WHY YOU SHOULD DO DDD

Actually, I've already given you some pretty good reasons why DDD makes so much practical sense. At the risk of breaking the DRY principle ("Don't repeat yourself"), I reiterate them here and also add to the earlier reasons. Does anyone hear an echo?

• Put domain experts and developers on a level playing field, which produces software that makes perfect sense to the business, not just the coders. This doesn't mean merely tolerating the opposite group. It means becoming one cohesive, tight-knit team.

• That "makes sense to the business" thing means investing in the business by making software that is as close as possible to what the business leaders and experts would create if they were the coders.

• You can actually teach the business more about itself. No domain expert, no C-level manager, no one, ever knows every single thing about the business. It's a constant discovery process that becomes more insightful over time. With DDD, everybody learns because everybody contributes to discovery discussions.

• Centralizing knowledge is key, because with that the business is capable of ensuring that understanding the software is not locked in "tribal knowledge," available only to a select few, who are usually only the developers.

• There are zero translations between the domain experts, the software developers, and the software. That doesn't mean maybe some few translations. It means zero translations because your team develops a common, shared language that everyone on the team speaks.

• The design is the code, and the code is the design. The design is how it works. Knowing the best code design comes through quick experimental models using an agile discovery process.

• DDD provides sound software development techniques that address both strategic and tactical design. Strategic design helps us understand what are the most important software investments to make, what existing software assets to leverage in order to get there fastest and safest, and who must be involved. Tactical design helps us craft the single elegant model of a solution using time-tested, proven software building blocks.

Like any good, high-yielding investment, DDD has some up-front cost of time and effort for the team. Considering the typical challenges encountered by every software development effort will

reinforce the need to invest in a sound software development approach.

For now think of a Bounded Context as a conceptual boundary around a whole application or finite system. The reason for this boundary is to highlight that every use of a given domain term, phrase, or sentence—the Ubiquitous Language—inside the boundary has a specific contextual meaning. Any use of the term outside that boundary could, and probably does, mean something different. Chapter 2 explains Bounded Context in depth.

Ubiquitous Language

The Ubiquitous Language is a shared team language. It's shared by domain experts and developers alike. In fact, it's shared by everyone on the project team. No matter your role on the team, since you are on the team you use the Ubiquitous Language of the project.

So, You Think You Know What a Ubiquitous Language Is

Obviously it's the language of the business.

Well, no.

Surely it must be adopting industry standard terminology.

No, not really.

Clearly it's the lingo used by the domain experts.

Sorry, but no.

The Ubiquitous Language is a shared language developed by the team—a team composed of both domain experts and software developers.

That's it. Now you've got it!

Naturally, the domain experts have a heavy influence on the Language because they know that part of the business best and may be influenced by industry standards. However, the Language is *more centered on how the business itself thinks and operates*. Also, many times two or more domain experts disagree on concepts and terms, and they are actually wrong about some because they haven't thought of every case before. So, as the experts and developers work together to craft a model of the domain, they use discussion with both consensus and compromise to achieve the very *best Language for the project*. The team never compromises on the quality of the Language, just on the best concepts, terms, and meanings. Initial consensus is not the end, however. The Language grows and changes over time as tiny and large breakthroughs are achieved, much like any other living language.

This is no gimmick to get developers to be on the same page as domain experts. It's not just a bunch of business jargon being forced on developers. It's a real language that is created by the whole team—domain experts, developers, business analysts, everyone

involved in producing the system. The Language may start out with terms that are the natural lingo of the domain experts, but it isn't limited to that because the Language must grow over time. Suffice it to say that when multiple domain experts are involved in creating the Language, they often disagree ever so slightly on the terms and meanings of what they thought were already ubiquitous.

In Table 1.4, we not only model the administration of flu vaccines in code, but the team must also speak the Language openly. When the team discusses this aspect of the model, they literally speak phrases such as "Nurses administer flu vaccines to patients in standard doses."

Table 1.4. Analyzing the Best Model for the Business

Which is better for the business? *Though the second and third statements are similar, how should the code be designed*	
Possible Viewpoints	**Resulting Code**

"Who cares? Just code it up."
Um, not even close.
	`patient.setShotType(ShotTypes.TYPE_FLU);`
	`patient.setDose(dose);`
	`patient.setNurse(nurse);`

"We give flu shots to patients." Better, but misses some important concepts.	`patient.giveFluShot();`
"Nurses administer flu vaccines to patients in standard doses." This seems like what we'd like to run with at this time, at least until we learn more.	`Vaccine vaccine = vaccines.standardAdultFl` `nurse.administerFluVaccine(patient, vaccin`

There will be some haggling and wrangling over the Language that exists in the minds of experts and what evolves from there. It's all part of the natural progression of developing the best Language that will matter a lot for a long time. This happens through open discussion, looking at existing documents, business tribal knowledge that finally surfaces, as well as referencing standards, dictionaries, and thesauruses. There's also a point reached where we come to terms with the fact that some words and phrases just don't aptly fit the business context as well as we once thought, and we realize that others fit it much better.

So how do you capture this all-important Ubiquitous Language? Here are some ways that work as experimentation leads to advancement:

• Draw pictures of the physical and conceptual domain and label them with names and actions. These drawings are mostly informal but may contain some aspects of formal software modeling. Even if your team does some formal modeling with Unified Modeling Language (UML), you want to avoid any kind of ceremony that will bog down discussions and stifle the creativity of the ultimate Language being sought.

• Create a glossary of terms with simple definitions. List alternative terms, including the ones that show promise and the ones that didn't work, and why. As you include definitions, you cannot help but develop reusable phrases for the Language because you are forced to write in the Language of the domain.

• If you don't like the idea of a glossary, still capture some kind of documentation that includes the informal drawings of important software concepts. Again, the goal here is to force additional Language terms and phrases to surface.

• Since only one or a few team members may capture the glossary or other written documents, circle back with the rest of the team to review the resulting phrases. You won't always, if ever, agree on all the captured linguistics, so be agile and ready to edit heavily.

Those are some ideal first steps to coining a Ubiquitous Language that fits your specific domain. However, this is absolutely not the model that you are developing. It's only the genesis of the Ubiquitous Language that will very soon be expressed in your system's source code. We are talking Java, or C#, or Scala, or some other programming language of choice. These drawings and documents also don't address that the Ubiquitous Language will continue to expand and morph over time. The artifacts that originally led us down an inspiring path to developing a useful Ubiquitous Language that was just right for our specialized domain will very likely be rendered obsolete over time. *That's why in the end it is team speech and the model in the code that are the most enduring and the only guaranteed current denotations of the Ubiquitous Language.*

Since team speech and the code will be the lasting expression of the Ubiquitous Language, be prepared to abandon the drawings, glossary, and other documentation that will be difficult to keep up-to-date with the spoken Ubiquitous Language and source code as they are rapidly enhanced. This is not a requirement of using DDD,

but it is pragmatic because it becomes impractical to keep all the documentation in sync with the system.

With this knowledge we can redesign the `saveCustomer()` example. What if we chose to make `Customer` reflect each of the possible business goals that it must support?

Click here to view code image

```
public interface Customer {
   public void changePersonalName(
      String firstName, String lastName);
   public void postalAddress(PostalAddress postalAddress);
   public void relocateTo(PostalAddress changedPostalAddress);
   public void changeHomeTelephone(Telephone telephone);
   public void disconnectHomeTelephone();
   public void changeMobileTelephone(Telephone telephone);
   public void disconnectMobileTelephone();
   public void primaryEmailAddress(EmailAddress emailAddress);
   public void secondaryEmailAddress(EmailAddress emailAddress);
}
```

We can argue that this is not the best model for a `Customer`, but when implementing DDD, questioning the design is expected. As a team we are free to haggle over what is the best model and settle only after we've discovered the Ubiquitous Language that is agreed upon. Still, the preceding interface does explicitly reflect the various business goals that a `Customer` must support, even if the Language could be improved by refinements again and again.

It's important to understand too that the Application Service would also be refactored to reflect the explicit intentions of the business goals at hand. Each Application Service method would be modified to deal with a single use case flow or user story:

Click here to view code image

```
@Transactional
public void changeCustomerPersonalName(
   String customerId,
   String customerFirstName,
   String customerLastName) {

   Customer customer = customerRepository.customerOfId(customerId);

   if (customer == null) {
```

```
        throw new IllegalStateException("Customer does not exist.");
    }

    customer.changePersonalName(customerFirstName,
customerLastName);
}
```

This is different from the original example because in that code a single method was used to deal with many different use case flows or user stories. In the new example we have limited a single Application Service method to deal with changing the personal name of the `Customer`, and nothing more. Thus, when using DDD, it is our job to refine Application Services accordingly. This implies that the user interface likewise reflects a narrower user goal, which may have previously been true. Now, however, this specific Application Service method doesn't require its client to pass ten nulls following the first- and last-name parameters.

Doesn't this new design put your mind at ease? You can read the code and easily comprehend it. You can also test it and confirm that it does exactly what it is meant to do, and that it doesn't do anything that it shouldn't.

Thus, the Ubiquitous Language is a team pattern used to capture the concepts and terms of a specific core business domain in the software model itself. The software model incorporates the nouns, adjectives, verbs, and richer expressions formally formulated and spoken by the close-knit team. Both the software and the tests that verify the model's adherence to the tenets of the domain capture and adhere to this Language, the same one spoken by the team.

Ubiquitous, but Not Universal

Some further clarification about the reach of a Ubiquitous Language is in order. There are a few basic concepts that we need to keep carefully in mind:

• *Ubiquitous* means "pervasive," or "found everywhere," as *spoken among the team and expressed by the single domain model* that the team develops.

• The use of the word *ubiquitous* is not an attempt to describe some kind of enterprise-wide, company-wide, or worldwide, universal domain language.

• There is one Ubiquitous Language per Bounded Context.

• Bounded Contexts are relatively small, smaller than we might at first imagine. A Bounded Context is large enough only to capture the

complete Ubiquitous Language of the isolated business domain, and no larger.

• The Language is ubiquitous only within the team that is working on the project that develops in an isolated Bounded Context.

• On a single project that develops a single Bounded Context, there are always one or more additional isolated Bounded Contexts with which it integrates using **Context Maps (3)**. Each of the multiple Bounded Contexts that integrate has its own Ubiquitous Language, even though some terms of each may overlap.

• If you try to apply a single Ubiquitous Language to an entire enterprise, or worse, universally among many enterprises, you will fail.

When you begin a new project in which you are properly using DDD, identify the isolated Bounded Context that is being developed. This places an explicit boundary around your domain model. Discuss, research, conceptualize, develop, and speak the Ubiquitous Language of the isolated domain model within the explicit Bounded Context. Reject all concepts that are not part of the agreed-upon Ubiquitous Language of your isolated Context.

1. The organization gains a useful model of its domain.

2. A refined, precise definition and understanding of the business is developed.

3. Domain experts contribute to software design.

4. A better user experience is gained.

5. Clean boundaries are placed around pure models.

6. Enterprise architecture is better organized.

7. Agile, iterative, continuous modeling is used.

8. New tools, both strategic and tactical, are employed.

The word *design* can evoke negative thoughts in the minds of business management. However, DDD is not a heavyweight, high-ceremony design and development process. DDD is not about drawing diagrams. It is about carefully refining the mental model of domain experts into a useful model for the business. It is not about creating a real-world model, as in trying to mimic reality.

The team's efforts follow an agile approach, which is iterative and incremental. Any agile process that the team feels comfortable with can be used successfully in a DDD project. The model that is

produced is the working software. It is refined continuously until it is no longer needed by the business.

A Domain, in the broad sense, is what an organization does and the world it does it in. Businesses identify a market and sell products and services. Each kind of organization has its own unique realm of know-how and way of doing things. That realm of understanding and its methods for carrying out its operations is its Domain. When you develop software for an organization, you are working in its Domain. It should be pretty obvious to you what your Domain is. You work in it.

Don't forget, a Bounded Context is an explicit boundary within which a domain model exists. The domain model expresses a Ubiquitous Language as a software model. The boundary is created because each of the model's concepts inside, with its properties and operations, has a special meaning. If you are a member of such a modeling team, you'd know exactly the meaning of each of the concepts in your Context.

> Bounded Context Is Explicit and Linguistic
>
> A Bounded Context is an explicit boundary within which a domain model exists. Inside the boundary all terms and phrases of the Ubiquitous Language have specific meaning, and the model reflects the Language with exactness.

It is often the case that in two explicitly different models, objects with the same or similar names have different meanings. When an explicit boundary is placed around each of the two models individually, the meaning of each concept in each Context is certain. Thus, a Bounded Context is principally a *linguistic boundary*. You should use these points of reasoning as a touchstone to determine if you are correctly using Bounded Contexts.

Some projects fall into the trap of attempting to create an all-inclusive model, one where the goal is to get the entire organization to agree on concepts with names that have only one global meaning. Approaching a modeling effort in this way is a pitfall. First, it will be nearly impossible to establish agreement among all stakeholders that all concepts have a single, pure, and distinct global meaning. Some organizations are so large and complex that you'd never be

able to get all stakeholders together, let alone establish total meaningful agreement among them. Even if you are working in a smaller company with relatively few stakeholders, establishing an enduring definition of a single global concept is still unlikely. Thus, the best position to take is to embrace the fact that differences always exist and apply Bounded Context to separately delineate each domain model where differences are explicit and well understood.