1. **Introduction**
2. **Ubiquitous Language**

**The Cost of Translation**

**TODO: Graphic Here**

According to Eric Evans in *Domain Driven Design,* a Ubiquitous Language is “..a language structured around the domain model and used by all team members to connect all the activities of the team with the software.”

The Ubiquitous Language should drive every piece of communication between a development team and the business domain, from spoken and written communication to models, diagrams and the code itself. Nothing should be allowed to bypass the requirement that the shared and codified language of the domain permeate through all aspects of a software project.

In keeping with Eric Evans definition*,* we believe that the development of a Ubiquitous Language is essential to success in software development. Taking the idea a step further, we also believe that an architect is an ideal steward and driver for the adoption of a Ubiquitous Language on a development team.

Consider the following dialogue between a domain expert and a development team:

**Expert**: We need to make sure that our support staff can change the rules that we use to create policies for customers.

**Architect**: ok, so we’ll use a Strategy pattern and make that config-driven…

**Developer**: we could just use IoC, build strategies for each implementation and let the users swap out implementations whenever they need to change them.

**Architect**: Yeah, that’s an option too. We’ll figure it out offline.

**Expert**: (confused) So will the support staff be able to change those?

**Architect**: Sure, they’ll just change config and it will just work.

**Developer**: Or swap out an implementation for the container in config.

**Expert**: What’s IoC?

**Architect**: well…

Now consider the following alternate take on the same conversation:

**Expert**: We need to make sure that our support staff can change the rules that we use to create policies for customers.

**Architect**: okay, so the POLICY BUILDER will need to be able to support the addition and/or replacement of POLICY RULES by a POLICY ANALYST?

**Expert**: Yeah, exactly. We call it the Policy Wizard, but I like your term better.

**Architect**: Can we agree to globally replace Policy Wizard with POLICY BUILDER in all of our discussions and usage then? We want to make sure that everyone understands these terms and uses them properly.

**Expert**: Sure. If you can help me write up an email, we can inform people of the change today.

**Developer**: Great. So what kinds of things do POLICY ANALYSTS change in a POLICY RULE?

**Expert**: Effective dates, amount limits, minor details.

**Developer**: So only attributes about the policy. Is there any swapping in and out of policies?

**Expert**: No. We don’t do that often. When we do, it requires executive approval and process changes.

**Architect**: Okay, so POLICY RULE changes performed by a POLICY ANALYST will be minor, otherwise we’ll need to perform system modifications as a part of the process changes.

**Expert**: Makes sense.

**The Importance of Language**

The only real difference between the two conversations above was the use of language to describe the domain. And yet, the use of language in these examples took each conversation in a completely different direction. In the first conversation, the architect and developer muddled the conversation with their domain expert by introducing technical detail that was essentially irrelevant to their conversation. If a strategy pattern is to be used to solve a business problem, it’s important to discuss how such a pattern should be implemented in one’s framework of choice. But it’s not useful to do so in a conversation that is designed to scope the domain and the software that will be created to add value to that domain.

What’s more, the architect and developer in this first example spent far too little time understanding the expert’s domain. The first mention of rules and runtime modifications of the system resulted in an immediate jump to patterns and framework details. A further elucidation of the requirements from the domain expert may have resulted in the revelation that the requirement was to allow analysts to change data attributes on entities at runtime, not the complex rule engine that the team jumped to at the first mention of “rules.”

On the other hand, the domain is also not well-served if the developer and architect sit idly by and allow the domain expert to define all project knowledge in the terms of the domain. Most domains suffer from inconsistencies and ambiguities that domain experts may not be aware of or that they allow to exist for various reasons. The jargon that invariably grows up around a domain is usually a mix of well-defined terminology, inexact analogies, muddled and overlapping ideas and contentious concepts that never reach resolution. Whereas the technological domain is precise but mostly irrelevant to the business domain, the business domain is imprecise and lacking the stability that a model and software effort requires to be successful.

The adoption of a Ubiquitous Language is not about embracing either technological or domain terminology. Rather, it is about embracing both, utilizing the domain to describe the model (as in, the conceptual representation of your software) and leveraging the model to bring accuracy and stability to the domain. While such a combination does require learning for both domain experts and the development team, the end result is a stable and rich model that accurately represents the needs of the business.

We understand that the use of a ubiquitous language, in addition to placing a new requirement upon all team members, also introduces the need for a Language Steward, someone who, while not the owner of the language, is responsible for ensuring that the language is used, maintained and developed. This person is the Webster’s of the development effort, and should be someone who deeply understands both worlds and the value that each party brings to bear in a software project.

We believe that architects are specially equipped for this role since most architects already live with one foot in both worlds. We have found from our own experiences that translation between the domain and the development team often becomes an unofficial job responsibility. Since the architect is already being asked to live in and understand both the domain and technology concerns, it makes sense that she would be the ideal candidate to reconcile the domain and technology and begin moving the entire team towards a Ubiquitous Language. When the architect moves from translator to steward, a Ubiquitous Language has an increased likelihood of success.

It’s important to note here that learning a language will always yield more accurate communication than relying upon translation. This is just as true in the realm of business and technical jargon as it is in the realm of spoken languages. Communication is the art of using language to convey meaning consistently. Jargon is the practice of using certain words and phrases in a way that assumes a known context, and thus, can serve as a shortcut in communication. But when domain experts and development teams get around the table without a Ubiquitous Language, the jargon each brings to the table necessitates translation and guarantees that confusion will propagate into software.

1. **Relevant Models**

**What’s a model?**

**Essential Complexity**

**Strategic Design**

1. **Emergent Architecture**

**Encapsulate, Don’t Coddle**

Too much of architecture and design seems to focus on specifying as much as possible up front before the development team is fully engaged on a project. While we understand that the intentions for such a motivation are pure—the powers that be typically want to reduce uncertainty and trashing before too many expensive resources are involved—the effect of such a motivation often tends to be seen by the development team as an attempt to reduce their role on a project to that of an automaton churning out predefined modules with little to no creative thought.

We believe that over-specifying your expectations to a developer is a form of coddling. Over-specification creates inflexible boundaries and results in brittle software, something you’re likely tasked with preventing as an architect. In the end your development team will feel both constrained and insulted by your effort.

On the other hand, a blank slate is no better. It’s dangerous to underspecify as system as well, and one with no boundaries and no architecture to speak of is destined to suffer from the implementation of sub-optimal decisions by domain experts and developers.

Balance is achieved through engagement and encapsulation. First, we believe that, in nearly all cases, architects should be active members of a development team, not only creating architecture models, diagrams and deliverables, but also writing code. In *Domain Driven Design*, Eric Evans postulates that working code is the best representation of a domain model that is relevant to both the domain and the development team, and we agree. An architect should be driving the development of the model through documentation AND through code.

By being engaged and active with development teams, an architect is less likely to make decisions that would be seen as coddling to the rest of the team. Not only will the architect learn to accurately value the contributions of the rest of the development team, but he will be forced to live with his own dictates, and will thus be less-likely to over-constrain himself, and thus, the team.

Where constrains are needed, architects should use the concepts behind encapsulation in OO design as a guide for specification. Simply put, architects should focus their efforts in the domain by clearly defining what a given capability provides and not how that capability should be implemented down to the precise details. Gartner uses the term Emergent Architecture to describe this practice, and summarizes Emergent Architecture with the phrase “Architect the lines and not the boxes.” Assuming that lines are interactions between software components (or objects) and the boxes are the components (or objects), we believe that this statement is inaccurate, but moving in the right direction. We see emergent architecture as defining the lines, the shape and boundaries of the boxes, and then iterating with the development team to fill—and possibly redefine—the boxes.

When you use specifications and architecture as a replacement for engagement with a development team, you’re coddling. On the other hand, when you’re focused on creating a loose boundary that is tightly coupled to the domain model, you’re encapsulating; allowing the system to emerge and evolve and, most importantly to the architect, remain relevant to both the domain and the development team.

**Design with Evolution in Mind**

**TODO: Graphic Here**

“Design for change” is a mantra we’ve often heard as architects and developers, but what does it mean? Does it mean that we should assume that every line of code in our system is subject to change and, as such, should be written to enable someone else to come along later and make that change easily? If it does, a team can quickly find itself in a death spiral of over engineering based on speculative requirements instead of actual ones.

To us, “Design for change” means understanding which coarse-grained aspects of a system are likely to change and then isolating those which are likely to change from those that aren’t. It also means understanding which aspects of the system change more often, or at different rates, and keeping those aspects separate from one another.

For the architect, Designing for change implies selecting an architecture or design that complements this ordering and isolation. Layered Architectures are typically leveraged to achieve the kind of ordering and isolation described here, but they tend to enable, if not encourage, the kind of accidental coupling that works against original purpose. As an alternative, consider the Onion Architecture approach. Originally described by Jeffrey Palermo, the Onion Architecture approach focuses on isolating layers through interfaces, leveraging IoC to minimize coupling and, first and foremost, making the Domain Model the star of the show.

For Domain-Driven Design to truly be effective, the Domain Model should be both core to the application and isolated from all concerns not relevant to the business domain. In practical terms, this means that orthogonal concerns like logging, security and Data Access should be implemented elsewhere, leaving the domain free to do what it does best: express the fundamental value of a business application through clean models that are accessible to developers and domain experts.

Once you’ve achieved the kind of isolation that patterns like the Onion Architecture provide, you have a structure that enables independent layers to evolve and change at different rates with little friction between and internal to those layers. Your Domain Model, which you should expect to change more than any other part of the system, can evolve even as infrastructure concerns like Data Access are implemented and tested.

In some ways, the kind of independence described here is exactly what we take the phrase “Architect the not the boxes” to intend. By leveraging clean interfaces, IoC and a rich domain model, the architect can maximize his or her value to the domain and development teams by delivering an architecture that is well-bounded, flexible and change-absorbent without being too prescriptive.

1. **Conclusion**

*Brandon Satrom is the Chief Architect with Thought Ascent, Inc., a Microsoft Gold Partner. Paul Rayner is an Architect and Principal for Virtual Genius, LLC. They both think that Architecture and Agile mix well.*