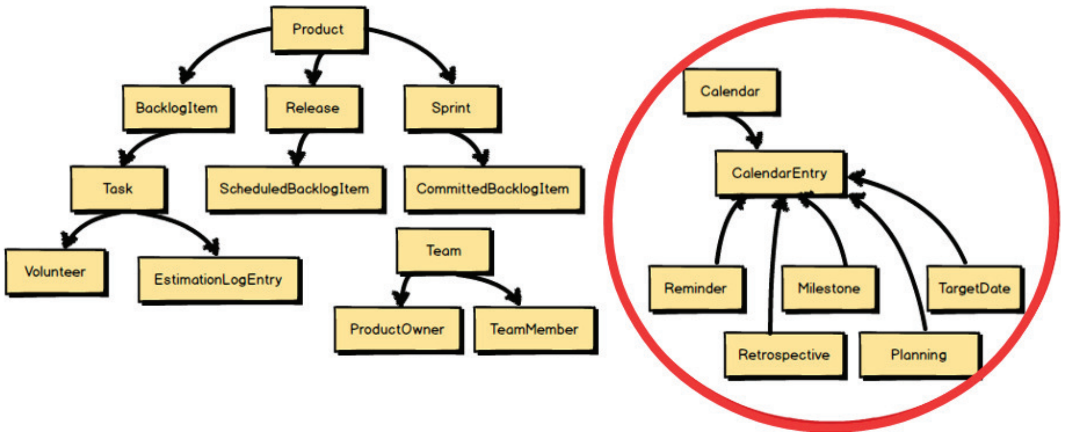
DOMAIN-DRIVEN DESIGN DISTILLED

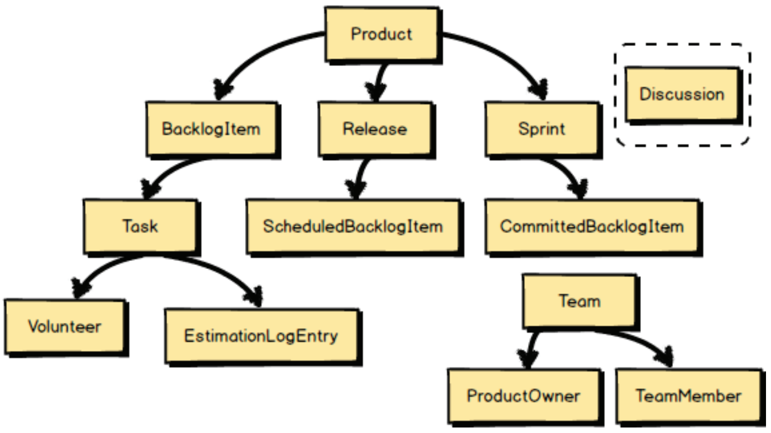
VAUGHN VERNON

Domain-Driven Design Distilled

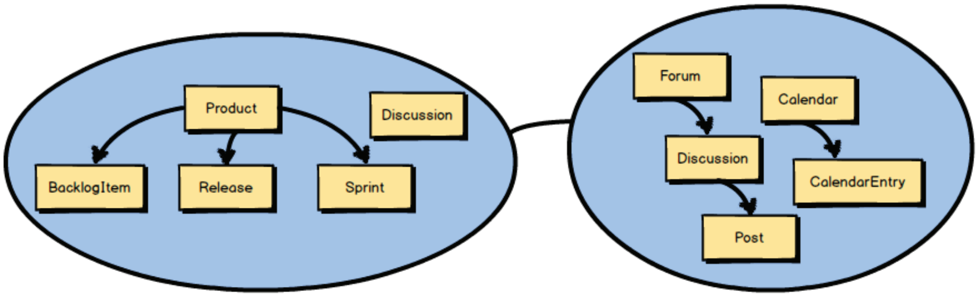
TeamMembers to work on Tasks. In Scrum this is known as a Volunteer. So, the Volunteer concept is in context and was included in the language of the core model.



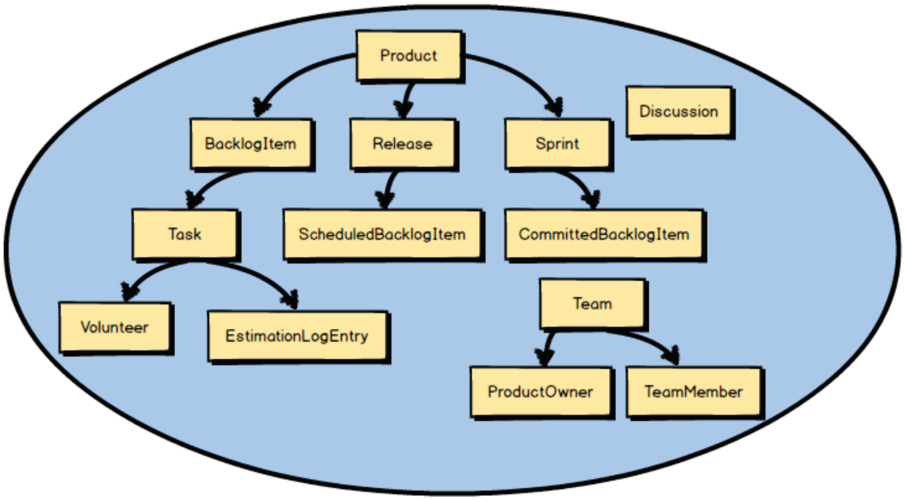
Even though calendar-based Milestones, Retrospectives, and the like are in context, the team would prefer to save those modeling efforts for a later sprint. They are in context, but for now they are out of scope.



Finally, the modelers want to make sure that they account for the fact that threaded Discussions will be part of the core model. So they model a `Discussion`. This means that `Discussion` is part of the team's *Ubiquitous Language*, and thus inside the *Bounded Context*.

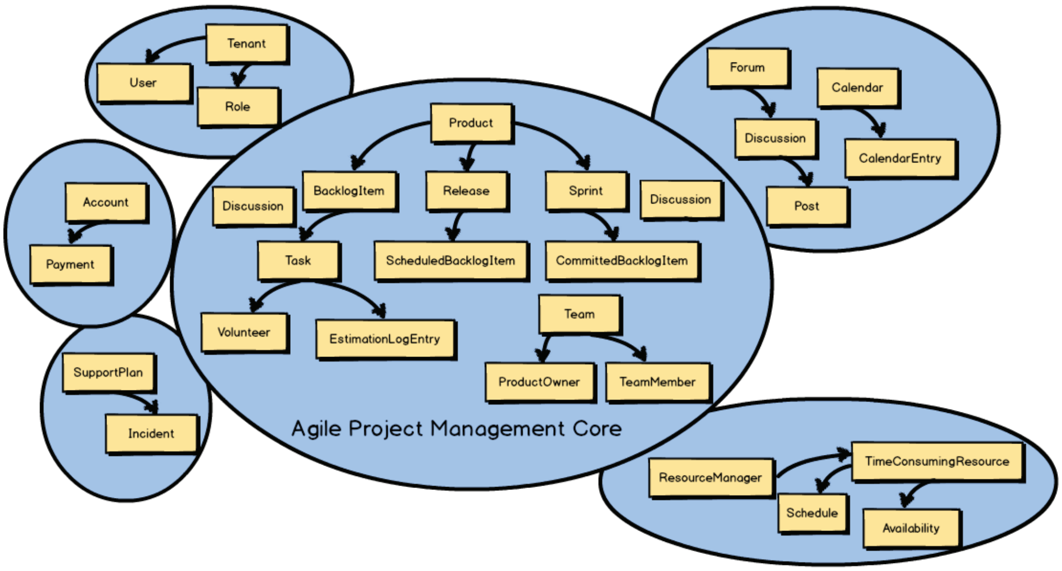


These linguistic challenges have resulted in a much cleaner and clearer model of the *Ubiquitous Language*. Yet how will the Scrum model fulfill needed *Discussions*? It would certainly require a lot of ancillary software component support to make it work, so it seems inappropriate to model it inside our Scrum *Bounded Context*. In fact, the full *Collaboration* suite is out of context. The *Discussion* will be supported by integrating with another *Bounded Context*—the *Collaboration Context*.



After that walk-through, we're left with a much smaller actual *Core Domain*. Of course the *Core Domain* will grow. We already know that

`Planning`, `Retrospectives`, `Milestones`, and related calendar-based models must be developed in time. Still, the model will grow only as new concepts adhere to the *Ubiquitous Language* of Scrum.



And what about all the other modeling concepts that have been removed from the *Core Domain*? It's quite possible that several of the other concepts, if not all, will be composed into their own respective *Bounded Contexts*, each adhering to its own *Ubiquitous Language*. Later you will see how we integrate with them using *Context Mapping*.

Developing a Ubiquitous Language

So how do you actually go about developing a *Ubiquitous Language* within your team as you put into practice one of the chief tools provided by DDD? Is your *Ubiquitous Language* formed from a set of well-known nouns? Nouns are important, but often software developers put too much emphasis on the nouns within a domain model, forgetting that spoken language is composed of far more than nouns alone. True, we have mainly focused on nouns within our previous sample *Bounded Contexts* to this point, but that's because we were interested in another aspect of DDD, that of constraining a *Core Domain* down to essential model elements.

Accelerate Your Discovery

You may want to try a few *Event Storming* sessions as you work on your scenarios. These can help you to quickly understand which scenarios you should be working on, and how they should be prioritized. Likewise, developing concrete scenarios will give you a better idea of the direction that you should take in your *Event Storming* sessions. They are two tools that work well together. I explain the use of *Event Storming* in Chapter 7, "Acceleration and Management Tools."

Don't limit your *Core Domain* to nouns alone. Rather, consider expressing your *Core Domain* as a set of concrete scenarios about what the domain model is supposed to do. When I say "scenarios" I don't mean use cases or user stories, such as is common in software projects. I literally mean scenarios in terms of how the domain model should work—what the various components do. This can be accomplished in the most thorough way only by collaborating as a team of both *Domain Experts* and developers.

Here's an example of a scenario that fits with the *Ubiquitous Language* of Scrum:

> *Allow each backlog item to be committed to a sprint. The backlog item may be committed only if it is already scheduled for release. If it is already committed to a different sprint, it must be uncommitted first. When the commit completes, notify interested parties.*

Notice that this is not just a scenario about how humans use Scrum on a project. We are not talking about human procedures. Rather, this scenario is a description of how the very real software model components are used to support the management of a Scrum-based project.

The previous scenario is not a perfectly stated one, and a perk of using DDD is that we are constantly on the lookout for ways to improve the model. Yet this is a decent start. We hear nouns spoken, but our scenario doesn't limit us to nouns. We also hear verbs and adverbs, and other kinds of grammar. You also hear that there are constraints—conditions that must be met before the scenario can be completed to its successful end. The most important benefit and empowering feature is that you can actually have conversations about how the domain model works—its design.

We can even draw simple pictures and diagrams. It's all about doing whatever is needed to communicate well on the team. One word of warning is appropriate here. Be careful about the time spent in your domain-modeling efforts when it comes to keeping documents with written scenarios and drawings and diagrams up-to-date over the long haul. Those things are not the domain model. Rather, they are just tools to help you develop a domain model. In the end the code is the model and the model is the code. Ceremony is for distinguished observances, like weddings, not domain models. This doesn't mean that you forgo any efforts to freshen scenarios, but only do so as long as it is helpful rather than burdensome.

What would you do to improve a part of the *Ubiquitous Language* in our previous example? Think about it for just a minute. What's missing? Before too long you probably wish for an understanding of *who* does the committing of backlog items to a sprint. Let's add the *who* and see what happens:

The product owner commits each backlog item to a sprint . . .

You will find in many cases that you should name each persona involved in the scenario and give some distinguishing attribute to other concepts such as to the backlog item and sprint. This will help to make your scenario more concrete and less like a set of statements about acceptance criteria. Still, in this particular case there isn't a strong reason to name the product owner or further describe the backlog item and sprint involved. In this case all product owners, backlog items, and sprints will work the same way whether or not they have a concrete persona or identity. In cases where giving names or other distinguishing identities to concepts in the scenario helps, use them:

The product owner Isabel commits the View User Profile backlog item to the Deliver User Profiles sprint . . .

Now let's pause for a moment. It's not that the product owner is the sole individual responsible for deciding that a backlog item will be committed to a sprint. Scrum teams wouldn't like that very much, because they would be committed to delivering software within some time frame that they had no say in determining. Still, for our software model it may be most practical for a single person to have the responsibility to carry out this particular action on the model. So in this case we have stated that it's the product owner role that does this. Even so, the nature of Scrum teams forces the question "Is there anything that must be done by the remainder of the team to enable the product owner to perform the commitment?"

Do you see what has happened? By challenging the current model with the *who* question, we have been led to an opportunity for deeper insight into the model. Perhaps we should require at least some team consensus that a backlog item can be committed before actually allowing the product owner to carry out the commit operation. This could lead to the following refined scenario:

The product owner commits a backlog item to a sprint. The backlog item may be committed only if it is already scheduled for release, and if a quorum of team members have approved commitment . . .

OK, now we have a refined *Ubiquitous Language,* because we have identified a new model concept called a *quorum*. We decided that there must be a *quorum* of team members who agree that a backlog item should be committed, and there must be a way for them to *approve* commitment. This has now introduced a new modeling concept and some idea that the user interface will have to facilitate these team interactions. Do you see the innovation unfolding?

There is another *who* missing from the model. Which one? Our opening scenario concluded:

When the commit completes, notify interested parties.

Who or what are the interested parties? This question and challenge further lead to modeling insights. Who needs to know when a backlog item has been committed to a sprint? Actually one important model