Domain-Driven Design

using Naked Objects

The Pragmatic Programmers

Domain-Driven Design Using Naked Objects

Dan Haywood

Edited by Susannah Davidson Pfalzer

Printed in the United States of America.

ISBN-10: 1-934356-44-1

ISBN-13: 978-1-934356-44-9

Printed on acid-free paper.

P1.0 printing, December 2009

Version: 2010-1-16

Chapter 1

Getting Started

To stop himself from procrastinating in his work, the Greek orator Demosthenes would shave off half his beard. Too embarrassed to go outside and with nothing else to do, his work got done.

We could learn a lesson or two from old Demosthenes. After all, we forever seem to be taking an old concept and inventing a new technology around it (always remembering to invent a new acronym, of course)—anything, it would seem, instead of getting down to the real work of solving business problems.

Domain-driven design (hereafter DDD) puts the emphasis elsewhere, “tackling complexity in the heart of software.” And Naked Objects—an open source Java framework—helps you build your business applications with ease. No beard shaving necessary, indeed.

In this chapter, we’re going to briefly describe the key ideas underlying DDD, identify some of the challenges of applying these ideas, and see for ourselves how Naked Objects makes our task that much easier.

1.1 Understanding Domain-Driven Design

There’s no doubt that we developers love the challenge of understanding and deploying complex technologies. But understanding the nuances and subtleties of the business domain itself is just as great a challenge, perhaps more so. If we devoted our efforts to understanding and addressing those subtleties, we could build better, cleaner, and more maintainable software that did a better job for our stakeholders. And there’s no doubt that our stakeholders would thank us for it.

A couple of years back Eric Evans wrote his book *Domain-Driven Design* [Eva03], which is well on its way to becoming a seminal work. In fact, most if not all of the ideas in Evans' book have been expressed before, but what he did was pull those ideas together to show how predominantly object-oriented techniques can be used to develop rich, deep, insightful, and ultimately useful business applications.

So, let's start off by reviewing the essential ideas of DDD.

1.2 The Essentials of DDD

There are two central ideas at the heart of domain-driven design. The *ubiquitous language* is about getting the whole team (both domain experts and developers) to communicate more transparently using a domain model. Meanwhile, *model-driven design* is about capturing that model in a very straightforward manner in code. Let's look at each in turn.

Creating a Ubiquitous Language

It's no secret that the IT industry is plagued by project failures. Too often systems take longer than intended to implement, and when finally implemented, they don't address the real requirements anyway.

Over the years we in IT have tried various approaches to address this failing. Using waterfall methodologies, we've asked for requirements to be fully and precisely written down before starting on anything else. Or, using agile methodologies, we've realized that requirements are likely to change anyway and have sought to deliver systems incrementally using feedback loops to refine the implementation.

But let's not get distracted talking about methodologies. At the end of the day what really matters is communication between the domain experts (that is, the business) who need the system and the techies actually implementing it. If the two don't have and cannot evolve a shared understanding of what is required, then the chance of delivering a useful system will be next to nothing.

Bridging this gap is traditionally what business analysts are for; they act as interpreters between the domain experts and the developers. However, this still means there are two (or more) languages in use, making it difficult to verify that the system being built is correct. If the analyst mistranslates a requirement, then neither the domain expert

DDD in context...

Ubiquitous Language

Build a common language between the domain experts and developers by using the concepts of the domain model as the primary means of communication. Use the terms in speech, in diagrams, in writing, and when presenting.

If an idea cannot be expressed using this set of concepts, then go back and extend the model. Look for and remove ambiguities and inconsistencies.

nor the application developer will discover this until (at best) the application is first demonstrated or (much worse) an end user sounds the alarm once the application has been deployed into production.

Rather than trying to translate between a business language and a technical language, with DDD we aim to have the business and developers using the same terms for the same concepts in order to create a single *domain model*. This domain model identifies the relevant concepts of the domain, how they relate, and ultimately where the responsibilities are. This single domain model provides the vocabulary for the *ubiquitous language* for our system.[1]

Creating a *ubiquitous language* calls upon everyone involved in the system's development to express what they are doing through the vocabulary provided by the model. If this can't be done, then our model is incomplete. Finding the missing words deepens our understanding of the domain being modeled.

This might sound like nothing more than me insisting that the developers shouldn't use jargon when talking to the business. Well, that's true enough, but it's not a one-way street. A *ubiquitous language* demands that the developers work hard to understand the problem domain, but it also demands that the business works hard in being *precise* in its naming and descriptions of those concepts. After all, ultimately the developers will have to express those concepts in a computer programming language.

1. In Extreme Programming, there is a similar idea called a *system of names*. But *ubiquitous language* is much more evocative.

Also, although here I'm talking about the "domain experts" as being a homogeneous group of people, often they may come from different branches of the business. Even if we weren't building a computer system, there's a lot of value in helping the domain experts standardize their own terminology. Is the marketing department's "prospect" the same as sales' "customer," and is that the same as an after-sales "contract"?

The need for precision within the *ubiquitous language* also helps us scope the system. Most business processes evolve piecemeal and are often quite ill-defined. If the domain experts have a very good idea of what the business process should be, then that's a good candidate for automation, that is, including it in the scope of the system. But if the domain experts find it hard to agree, then it's probably best to leave it out. After all, human beings are rather more capable of dealing with fuzzy situations than computers.

So, if the development team (business and developers together) continually searches to build their *ubiquitous language*, then the domain model naturally becomes richer as the nuances of the domain are uncovered. At the same time, the knowledge of the business domain experts also deepens as edge conditions and contradictions that have previously been overlooked are explored.

We use the *ubiquitous language* to build up a domain model. But what do we do *with* that model? The answer to that is the second of our central ideas.

Model-Driven Design

Of the various methodologies that the IT industry has tried, many advocate the production of separate analysis models and implementation models. A recent example is that of the OMG's Model-Driven Architecture (MDA) initiative, with its platform-independent model (the PIM) and a platform-specific model (the PSM).

Bah and humbug! If we use our *ubiquitous language* just to build up a high-level analysis model, then we will re-create the communication divide. The domain experts and business analysts will look only to the analysis model, and the developers will look only to the implementation model. Unless the mapping between the two is completely mechanical, inevitably the two will diverge.

DDD in context. . .

Model-Driven Design

There must be a straightforward and very literal way to represent the domain model in terms of software. The model should balance these two requirements: form the *ubiquitous language* of the development team and be representable in code.

Changing the code means changing the model; refining the model requires a change to the code.

What do we mean by *model* anyway? For some, the term will bring to mind UML class or sequence diagrams and the like. But this isn't a model; it's a visual *representation* of some aspect of a model. No, a domain model is a group of related concepts, identifying them, naming them, and defining how they relate. What is in the model depends on what our objective is. We're not looking to simply model everything that's out there in the real world. Instead, we want to take a relevant abstraction or simplification of it and then make it do something useful for us. Oft quoted and still true is that a model is neither right nor wrong, just more or less useful.

For our *ubiquitous language* to have value, the domain model that encodes it must have a straightforward, literal representation to the design of the software, specifically to the implementation. Our software's design should be driven by this model; we should have a *model-driven design*.

Here also the word *design* might mislead; some might again be thinking of design documents and design diagrams. But by *design* we mean a way of organizing the domain concepts, which in turn leads to the way in which we organize their representation in code.

Luckily, using *object-oriented* (OO) languages such as Java, this is relatively easy to do; OO is based on a modeling paradigm anyway. We can express domain concepts using classes and interfaces, and we can express the relationships between those concepts using associations.

So far so good. Or maybe, so far so much motherhood and apple pie. Understanding the DDD concepts isn't the same as being able to apply them, and some of the DDD ideas can be difficult to put into practice.

What this book is about is how Naked Objects eases that path by applying these central ideas of DDD in a very concrete way. So, now would be a good time to see how.

1.3 Introducing Naked Objects

Naked Objects is both an architectural pattern and a software framework. The pattern was originally conceived and articulated by Richard Pawson as a means of engaging business stakeholders and experts in developing more expressive domain-driven applications. Richard discusses this in more detail in the foreword.

The framework, then, is an implementation of the pattern to help you rapidly prototype, develop, and deploy domain-driven applications:

- *Rapid prototyping* comes from the fact that you can develop an application without spending any time writing user interface code or persistence code. This creates a very tight feedback loop with your domain experts.
- The *development support* comes from the close integration with developer tools such as *Eclipse* (for coding), *FitNesse* (for testing), *Maven* (for building and packaging), and *Hudson* (for continuous integration).
- The *deployment support* comes from Naked Objects' pluggable architecture allowing different viewers, persistence mechanisms, and security. In fact, the domain model has no runtime dependencies on the framework, so you can deploy your application on any Java-based enterprise architecture with any UI you want.

For more on the original philosophy that drove Naked Objects' development, see Richard Pawson and Robert Matthews' book, *Naked Objects* [PM02], and Richard's later PhD thesis.[2]

I could talk at length in a highly theoretical fashion about Naked Objects and how it relates to DDD for the next thirty pages, but what we're going to do instead is see Naked Objects in action.

2. http://www.nakedobjects.org/downloads/Pawson%20thesis.pdf

Joe Asks...

How Does Naked Objects Compare to Other Frameworks?

Many other frameworks promise rapid application development and provide automatically generated user interfaces, so how do they compare to Naked Objects?

Some of most significant are *Rails* (for the Ruby programming language), *Grails* (Groovy), and *Spring Roo* (Java with AspectJ).[*] These frameworks all use the classic *model-view-controller* (MVC) pattern for web applications, with scaffolding, code-generation, and/or metaprogramming tools for the controllers and views, as well as convention over configuration to define how these components interact. The views provided out of the box by these frameworks tend to be simple CRUD-style interfaces. More sophisticated behavior is accomplished by customizing the generated controllers.

For many developers, the most obvious difference of Naked Objects is its deliberate lack of an explicit controller layer; non-CRUD behavior is automatically made available in its generic object-oriented UIs. More sophisticated UIs can be built either by skinning Naked Objects (see Chapter 15, *Integrating with Web Frameworks*, on page 283) or by using a newer viewer that supports easy customization (see Chapter 18, *Deploying the Full Runtime*, on page 347).

Like all of these frameworks, Naked Objects can expose domain objects as a RESTful web service. However, it has some other tricks you may not find in a typical MVC framework: it supports client-server (rich-client) deployments as well as on the Web; it supports non-RDBMS as well as RDBMS object stores, with an in-memory object store for rapid prototyping; it supports domain-driven concepts such as values, repositories, and domain services; it supports agile scenario testing using FitNesse; and it puts the domain metamodel at the center, allowing the programming model to be redefined.

[*]. The frameworks mentioned here are hosted at http://rubyonrails.org/, http://www.grails.org, and http://www.springsource.org/roo.