1. **Introduction**

Too many systems seem to become legacy upon release, while some never even have a chance to move into production before they are undermined by the calcification of unmet expectations and mismatched domain needs. Regardless of the design effort early in the lifecycle, neglecting the domain model and producing inflexible design results in the increasing irrelevance of the initial architecture of a system. The accidental complexity of that system rises and communication between developers and customers deteriorates. Changes and new features become more difficult to accommodate as the richness and value of the system's essential complexity is eroded. Sustainable and successful software development is all about managing complexity and enabling change, and successful architects create designs that clearly address both.

Architects, domain experts and developers collaborate to mitigate complexity through strategic modeling and design. This requires a focus on the core domain and the continuous application of germane design patterns. Ongoing effort should be expended on defining and refining the domain model through the establishment and exercise of a language that everyone shares. The development of this ubiquitous language, along with the use of domain-driven design techniques, enables business problems and their solutions to be expressed through rich domain models that are both meaningful to business experts and executable by the development team.

Keeping our architectures relevant also means enabling change. As architecture is allowed to emerge, evolve, and mature, it becomes a true reflection of the deep understanding of both domain experts and developers. Architects who expect their initial design to evolve, and who design with evolution in mind, create architectures that deliver a strong competitive advantage to the business.

1. **Ubiquitous Language**

**The Cost of Translation**

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 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.”

***TODO: Graphic Here? – Diagram illustrating cost of translation (Venn diagram with two options)***

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 Architect as Language Steward**

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?**

According to the Encarta World English Dictionary, a model is “a simplified version of something complex used in analyzing and solving problems or making predictions.” A model is a representation, a simplification and an interpretation of reality. A model airplane represents the shape and form of an actual airplane, yet it is simplified—it is often vastly smaller and cannot fly—and only copies those aspects of the original that the designer found important to imitate—it has doors and wheels, but no engine or complex machinery.

Beyond being a simplified representation of a thing, a model must have a purpose, that of “solving problems or making predictions.” When used for scientific or engineering purposes, a model exists to enable to model-makers to express something nebulous and complex as something that can be understood, communicated and manipulated. Thus, a model, while simplified, must remain connected to the thing it represents in order to be useful in solving problems. A model car created to illustrate artistic flair and irrelevant detail is useless to the people who rely on that model.

A domain model is no different. It’s a widely accepted fact in software that domain models are intended to represent a business domain. What seems to be less accepted is the idea that the model must be, first and foremost, tightly coupled to the business domain, and not an expression of technical jargon or framework limitations. The establishment of a Ubiquitous Language aids the architect in emphasizing a domain model that expresses that language over a model that is filled with inexact terminology or obfuscating technical detail.

It’s important to note here that a model is not any one artifact. It is not merely a UML diagram or a SQL Schema. Collectively, a model is the idea that all of these artifacts are meant to convey. This means that diagrams, documents and code all form the Domain Model for your project. Keep in mind, however, that in order for a model to be valuable, it must be relevant to the domain and development teams. A ubiquitous language is perfectly described in a document or wiki for team consumption, but there is no substitute, in terms of relevance, for working code when it comes to describing the entities and interactions of a domain model.

***TODO – Table? – List of artifacts that can be used to express a model***

A model expressed in code provides relevance to architecture, but it also aids greatly in minimizing complexity often found in software and the domain.

**Essential Complexity**

The model’s most important job is dealing with complexity, both in the domain and in software itself. To remain relevant, a domain model must deal three different types of complexity:

1. Non-Essential Complexity – Complexity that is embedded in the business domain, but which is not core to the problem being addresses, or which is a commodity that can be brought into the system. This should be purged from the model;
2. Accidental Complexity – non-essential complexity introduced by designs, frameworks and code that bleed into the domain model and create coupling between concerns. This should be prevented through isolation.
3. Essential Complexity – This is core to the success of the business domain—a strategic advantage even—and should be a primary focus of the model;

Eric Evans proposes strategies for dealing with each of these types of complexity in Part IV of *Domain-Driven Design*. Evans summarizes those strategies under the heading of “Strategic Design,” and these practices are meant to be leveraged as a domain model grows and evolves over time.

We’ll take Evans’s advice a step further and say that an architect should hold the role of strategic designer on a team. While the management of complexity in the software is a role for all members of the team, it should be a success metric for the architect. By assuming the mantle of strategic designer, the architect ensures that her architecture enables the right kind of complexity, while walling-off the wrong kind. She also enables that architecture to evolve and mature as the domain and software mature.

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—this action 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 and localized decisions by domain experts and developers.

Balance between over and under specification can be 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 with a development team, an architect is less likely to make decisions that would be perceived as coddling. 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 encapsulation 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 the term with the phrase “Architect the lines and not the boxes.” Assuming that lines are interactions between software components and the boxes are the components, we believe that this statement is inaccurate, albeit moving in the right direction. We see emergent architecture as more than just defining the lines; it is also about defining the shape and boundaries of the boxes, and then iterating with the development team to fill—and possibly redefine—the boxes.

When you use architecture specifications and models 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**

“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 at different rates and keeping those aspects separate from one another.

***TODO: Graphic Here? – Rates of Change***

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, most important, making the Domain Model the star of the show.

***TODO: Graphic Here? – Layered and Onion***

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 alike.

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 at a faster rate 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 lines, 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**

To remain valued and valuable, the architecture of a system must be relevant—connected to both the domain and the development team. An architect can establish relevance through the stewardship of a Ubiquitous Language, which eliminates the need for translation and fosters collaboration between domain experts and developers. That relevance grows as the domain model is established as the core of the software effort and the architect insures that the model both expresses the domain and remains free from orthogonal concerns. Finally, the architect solidifies relevance by creating an architecture that emerges and evolves with the deeper understanding of domain experts and developers. All of these steps require an architect who is deeply engaged with a team, and fully invested in its success. The result, we believe, is the simplest yet most powerful result of all: software that solves a core problem and delights users.

*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.*