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

Brandon Satrom and Paul Rayner

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

#### 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 life cycle, 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. Successful architects create designs that address both.

Architects, domain experts, and developers collaborate to mitigate complexity through strategic modeling and design. This requires a focus on the core business 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; still, architects who distill the business 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 a strong, sustainable competitive advantage to the business.

#### Ubiquitous Language

##### The (Hidden) Cost of Translation

According to Eric Evans, 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” 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 conversation 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:** Okay, 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’ll 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 e-mail, 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:** That makes sense.

In the first conversation, the architect and developer muddled the dialogue with the domain expert by introducing technical detail that was essentially irrelevant to the business domain. If a strategy pattern is to be used to solve a business problem, it is important to discuss how such a pattern should be implemented in one’s framework of choice. However, it is not useful to do so in a conversation that is designed to scope the domain and the software that is 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 business 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. Business domains typically suffer from inconsistencies and ambiguities that experts either are not aware of or allow to exist for various reasons. The jargon that invariably grows 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 lacks 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.



Translate

Refine

Translate

Agree

Figure 1. Cost of translation

Figure 2 illustrates an alternative model—one in which 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 2. Creation of 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. However, when domain experts and development teams get around the table without a Ubiquitous Language, the jargon that each brings to the table necessitates translation and guarantees that confusion will propagate into software. So, while deep domain knowledge and development of 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 business domain and technology concerns. Translation between domain experts and development teams often becomes an unofficial job responsibility. However, translation is not enough. The adoption of a Ubiquitous Language by everyone who is involved in developing the software involves a commitment to take the business domain seriously and focus on incorporating it as much as possible into both conversation and code. This means using the domain to develop the model in code, and leveraging the model to bring accuracy, clarity, and stability to the domain and ubiquitous language. With respect to the development team, many architects are also in leadership roles 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.

#### Relevant Models

##### What Is a Model?

A *model* can be defined as “a simplified version of something complex used in analyzing and solving problems or making predictions.2” It is a representation, simplification, and 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 copies only 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” When it is 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 that it represents in order to be useful in solving problems.

A *domain model* is no different. It is a widely accepted fact in software that domain models are intended to represent a business domain or “problem space.” What seems to be less accepted is the idea that these models first and foremost must express the business domain clearly, and not be an expression of technical jargon or framework limitations. The establishment of a Ubiquitous Language enables emphasis of a domain model that represents the domain accurately and deeply, instead of one that is filled with inexact terminology or obfuscating technical detail.

It is 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. However, 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 reflect the domain accurately when it comes to describing the entities and interactions of a domain model. Incorporating story-testing into the development process is one particularly effective way of saturating feature discussions and executable documentation with the ubiquitous language, which naturally leads to incorporating it into the subsequent automated tests and the production code.4 “Writing concrete examples as tests explores ways in which to use and evolve the ubiquitous language for expressing business objects, constraints, and rules5.”



Figure 3. Model artifact matrix6

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

##### Managing Complexity

The most important job of the model 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:

**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.

**Orthogonal complexity**—This type of complexity is embedded in the business domain, but is not core to the problem that is being addressed, or is a commodity that can be brought into the system. This should be purged from the domain model, as it is distilled over time.

**Accidental complexity**—This type of complexity is 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*7is a collection of principles and strategies that are 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 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 the architecture enables essential complexity, while walling-off the accidental and orthogonal complexities that tend to creep into systems over time. The architect also enables that architecture to evolve and mature as the system changes, to accommodate future shifts in business needs.

#### Emergent Architecture

##### Don’t Coddle, Encapsulate

Many architects prefer to detail architecture up front, before the development team is fully engaged on a project. While the 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 its role on a project to that of an automaton that churns out predefined modules with little-to-no creative thought. Too much upfront architecture is a form of over-specification, and over- specification of design details to developers is a form of coddling. Over-specification of internal component details creates inflexible boundaries and results in brittle software—something with which, as an architect, you are likely tasked. The development team will be inappropriately constrained, perhaps even insulted, by this approach.

However, a blank slate is no better. It is also dangerous to under-specify a system. With no boundaries and no intentional architecture, a design is destined to suffer from the implementation of suboptimal and localized decisions by both domain experts and developers. Keeping development team members 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 that the team is aware of the value of what it is delivering is for the architect to lead in the creation and communication of a domain-vision statement that elucidates the core domain and its value8.

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, as the code *is* the design9. An architect should be involved in the development of the model through conversation, modeling, documentation, prototyping, and coding10. By being actively engaged with a development team, the architect is less likely to make decisions that would be perceived as coddling. Architects will not only learn to value accurately the contributions of the rest of the development team, but will also be forced to keep their skills current, live with their own dictates, and avoid over-constraining themselves or the team.

Where constraints are needed, an architect 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 development team, through the rigorous use of automated unit and story tests via continuous integration, is then able to improve the system design incrementally and continually—both within and across model-context boundaries—without compromising system functionality. Gartner uses the term “Emergent Architecture”to describe this practice11.

When you use architectural specifications and models as a replacement for engagement with a development team, you are coddling. On the other hand, when you are focused on creating a loose boundary that exposes domain knowledge, you are encapsulating. Focusing on the latter allows the architecture to emerge, evolve, and—more 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 that we have 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 that is based on speculative requirements, instead of 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 that is described here, but they often 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” approach12. Originally described by Jeffrey Palermo, the “Onion Architecture” approach focuses on isolating layers through interfaces; leveraging inversion of control to minimize coupling; and, more importantly, making the domain model the star of the show.

For Domain-Driven Design and emergent architecture to be truly effective partners, the domain model should be both core to the application and isolated as much as possible from all concerns that are not relevant to the business domain. In practical terms, this means that orthogonal concerns such as logging, security, and persistence 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.

When you have achieved this kind of isolation, you have a structure that enables independent layers to evolve and change at different rates, and 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 such as 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 contexts13. The architect should also help articulate the value of core domain distillation to stakeholders.

In some ways, the kind of independence that is 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, architects can maximize their value to the domain and development teams by delivering an architecture that is flexible and change-absorbent without being too prescriptive.

#### Conclusion

To remain valued and valuable, the architecture of a system must be relevant—that is, intimately connected to both the core business domain and the 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 express the core business domain deeply, 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.

#### Acknowledgements

Thank you to Eoin Woods, Vaughn Vernon, Jeffrey Palermo, and Dan Haywood for providing valuable feedback on the draft manuscript.

#### References

1. Evans, Eric. *Domain Driven Design: Tackling Complexity in the Heart of Software.* Boston, MA: Addison-Wesley, 2004. (For more resources on domain-driven design, see <http://bit.ly/ddd_resources>.)
2. Microsoft Corporation.“[Model](http://encarta.msn.com/dictionary_1861630702/model.html)” (definition). *Encarta World English Dictionary*, 2009.
3. Ibid.

For more on story tests, see <http://bit.ly/storytesting>.

Mugridge, Rick, and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Upper Saddle River, NJ: Prentice Hall PTR, 2005; 336.

Adapted from “Agile Testing Matrix,” by Brian Marick, in *Implementing Lean Software Development: From Concept to Cash*, by Mary Poppendieck and Thomas D. Poppendieck (Upper Saddle River, NJ: Addison-Wesley, 2007), 199.

Evans, 327.

Evans, 415.

Jack Reeves’s seminal article “What Is Software Design?” is available at <http://www.developerdotstar.com/mag/articles/reeves_design_main.html>.

See Grady Booch’s comments at <http://www.informit.com/articles/article.aspx?p=1405569>.

<http://www.gartner.com/it/page.jsp?id=1124112>.

<http://jeffreypalermo.com/blog/the-onion-architecture-part-1/>.

Alistair Cockburn’s idea of hexagonal architecture (<http://alistair.cockburn.us/Hexagonal+architecture>) is similar and predates Palermo’s work.

Evans, 335 and 344.

###### About the authors

**Brandon Satrom** is the Chief Architect with Thought Ascent, Inc., a Microsoft Gold Partner. He blogs at [www.userinexperience.com](http://www.userinexperience.com) and can be reached at [bsatrom@gmail.com](mailto:bsatrom@gmail.com).

**Paul Rayner** is a Solutions Architect and Principal for Virtual Genius, LLC. He blogs at [www.virtual-genius.com/blog](http://www.virtual-genius.com/blog) and can be reached at [paul@virtual-genius.com](mailto:paul@virtual-genius.com).