wants to choose the Strategy used.
- The class is the type. There is no hierarchy involved, so no need to choose between a list of concrete implementations.

Factories may need to create new objects or reconstitute objects which previously existed and may have been persisted to a database, both requiring completely different processes. When a new object is created violation of invariants result in exceptions. Objects recreated from a database retrieval somehow need to be repaired so they can function otherwise there is data loss.

**Repositories – Responsible for storing objects**

A constructor or a Factory takes care of object creation and objects are created to be used. In OO languages a reference to an object must be obtained before you can use it. The client either has to create an object or obtain the object from another. This can increase coupling – how do you get a Value Object from an Aggregate? It has to be requested from the Aggregate's root. You need the Aggregate root to get access to the other object in the Aggregate.

Although a lot of objects could be retrieved directly from the database this isn't desired. Databases are part of the infrastructure and the client shouldn't be aware of the details needed to access a database.

Clients need a practical means of acquiring references to domain objects which already exist.
A Repository's purpose is to encapsulate all of the logic required to obtain object references.
The domain objects don't deal with infrastructure to get references to the objects they need, they get them from the Repository.

When an object is created, it may be saved in the Repository and retrieved from the Repository to be used later. If a client requests an object from the Repository and the Repository doesn't have it, it may retrieve it from storage. The Repository acts as a storage place for globally accessible objects.

For each type of object that needs global access, create an object that looks like an in-memory collection of all objects of that type. Access should be provided through a well-known global interface and methods should be provided to add or remove objects encapsulating the actual insertion or deletion of data in the data store. Methods should be provided to select objects based on criteria and return fully instantiated objects or a collection of them where the attributes of the criteria are met. This encapsulates the actual storage and query technology.

Provide repositories only for Aggregate roots than actually need direct access and delegate all object storage and access to Repositories.

While a Repository may contained detailed information used to access the infrastructure its interface should be simple. Methods should be provided to allow the retrieval of objects. Entities can easily be specified by their identity as well as other attributes. The Repository should be able to compare all objects against this criteria and return those that satisfy it. Another method may perform calculations like the number of objects of a certain type.

Selection criteria can be provided as a Specification which allows more complex criteria to be defined.

Repositories should not be confused with Factories. Factories create new objects, Repositories find objects which already exist. When a new object is added to the Repository it should be created first with the Factory and then given to the Repository to store it.

**Refactoring Toward Deeper Insight**

**Continuous Refactoring**

Refactoring is when you change code to improve it but without changing the application's behaviour. It's normal to refactor in very small steps to ensure that functionality isn't changed or bugs are introduced. Automated testing are used to great effect here as they help to ensure that the code is still working as it should.

Refactoring code shouldn't be difficult. If it's proving difficult then there are obviously problems somewhere.

As well as refactoring code, the domain model itself can be refactored. The development of a domain model is normally an iterative process. At the outset the model is quite abstract or basic. Over time it develops as new insights are gained with ongoing communication between the domain experts and developers.

**Key concepts**

Refactorings are normally small but there are times when a few small changes can make a lot of difference – a Breakthrough.

A Breakthrough is normally a change to the way we see the model or the way we think something works. Although it's seen as progress sometimes it brings a lot of work with it.

A Breakthrough comes from making implicit concepts explicit. While some ideas or concepts might make it into the Ubiquitous Language when initially speaking with domain experts, sometimes their significance is missed. Sometimes the concept is missed completely. These are implicit concepts, normally used to help explain other concepts which are already in the model. Recognition that a concept is key to the design indicates that the concept should become explicit, they should have classes and relationships. This could lead to a Breakthrough.

If procedures or processes in the model seem overly complicated or difficult to understand the chances are that there are hidden concepts, something is missing. If a key concept is missing then it's probably that some objects are attributed with behaviour which isn't theirs. On discovering a missing concept make it explicit and refactor the design to make it simpler and more flexible.

Sometimes there appear to be contradictions. These need to be sorted. It's normally down to miscommunication or a lack of description and a bit more digging can sort this out. Try to keep everything clear.

Domain literature can be important. Chances are there's a book on it somewhere which might provide an insight into the common processes of a domain.

Other concepts which can prove to be useful are Constraint, Process and Specification.

A Constraint is used to express an invariant, a rule which must be maintained whenever data changes. The invariant logic is held within a Constraint, for example a check on a capacity within an object. This process may have been implicit, with code held in another method but by moving it to its own method it becomes explicit and easier to see and understand.

Processes usually map to procedures in code which isn't really relevant to object-orientation. We need to choose an object for the process and add a behaviour to it and the best way to do this is with a Service. If there are different ways of performing the process then this can be encapsulated with an algorithm and a Strategy used for the variations.

A Specification tests an object to see if it matches a given criteria. Sometimes these rules are straightforward and can be answered with straightforward Boolean results. Sometimes the rules can be a series of logical operations but putting all of these in the object itself isn't ideal. It's going to make the object large and change it from its original purpose. It's time to refactor the rule into an object of its own, into a Specification which can be kept in the Domain layer.

Each method will play the role of a small test with all methods combined providing an answer to the original one test or question. Keeping all the tests together in one object ensures that the code isn't split across several different objects.

A Specification tests objects to see if they meet a given need, criteria or fit for a purpose. They can also be used as selection criteria when retrieving specific objects from a collection or as a condition during an objects creation.

A single Specification can check if a simple rule is satisfied and a number of these can be combined into a composite Specification to express the complex rule.

**Preserving Model Integrity**

Enterprise projects involving large teams may involve different technologies and resources. These projects should still be based on a domain model though.

It's likely that individual teams will be working on different Modules of the model, coding in parallel. Once teams start plugging in to Modules from another team they may encounter missing functionality they require for their own area of work. In cases like this the easy path is to add the code yourself but this could adversely affect the domain model and break code.

Models must be consistent with invariable terms and no contradictions. Unification is the term given for the internal consistency of a model.

A unified enterprise model without contradictions and overlapping, covering the whole domain may not be achievable. You can't expect the individual teams to be able to coordinate their work to such a level, it would take too much time and resource. Striving to maintain one large unified model isn't going to work.

The solution is to divide the model into several smaller models as long as these develop to a contract that they are strictly bound to. These models should have a clearly defined border with the relationships between the modules defined precisely.

**Bounded Context**

A Bounded Context is not a Module, a Bounded Context will encompass a Module. A model should contain related elements, those which form a natural concept. The model should also be small enough for one team to handle. The context of a model is the conditions applied to ensure that the terms within the model have a specific meaning.

The Model should target a specific area. The boundaries should be set in terms of team organisation, where it will be used within the application and its physical manifestation (code base and database schema). The model needs to be kept within these boundaries.

There's an overhead in having multiple models, extra work and design effort. Objects can't be transferred between models either.

**Continuous Integration**

With a team working on a Bounded Context there's a possibility that the model will fragment. Communication between team members is essential so everyone understands how the elements within the model work and interact. If someone doesn't have the same understanding then they may add code, duplicate code or break existing code.

As new concepts are realised and added to the Domain there needs to be a process of integrating them while making sure they don't have an adverse effect on the rest of the model, specifically the code. Continuous Integration is the process of merging source code together and automatically building and then performing automated tests to indicate problems early so they can be addressed as soon as possible.

**Context Map**

A Context Map provides an overall picture of the model illustrating the different Bounded Contexts within the model itself and the relationships between them. This can be done in a diagram or in a document.

The level of detail may vary, the important thing is that it is shared amongst the project members and everyone understands it.

Separate unified models aren't enough, they're all part of a system and they need to be brought together for the system to work. If the contexts aren't properly defined then there may be overlap and if the relationships between the contexts aren't given then there's a chance the system won't work when it's brought together.

Bounded Contexts should take a name from the Ubiquitous Language. This helps in communication between teams.

Naming conventions should be used indicating the context modules belong to.

**Shared Kernel**

It's possible that teams working independently on closely related applications may make quick progress but ultimately the work they produce doesn't work when brought together. The end result is that more time is spent on getting the finished items to work as a finished item than on the individual pieces themselves.

In these situations it's better to designate a subset of the domain model for the teams to share. This extends to the code and the database design as well as the model design. This shared area (Shared Kernel) is ring-fenced and isn't changed without the other teams consultation.

Reducing duplication is the main goal here while the contexts remain separate. It's essential that if the Shared Kernel is updated that the changes are integrated as soon as possible to ensure there is no knock-on effect with other teams. Similarly, changes to the Shared Kernel, new functionality for example, needs to be communicated to the other teams at the same time.

**Customer-Supplier**

When two different subsystems have a relationship where one depends on the other or rather the results of one are fed in to the other, but they do not have a Shared Kernel, they are said to a have a Customer-Supplier relationship.

A good example is where you have a system recording transactions or purchases and another system which reports on these. The shared element could be the database with changes made by either team severely affecting the other. In this example the reporting system would be the Customer of the Purchase system.

The interface between systems needs to be defined and tests created and exercised to ensure the interface is respected by both sides. The automated tests, agreed between both the Customer and Supplier form a contract between the two systems and are run as part of the Supplier team's Continuous Integration. This means the Supplier side can make the changes they need to without breaking their agreement with the Customer.

**Conformist**

Where a Customer-Supplier relationship exists but the supplier for whatever reason, but perhaps with the best intentions, can't provide what the customer requires the customer has a few options. One is to design their own model, ignoring the suppliers.

The supplier's model may have some value, it's just they can't or won't support the customer team. If the customer decides they are still going to use it then they must try to insulate any changes to the model made by the supplier team. This is done with an Anticorruption Layer.

If the customer wants to use the suppliers model, and conform to it completely then there is another option. With a Shared Kernel the customer team could make changes but with this strategy they can't. What they must do is use the model provided by the supplier (their code) and build on it. If the supplier provided a rich component the customer can build on it to produce their own model which incorporates the supplier one. Another option is to create an adapter for it, if the component has a small interface and this would translate between the supplier and customer models. This approach ensures that the customer team can develop what they need without recourse to the supplier team.

**Anticorruption Layer**

Often we need to create an application which interacts with legacy or external systems. In a lot of these cases the these applications haven't been built using domain modelling techniques and as a result their model is unusable. Even where it is usable the model for the new application is likely to be different.

Two ways the client system could interact with an external system is via network connections, the client adheres to the interface used by the external system and/or the system is to access the same database as the external system.

Both of these approaches have the same problem, the data being moved around is primitive. With a relational database the database does contain primitive data but relationships exist between them and this is very important to note. The new application needs to know what the relationships are so they can be used in the model.

While interaction with the external model can't be ignored we shouldn't allow it to change the client model. To make sure this doesn't happen an Anticorruption Layer is used and this sits between the client model and the external model.

The Anticorruption Layer is part of the client model. The way the layer operates in much the same was as other objects in the client model but the layer talks to the external model using the external model's language.

In effect the layer is a translator sitting between the two domains and their languages with the client model untainted by the external model.

An Anticorruption Layer can be seen as a service from the client model. The Service will translate the external interface into a form the client model can understand.

The service is implemented as a Facade (design pattern) and will probably need an Adaptor which will convert the interface of a class into one the client can understand.

The Anticorruption Layer may contain more than one Service with each Service having its own Facade and in turn each Facade having its own Adapter. Using one Adapter for several Services would produce a Swiss army knife Adapter.

Although the Adapter concerns itself with the behaviour of the external system we still need object and data conversion and this is done with Translators.

**Separate Ways**

Sometimes just is not feasible to spend a lot of time and effort integrating subsystems, compromising on code to make ensure other teams' requirements are met, developing layers for translation between systems etc.

When an enterprise application can be made up of several smaller applications with little or nothing in common from a modelling perspective the Separate Ways pattern can be used.

If the requirements indicate that they can be neatly split into two or more different sets of requirements which have no commonality then we can create Bounded Contexts and model them independently.

The benefit of this approach is that the different technologies can be used for the different implementations though they may share a common thin GUI.

The only thing to make sure of is that there won't be a need to come back to an integrated system as once you've gone Separate Ways this is very difficult.

**Open Host Service**

A translation layer is usually sufficient for two subsystems to integrate, the layer acts as a buffer between the client and external subsystem we want to integrate with.

If the external subsystem is used by several client subsystems and not one then, potentially, translation layers would be needed for all of them with all layers performing the same translation tasks and containing similar code.

If the external subsystem is seen as a provider of services then we can wrap another set of Services around it. All the subsystems can then access these Services without the need for individual translation layers.

Differing needs of each subsystem may make the provision of a coherent set of Services problematic. A protocol can be used to expose the subsystem as a set of services so those who need it can use it. New requirements can see the protocol expand except when there's a single requirement from one team. In this scenario a one-off translator can expand the protocol for the special case so the shared protocol remains simple.

**Distillation**

Distillation is the process of extracting specific substances from a mixture. The process can sometimes provide interesting by-products.

You may end up with a large domain model even after refactoring so a distillation can be used to try to define a Core Domain, the essence of the domain itself. The by-products of this process will be Generic Subdomains.

When working with a large model the essential concepts should be separated from the generic ones.

The idea is to find the Core Domain and make it easily distinguishable from the other supporting model and code, emphasizing the most valuable and specialised concepts.

Establishing the Core Domain will take several steps. The process of refinement and refactorings are needed to establish the Core and the Core is central to the design so it's boundaries need to be marked.

Other elements of the model need to be looked at in relationship with the new core, they may need to be refactored with functionality changed.

Although other elements of the model are essential they may add complexity without capturing or communicating specialised knowledge. These can make the Core Domain harder to distinguish from the rest of the domain.

Generic models of the subdomains should be factored out into separate Modules. These can then take a lower priority than the Core Domain and sometimes these can be provided as off-the-shelf solutions or published models.

Different ways of implementing a Generic Subdomain:

***Of-the-shelf solution*** :Someone else has already done the hard work. OK there's still an element of learning involved and integration may not be as easy but it could save time and money.

***Outsourcing*** :Give the design to another team or company and focus on the Core Domain yourself. The outsourced code still needs to be integrated though.

***Existing Model***: Use an already created model. Books may already provide published analysis patterns which can be used with small changes.

***In-House Implementation*** : This is the best for integration though it's got the on-going maintenance to go with it.



**Share this:**

Twitter Facebook

Loading...

September 18, 2009 in Alt.Net, Programming.

## Related posts

Things to remember….

Where do I start? At the bottom.

Project Structure

← .NET Runtime version 2.0.50727.3074 – Fatal Execution Engine Error opening file in VS

LLBLGen Connecting to wrong database / catalog ~ .config settings →

## Leave a Reply

Enter your comment here...

September 2009

| M | T | W | T | F | S | S |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |

| M | T | W | T | F | S | S |
|---|---|---|---|---|---|---|
| 28 | 29 | 30 | | | | |

« Aug Oct »

## Recent Posts

Xamarin – “Project not Built in Active Configuration”

VS2012 not loading project -Unable to read project file.

Xamarin / Monotouch Scrollview not scrolling.

SQLite.Net MonoAndroid Giving “Could not resolve type” Error

Xamarin / iOS [UITableViewController loadView] loaded the “Controller” nib but didn’t get a UITableView

## Categories

.NET 3.5

About me

Alt.Net

Auditing

Error Logging / Handling

LLBLGEN

Mobile

Motorbikes

MS SQL 2000 – 2008

MVC2

NAnt

Nhibernate

ORMs

Programming

SSRS

Stepping Stones

Travelling

Windows

Xamarin

Xamarin – MonoAndroid

Xamarin – MonoTouch

Xamarin Forms

## Archives

October 2014

June 2014

December 2013

November 2013

October 2013

June 2013

June 2011

December 2010

September 2010

August 2010

February 2010

December 2009

November 2009

October 2009

September 2009

August 2009

July 2009

May 2009

December 2008

November 2008

February 2008