**Keeping Architectures Relevant: Using Domain-Driven Design and Emergent Architecture to Manage Complexity and Enable Change**

**Paul Rayner and Brandon Satrom**

**I. Summary**

Sustainable and successful software development is all about managing complexity and enabling change, and successful software architects create designs that clearly address both concerns. For businesses with complex domains, designing with evolution in mind and utilizing techniques from Domain-Driven Design will result in systems with architectures that deliver a strong, sustainable competitive advantage.

**II. Introduction**

Too many systems become legacy upon release, while some never 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 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 thataddress 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 appropriate 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. Combining a strong domain model focus with continuous attention to growing the software architecture can be a potent way to enable change while managing complexity. This does not guarantee success, but architects who distill the problem domain into a rich model, incorporate it deeply into the system and design with evolution in mind are on the path to creating architectures that can deliver strong, sustainable competitive advantage to the business.

**III. Ubiquitous Language**

**The (Hidden) Cost of Translation**

                According to Eric Evans*,* 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.[[1]](#footnote-1)” Ubiquitous Language should drive every piece of communication between a development team and the business domain, from spoken and written communication to models, system documentation, automated tests, 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. 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**: 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 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? We want to make sure that everyone understands these terms and uses them consistently.

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

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

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

**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.

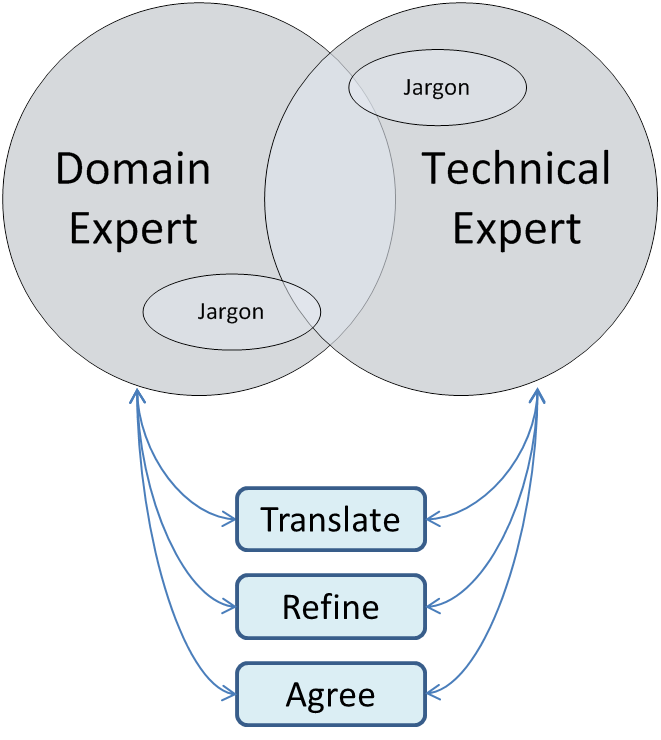
**Architect**: Okay, so POLICY RULE changes performed by a POLICY ANALYST will be minor, otherwise we’ll need to perform system changes as a part of those process changes.

**Expert**: Makes sense.

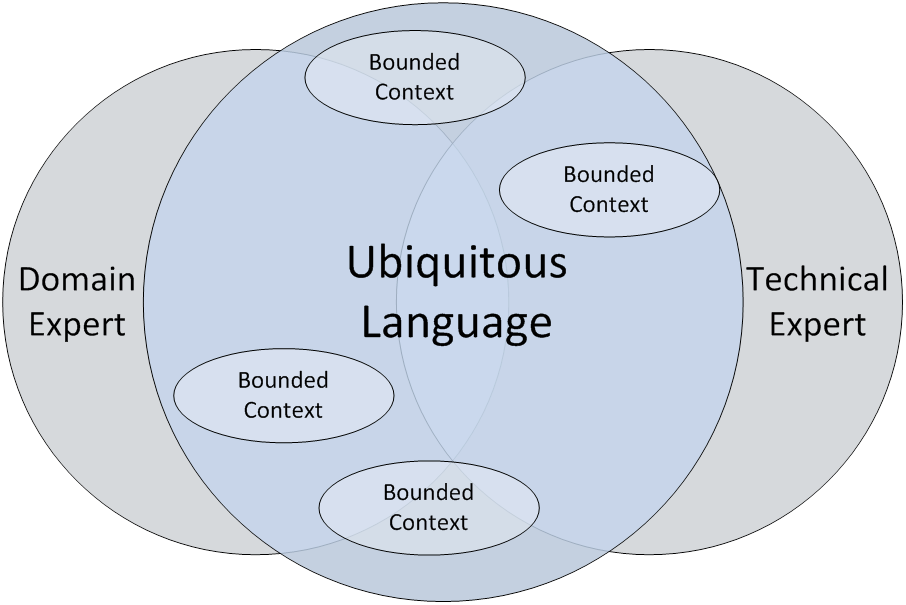
In the first conversation, the architect and developer muddled the conversation with their domain expert by introducing technical detail that was essentially irrelevant to the problem domain. If a strategy pattern is to be used to solve a business problem, it’s important to discuss how such a pattern 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 software being created to add value to that domain. In the first example, the architect and developer spent far too little time understanding the expert’s domain. The mention of rules and runtime modifications of the system resulted in an immediate jump to patterns and framework details.

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 terms of the business domain. Business domains typically suffer from inconsistencies and ambiguities that domain experts either are not aware of, or allow to exist for various reasons. The jargon that invariably grows up around a business domain is usually a mix of well-defined terminology, inexact analogies, muddled and overlapping ideas, and contentious concepts that never reach resolution. Whereas the technical jargon is precise but mostly irrelevant to the business domain, the business domain is imprecise and lacking the stability that a model and software require to be successful. As illustrated in Figure 1, the typical tactic of translation adds overhead and process without enhancing the long-term understanding of either party. Figure 2 illustrates an alternative model, one where the knowledge of both the business and technical domains are combined, along with new information, to create a richer, shared understanding of the domain.

**Figure 1: The Cost of Translation**



**Figure 2: Creating a new, Ubiquitous Language**



Creating a robust Ubiquitous Language requires time and effort, but leads to far more accurate communication than translation alone. 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 and clearly. 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. So, while deep domain knowledge and developing a Ubiquitous Language take time to acquire and require collaborative learning for both domain experts and the development team, the end result is a stable and rich model that more accurately represents the core needs of the business and supports future growth.

Architects typically work across a variety of business contexts in a company, acquiring significant domain knowledge in the process, and are responsible for understanding both domain and technology concerns. Translation between domain experts and development teams often becomes an unofficial job responsibility. But translation is not enough. The adoption of a Ubiquitous Language by everyone involved in developing the software involves a commitment to take the business domain seriously and focus on incorporating it into both conversation and code as much as possible. This means utilizing the domain to develop the model in code, and leveraging the model to bring accuracy, clarity and stability to the domain and Ubiquitous Language. Many architects are also in leadership roles with respect to the development team, and thus in an ideal position to champion this effort. By moving from translator to advocate of a Ubiquitous Language, the architect facilitates more effective communication between all parties and enables software that can better express a deep domain model.

**IV. Relevant Models**

**What’s a model?**

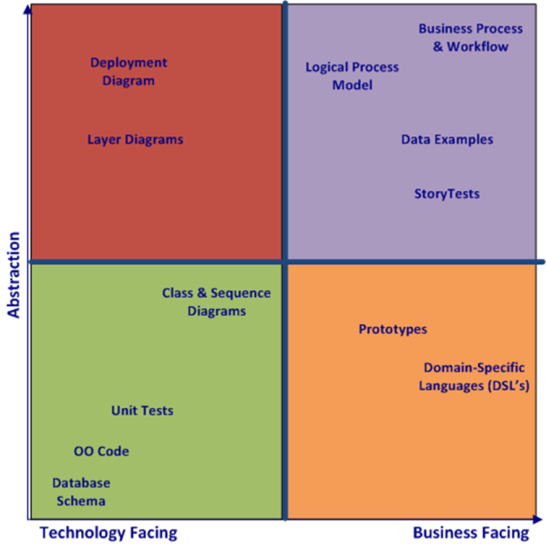
A model can be defined as “a simplified version of something complex used in analyzing and solving problems or making predictions.[[2]](#footnote-2)” It is a representation, a simplification and an interpretation of reality. For example, a model airplane represents the shape and form of an actual airplane, yet it is simplified—it is 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.[[3]](#footnote-3)” When used for scientific or engineering purposes, a model exists to enable the model-makers to express something nebulous and complex in a manner that can be understood, communicated and manipulated. Thus, a model, while simplified, must remain meaningfully connected to the thing it represents in order to be useful in solving problems.

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 these models must, first and foremost, clearly express the business domain, and not be an expression of technical jargon or framework limitations. The establishment of a Ubiquitous Language enables emphasizing a domain model that represents the domain accurately and deeply, instead of one filled with inexact terminology or obfuscating technical detail.

It’s important to note here that a model is not merely a UML diagram or a Database Schema. As illustrated in Figure 3, Diagrams, documents, wikis, automated tests, domain-specific languages, and especially code all instantiate aspects of the Domain Model for a system; each provides clarity to the business or technology side of the domain with varying levels of abstraction. But for such a domain model to be valuable, it must be relevant both to domain experts and development teams. There is no substitute for ensuring that the production code and associated automated test code accurately reflects the domain when it comes to describing the entities and interactions of a domain model. Incorporating Storytesting into the development process is one particularly effective means for saturating feature discussions and executable documentation with the Ubiquitous Language, which naturally leads into incorporating it into the subsequent automated tests and the production code[[4]](#footnote-4). "Writing concrete examples as tests explores ways in which to use and evolve the Ubiquitous Language for expressing business objects, constraints and rules.[[5]](#footnote-5)"

**Figure 3. Model Artifact Matrix**[[6]](#footnote-6)



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

**Managing 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 address three different types of complexity:

1)      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;

2)      Orthogonal Complexity – Complexity that is embedded in the business domain, but which is not core to the problem being addressed, or which is a commodity that can be brought into the system. This should be purged from the domain model as it is distilled over time;

3)      Accidental Complexity – Complexity introduced by designs, frameworks and code that bleed into the domain model and create coupling between concerns. This bleeding should be prevented through isolation of the infrastructure from the domain model.

Part IV of *Domain-Driven Design[[7]](#footnote-7)*is a collection of principles and strategies targeted at dealing with domain complexity. Evans summarizes those under the heading of “Strategic Design,” and they are meant to be leveraged as a system grows and evolves over time. The architect should hold the role of strategic designer on a team, and while the management of complexity in the software is the responsibility of all team members, it should be a success criterion for the architect. By assuming responsibility for driving strategic design, the architect ensures that her architecture enables essential complexity, while walling-off the accidental and orthogonal complexities that tend to creep into systems over time. She also enables that architecture to evolve and mature as the system changes to accommodate future shifts in business needs.

**V.                  Emergent Architecture**

**Encapsulate, Don’t Coddle**

Many architects prefer to detail architecture up front, before the development team is fully engaged on a project. While the typical intent is to reduce uncertainty and thrashing before too many costly resources are involved, this action is often 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 too much up-front architecture is a form of over-specification, and that over-specifying design details to developers is a form of coddling. Over-specification of internal component details creates inflexible boundaries and results in brittle software, something you’re likely tasked with preventing as an architect. The development team will be inappropriately constrained, and perhaps even insulted, by this approach.

A blank slate is no better, though. It’s also dangerous to under-specify a system. With no boundaries and no intentional architecture, a design is destined to suffer from the implementation of sub-optimal and localized decisions by both domain experts and developers. Keeping the development team all moving in the same direction as they seek to distill the model and code itself is not easy. One way to connect the domain model to business drivers and ensure the team is aware of the value of what they are delivering is for the architect to lead in the creation and communication of a domain vision statement which elucidates the core domain and its value[[8]](#footnote-8).

The balance between over and under-specification can be achieved through *engagement*and *encapsulation*. Architects should spend at least part of their time as active members of a development team, not only creating architecture models, diagrams and deliverables but also writing code, since the code *is* the design[[9]](#footnote-9). An architect should be involved in the development of the model through conversation, modeling, documentation, prototyping and coding[[10]](#footnote-10). By being actively engaged with a development team, the 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 keep his skills current, live with his own dictates, and avoid over-constraining himself or the team.

Where constraints 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.  The architect should collaborate with the development team to define and code higher-level contexts, responsibilities, interfaces and interactions as needed, and leave the details to the team. The developers, through the rigorous use of automated unit and storytests via continuous integration, are then able to incrementally and continually improve the system design (both within and across model context boundaries) without compromising system functionality. Gartner uses the term *Emergent Architecture*to describe this practice[[11]](#footnote-11).

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 exposes domain knowledge, you’re encapsulating. Focusing on the latter allows the architecture to emerge, 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? When a team assumes that it must design for *everything* to change, it quickly finds itself in a death spiral of over engineering based on speculative requirements rather than actual ones. In reality, design for change requires managing dependencies carefully by ordering and isolating cohesive areas of the system from each other. For the architect, designing for change implies selecting an architecture or design that complements this ordering and isolation.

Layered Architectures are typically employed to achieve the kind of ordering and isolation described here, but violate the Dependency Inversion Principle and thus enable, if not encourage, the kind of accidental coupling that works against the original purpose. As an alternative, consider the *Onion Architecture* approach[[12]](#footnote-12). Originally described by Jeffrey Palermo, the Onion Architecture approach focuses on isolating layers through interfaces, leveraging Inversion of Control to minimize coupling and, most important, making the Domain Model the star of the show.

For Domain-Driven Design and emergent architecture to truly be effective partners, the Domain Model should be both core to the application and isolated as much as possible 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 this kind of isolation, you have a structure that enables independent layers to evolve and change at different rates with little friction between and internal to those layers. The Domain Model can then be distilled as deeper insights into the domain become apparent, and thus can evolve even as infrastructure concerns like data access are implemented and tested. This applies to more than just vertical layering, as the architect can also provide strategic value by explicitly defining a context for each model and maintaining model integrity within and across bounded contexts[[13]](#footnote-13). The architect should also help articulate the value of core domain distillation to stakeholders.

In some ways, the kind of independence described here is exactly what the phrase “architect the lines, not the boxes” is intended to convey. By leveraging clean interfaces, Inversion of Control 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 flexible and change-absorbent without being too prescriptive.

**VI. Conclusion**

To remain valued and valuable, the architecture of a system must be relevant: intimately connected to both the core business domain and development team. An architect can establish this relevance by advocating the development of a Ubiquitous Language, eliminating the need for translation and fostering collaboration between domain experts and developers. That relevance will grow as the domain model is established as core to the software effort, is refined over time to deeply express the core business 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 the development team, and fully invested in the success of the software solution.  A commitment to the principles, patterns and practices of Domain-Driven Design and emergent architecture can provide the simplest yet most powerful result of all: software that solves a core business problem, adapts to new business needs, and continues to delight users for years to come.

*Paul Rayner is Solutions Architect and Principal for Virtual Genius, LLC. He blogs at www.virtual-genius.com/blog and can be reached at paul@virtual-genius.com. Brandon Satrom is the Chief Architect with Thought Ascent, Inc., a Microsoft Gold Partner. He blogs at www.userinexperience.com and can be reached at bsatrom@gmail.com.*

1. Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*(Boston: Addison Wesley, 2004).  For Domain-Driven Design resources see <http://bit.ly/ddd_resources>. [↑](#footnote-ref-1)
2. Encarta World English Dictionary <http://encarta.msn.com/dictionary_1861630702/model.html> [↑](#footnote-ref-2)
3. Ibid. [↑](#footnote-ref-3)
4. For more on StoryTests see <http://bit.ly/storytesting>. [↑](#footnote-ref-4)
5. Rick Mugridge and Ward Cunningham, *Fit for Developing Software: Framework for Integrated Tests*(Boston: Addison Wesley, 2004)*, 336.* [↑](#footnote-ref-5)
6. Adapted from Agile Testing Matrix by Brian Marick, in *Implementing Lean Software Development: From Concept to Cash* by Mary and Tom Poppendieck (Boston: Addison Wesley, 2007)*,*199. [↑](#footnote-ref-6)
7. Evans, 327. [↑](#footnote-ref-7)
8. Evans, 415. [↑](#footnote-ref-8)
9. Jack Reeves' seminal article "What is Software Design?" is available at <http://www.developerdotstar.com/mag/articles/reeves_design_main.html> [↑](#footnote-ref-9)
10. See Grady Booch's comments at <http://www.informit.com/articles/article.aspx?p=1405569> [↑](#footnote-ref-10)
11. <http://www.gartner.com/it/page.jsp?id=1124112> [↑](#footnote-ref-11)
12. <http://jeffreypalermo.com/blog/the-onion-architecture-part-1/>. Hexagonal Architecture by Alistair Cockburn (<http://alistair.cockburn.us/Hexagonal+architecture>) is a similar idea and predates Palermo's work. [↑](#footnote-ref-12)
13. Evans, 335 and 344. [↑](#footnote-ref-13)