Domain-Driven Design – A *Modern* Approach

What is Domain-Driven Design?

What is the process used in Domain-Driven Design?

What are the building blocks of Domain-Driven Design?

What is a layered architecture and how is it used in Domain-Driven Design?

What are the tools used in Domain-Driven Design?

How do I get started with Domain-Driven Design?
What is Domain-Driven Design?

- Domain-Driven Design is not a new doctrine or methodology. It's a collection of time-tested strategies.
- Domain-Driven Design is about mapping business domain concepts into software artifacts.
- Domain-Driven Design focuses on the problem domain as a whole, which not only includes the data, but also the behavior.

Domain-Driven Design is an approach to software development that suggests:

- The primary focus should be on the domain and domain logic.
- Complex domain designs should be based on a model.
- Initiate a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.

Core definitions of Domain-Driven Design:

- Domain: A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.
- Model: A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.
- Ubiquitous Language: A language structured around the domain model and used by all team members to connect all the activities of the team with the software.
- Model-Driven Design: Provides a set of guidelines for the structuring of specifications, which are expressed as models.
- Test-Driven Development: An evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring.
- Behavior-Driven Development: A specialized version of test-driven development which focuses on behavioral specification of software units.
What is the process used in Domain-Driven Design?

1. Building Domain Knowledge.
   - It is important for the software specialists (analysts, architects, designers and developers) to learn as much as possible about the domain from the experts.
   - Communication is a two-way street, from the domain experts to the business analyst, software architect system designer and further to the developers.
   - The software specialists and the domain experts are creating the model of the domain together, and the model is the place where those two areas of expertise meet. This might seem like a very time consuming process, and it is.
   - Analyze data and information from all existing sources. Understand how the existing data will fit into the new model.

2. Creating a common language.
   - A core principle of Domain-Driven Design is to use a language based on the model. Since the model is the common ground, the place where the software meets the domain, it is appropriate to use it as the building ground for this language.
   - Ubiquitous Language: A language structured around the domain model and used by all team members to connect all the activities of the team with the software.
   - Use the model as the backbone of a language. Request that the team use the language consistently in all communications, and also in the code. While sharing knowledge and hammering out the model, the team uses speech, writing and diagrams.
   - Make sure this language appears consistently in all the communication forms used by the team.

3. Exploring/refining Models in a creative collaboration of domain experts and software specialists and can also be called Model-Driven Design.
   - It is fundamentally important to create a model which is deeply rooted in the domain, and should reflect the essential concepts of the domain with great accuracy.
   - It is also highly recommended for the developers to implement the main concepts of the model in the code. Code becomes an expression of the model, so a change to the code may be a change to the model.
   - Demand a single model that serves both purposes well, in addition to supporting a fluent Ubiquitous Language.
4. Architecting and designing a software solution to solve the business problem.

- A software architect should have a very good abstract understanding on how processes work and how they are designed and optimized.
- It is important to divide an application in separate layers, and establish rules of interactions between the layers.
- If the code is not clearly separated into layers, it will soon become so entangled that it becomes very difficult to manage changes.
- However, don’t add all layers upfront, we wait until it is a problem.
- And, don’t add additional layers just to have layers.
- Loose coupling is great but not 100%. Take a balanced approach between loose coupling and high cohesion.

5. Testing all aspects of integration and behavior.

- Understand all aspects of integration and behavior.
- Begin by writing a unit test for the functionality that you intend to write.
What are the building blocks of Model-Driven Design?

- **Layered Architecture**: A technique for separating the concerns of a software system, isolating a domain layer, among other things.
- **Entity**: An object that is not defined by its attributes, but rather by a thread of continuity and its identity. An entity is something that exists by itself. Implementing entities means creating identity.
- **Value Object**: An object that contains attributes or describes some characteristic of an entity but has no conceptual identity. They should be treated as immutable and be kept thin and simple.
- **Service**: An operation offered as an interface that stands alone in the model, with no encapsulated state and which provides operations.
- **Aggregate**: A cluster of associated objects that are treated as a unit for the purpose of data changes. External references are restricted to one member of the AGGREGATE, designated as the root. A set of consistency rules applies within the AGGREGATE’S boundaries.
- **Factory**: A mechanism for encapsulating complex creation logic and abstracting the type of a created object for the sake of a client.
- **Repository**: A mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects.
What is a layered architecture and how is it used in Domain-Driven Design?

Traditional Architecture: Partition a complex program into LAYERS. Develop a design within each LAYER that is cohesive and that depends only on the layers below. Follow standard architectural patterns to provide loose coupling to the layers above.

- User Interface (Presentation Layer): Responsible for presenting information to the user and interpreting user commands.
- Application Layer: This layer coordinates the application activity. It doesn't contain any business logic. It does not hold the state of business objects, but it can hold the state of an application task's progress.
- Domain Layer: This layer contains information about the business domain. The state of business objects is held here. Persistence of the business objects and possibly their state is delegated to the infrastructure layer.
- Infrastructure Layer: This layer acts as a supporting library for all the other layers. It provides communication between layers, implements persistence for business objects, contains supporting libraries for the user interface layer, etc.

- This is the basic architecture most frequently used.
- Each subsequent layer depends on the layers beneath it, and then every layer normally will depend on some common infrastructure and utility services.
- The big drawback to this top-down layered architecture is the coupling that it creates.
- Each layer is coupled to the layers below it, and each layer is often coupled to various infrastructure concerns.
- This architecture creates unnecessary coupling.
Onion Architecture (coined by Jeffrey Palermo in 2008): The main premise is that it controls coupling. The fundamental rule is that all code can depend on layers more central, but code cannot depend on layers further out from the core. In other words, all coupling is toward the center. This architecture is unashamedly biased toward object-oriented programming, and it puts objects before all others.

- In the very center we see the Domain Model, which represents the state and behavior combination that models truth for the organization.
- Around the Domain Model are other layers with more behavior.
- The number of layers in the application core will vary.
- But remember that the Domain Model is the very center, and since all coupling is toward the center, the **Domain Model is only coupled to itself**.

Key tenets of Onion Architecture:
- The application is built around an independent object model.
- Inner layers define interfaces. Outer layers implement interfaces.
- Direction of coupling is toward the center.
- All application core code can be compiled and run separate from infrastructure.

Starting point for layering:

Love it!
Model, code, etc... How do they fit together?

- Problem and chosen solution
- Model
- UmlAsASketch
- Ubiquitous Language
- Code
- UI
What are some tools used in Domain-Driven Design?

Design Principles, Patterns and Methodologies:
- Domain-Driven Design
- Model-Driven Design
- Test-Driven Development
- Behavior-Driven Development
- Object Oriented Programming (OOP)
- SOLID Principles
- Dependency Inversion
- DRY – Don’t Repeat Yourself
- CQRS

Communication Tools:
- Whiteboards
- Unified Modeling Language (UML)
- Entity-relationship diagrams (ERDs)
- Sequencing diagrams
- Skills Matrix
- Kanban Board
- Burn-Down Charts

Development and Implementation Tools:
- ASP.NET MVC C#
- Object-Relational-Mapper (ORM) – Entity Framework
- Dependency Injection Frameworks (DI) – Ninject
- Unit Testing - NUnit
- SQL Server
- SQL Azure
- Table Storage
- Service Bus
How do I get started with Domain-Driven Design?

If you have made it this far you are well under way!

- DDD is not a new doctrine or methodology. It's a collection of time-tested strategies.
- When you get ready to practice, try adapting those philosophies, techniques, and patterns that make the most sense in your situation.
- Some elements of DDD apply more universally than others.
- Become proficient with Object-Oriented Programming (OOP)
- Learn about Object-Orientated Design and the SOLID principles
- Understanding the reason for your core domain's existence by uncovering and using the Ubiquitous Language and identifying the contexts in which we're modeling are way more important than nailing that perfectly opaque and one-size-fits-all repository.

1. Change your mindset.
2. Focus on the Domain.
3. Follow the ideas and patterns presented.
4. Do you own research.

DDD Application Life Cycle

Thank you!
Resources

Mini Book: Domain-Driven Design Quickly – Abel Avram & Floyd Marinescu
http://www.infoq.com/minibooks/domain-driven-design-quickly

Book: Domain-Driven Design: Tackling Complexity in the Heart of Software – Eric Evans
http://www.amazon.com/exec/obidos/ASIN/0321125215/domainlanguag-20

An Introduction to Domain-Driven Design

Getting Started with Domain-Driven Design
http://refcardz.dzone.com/refcardz/getting-started-domain-driven

Domain-Driven Design and Development In Practice
http://www.infoq.com/articles/ddd-in-practice

Domain-Driven Design Community
http://domaindrivendesign.org/

Glossary of Domain-Driven Design Terms
http://dddcommunity.org/resources/ddd_terms/

The Onion Architecture
http://jeffreypalermo.com/blog/the-onion-architecture-part-1/

Wikipedia
http://en.wikipedia.org/wiki/Model_driven_design
http://en.wikipedia.org/wiki/Value_object

Developing Core Business Applications with Domain-Driven Design (DDD) and Microsoft .NET

Up and Down the Ladder of Abstraction
http://worrydream.com/LadderOfAbstraction/
Appendix 1 – DDD In Action Images

Discovery Documents:

Skills Matrix:
Kanban Board:

Burn-Down Charts:
Unit Tests with NUnit:

Layered Architecture 1:
Layered Architecture 2:

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>.nuget</td>
</tr>
<tr>
<td>CQRS.Framework</td>
</tr>
<tr>
<td>FL.DOS.Corporation.DataMigration</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Azure</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Azure.DataBus</td>
</tr>
<tr>
<td>FL.DOS.Corporations.BatchWorker.Azure</td>
</tr>
<tr>
<td>FL.DOS.Corporations.BatchWorker.CommandService</td>
</tr>
<tr>
<td>FL.DOS.Corporations.BulkCorrections.RegisteredAgent</td>
</tr>
<tr>
<td>FL.DOS.Corporations.CommandHandlers</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Commands</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Data.Azure</td>
</tr>
<tr>
<td>FL.DOS.Corporations.DataMigration.Azure</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Domain</td>
</tr>
<tr>
<td>FL.DOS.Corporations.EventHandler</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Events</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Interfaces</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Model.Diagrams</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Model.Shared</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Reconciliation.Voucher</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Search.Azure</td>
</tr>
<tr>
<td>FL.DOS.Corporations.ServiceBus.Host</td>
</tr>
<tr>
<td>FL.DOS.Corporations.TestReports</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Tests.Behaviour</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Tests.Integration</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Tests.Migration</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Validation</td>
</tr>
<tr>
<td>FL.DOS.Corporations.ViewModels</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Web</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Web.Helpers</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Web.Internal</td>
</tr>
<tr>
<td>FL.DOS.Corporations.Web.Search</td>
</tr>
</tbody>
</table>