RIDDL: Solving the Communication Gap Between Business and Technology

How a specification language rooted in Domain-Driven Design gives every stakeholder—from domain expert to software engineer—a single, shared, parsable language

Published: February 2026

Author: Reid Spencer, Founder & CTO, Ossum Inc.

Website: ossum.ai

Executive Summary

Software projects fail at alarming rates, and the root cause is rarely technical. Projects fail because the people who understand the business and the people who build the software do not speak the same language. Requirements are translated, interpreted, summarized, and re-translated until the original intent is unrecognizable. Eric Evans identified this problem in 2003 and proposed a cultural remedy: a shared, ubiquitous language for the team. Two decades later, most teams still struggle to sustain that shared language in practice.

RIDDL (Reactive Interface to Domain Definition Language) transforms Evans's cultural aspiration into a concrete, parsable artifact. It embodies every strategic and tactical concept from Domain-Driven Design—bounded contexts, entities, value objects, aggregates, sagas, and more—in a language that domain experts can read and that compilers can validate. Crucially, RIDDL is declarative: it captures what a system is, not how it is implemented. This paper explains the communication gap, shows how RIDDL closes it, and maps DDD concepts to their RIDDL equivalents.

The Communication Gap

A Persistent Problem

The Standish Group's annual CHAOS reports have documented project failure rates for over thirty years. While success rates have improved from roughly 16% in 1994 to around 35% today, the majority of software projects still arrive late, over budget, or with incomplete functionality. The leading causes are consistently rooted in communication: unclear requirements, changing specifications, and misalignment between stakeholders and development teams.

This is not a tooling problem. Organizations have invested heavily in project management software, diagramming tools, documentation platforms, and collaboration suites. Yet the fundamental gap persists because these tools address the symptoms of miscommunication without addressing its cause: business experts and technical teams speak different languages about the same system.

A business analyst writes "the customer places an order." A developer reads that and thinks about HTTP endpoints, database transactions, and message queues. A domain expert thinks about fulfillment timelines, inventory reservations, and customer notifications. They are describing the same event, but the words carry entirely different connotations for each audience. Every handoff between these groups introduces distortion, and that distortion compounds through the lifecycle of a project.

Evans and the Ubiquitous Language

In Domain-Driven Design: Tackling Complexity in the Heart of Software (2003), Eric Evans proposed that development teams should establish a "ubiquitous language"—a shared vocabulary drawn from the business domain that is used consistently in conversation, documentation, and code. When the language of the domain model appears directly in the source code, changes in understanding propagate naturally from conversations to implementation.

> *"A project faces serious problems when its language is fractured. Domain experts use their jargon while technical team members have their own language tuned for discussing the domain in terms of design… The terminology of day-to-day discussions is disconnected from the terminology embedded in the code."*
> — Eric Evans, Domain-Driven Design (2003)

Evans's insight was profound, but the practice has proven difficult to sustain. The ubiquitous language tends to exist only in conversation and informal documents. It drifts over time. New team members miss the nuances. The code gradually diverges from the domain model. Without a formal artifact that is the ubiquitous language, teams must rely on discipline and institutional memory—both of which erode under schedule pressure.

RIDDL: Crystallizing the Ubiquitous Language

From Cultural Practice to Formal Artifact

Evans envisioned the ubiquitous language as a living practice sustained by conversation, modeling sessions, and disciplined refactoring. This works well when a small, stable team maintains close collaboration. It breaks down as teams grow, members rotate, and organizations scale. The language fragments because it has no authoritative source—no artifact that everyone can point to and say, "this is what we mean."

RIDDL provides that artifact. A RIDDL specification is a hierarchically composed set of definitions that captures the domain model in a structured, version-controlled document. Every concept has a precise definition. Every relationship between concepts is explicit. Every term in the ubiquitous language appears as a named definition with a description that any stakeholder can read.

Hierarchical Composition

RIDDL organizes definitions in a natural hierarchy that mirrors how organizations think about their domains:

```
domain OnlineRetail is {
  briefly "The online retail business"

  context OrderProcessing is {
    briefly "Handles order lifecycle from placement to completion"

    entity Order is {
      briefly "A customer's purchase request"
      state Active is {
        fields {
          items: many OrderItem,
          total: Currency,
          status: OrderStatus
        }
      }
      handler Commands is {
        on command PlaceOrder {
          set field Order.status to OrderStatus.Placed
          send event OrderPlaced to context Fulfillment
        }
      }
    }
  }

  context Fulfillment is {
    briefly "Manages picking, packing, and shipping"
    // ...
  }
}
```

A domain expert reading this specification sees the business organized as they understand it: a retail domain containing order processing and fulfillment, orders with items and statuses, and clear events that flow between business areas. They do not need to understand programming to review and validate this model. The `briefly` and `described by` clauses provide plain-language documentation embedded directly in the model.

A software engineer reading the same specification sees typed data structures, command-event patterns, bounded context boundaries, and inter-context messaging. Both audiences read the same document, and both understand it in their own terms. This is the ubiquitous language made real: not a glossary pinned to a whiteboard, but a living, validated, version-controlled artifact.

Declarative by Design: What, Not How

A critical design decision in RIDDL is its declarative nature. RIDDL specifications describe what a system is and what it does—never how it does it. There are no implementation details, no technology choices, no framework configurations. This is deliberate and essential to RIDDL's role as a communication tool.

When a specification includes implementation details, it becomes illegible to non-technical stakeholders. A domain expert can review "when a PlaceOrder command arrives, the order status changes to Placed, and the Fulfillment context is notified." That same expert would lose the thread entirely if the specification included database connection-pooling configuration or message-broker topic names.

RIDDL includes function definitions, but they serve as a shorthand for declarative composition, not as procedural code. A function in RIDDL groups a sequence of declarative statements—setting fields, sending messages, telling entities—into a reusable, named unit. It is a description of behavior, not an implementation. The distinction matters: RIDDL functions say "these things happen" without prescribing the mechanism by which they happen.

This separation of what from how is precisely what makes RIDDL accessible to every stakeholder. Domain experts contribute their knowledge of business rules and processes. Architects contribute structural decisions about context boundaries and message flows. Implementation teams take the validated specification and choose the technologies, frameworks, and patterns that best realize it. Each group works in its area of expertise, and the RIDDL model serves as the shared contract between them.

DDD Concepts in RIDDL

RIDDL was designed to express every concept from Domain-Driven Design directly. Each DDD pattern has a corresponding RIDDL construct, enabling teams already familiar with DDD to adopt RIDDL without learning a new conceptual framework.

Strategic Design

DDD Concept	RIDDL Construct	Purpose
Domain	`domain`	Top-level business area; contains all related contexts
Subdomain	Nested `domain`	Subdivisions within a domain for finer organization
Bounded Context	`context`	Autonomous boundary with its own model and language
Context Map	Cross-context references	Explicit message flows between contexts
Ubiquitous Language	The specification itself	Every named definition is a term in the shared language

Tactical Design

DDD Concept	RIDDL Construct	Purpose
Entity	`entity`	Stateful object with identity and lifecycle
Value Object	`type`	Immutable data defined by its attributes
Aggregate	`entity` (root) + contained types	Consistency boundary around related objects
Repository	`repository`	Persistence abstraction for entities
Domain Event	`event`	Something that happened in the domain
Command	`command`	Request to change state
Query / Read Model	`query`/`projection`	Read-side data retrieval and CQRS projections
Service	`Context` with `handler`	Stateless operations responding to messages
Saga	`saga`	Long-running process coordinating multiple contexts
Factory	`handler` on creation commands	Entity creation logic

Beyond DDD: RIDDL Extensions

RIDDL goes beyond the original DDD tactical patterns with constructs that address the needs of modern distributed systems:

RIDDL Construct	Purpose
`user`	Named actor who interacts with the system
`epic` / `story`	User journeys expressed in domain terms, linking requirements directly to the model
`streamlet`	Reactive stream processing component for event-driven data flow
`connector`	Typed data flow between streamlets, making integration patterns explicit
`adaptor`	Translation layer between bounded contexts, formalizing the DDD anti-corruption layer
`projection`	CQRS read model with explicit update handlers
`state`	Explicit state definitions on entities, supporting state machine patterns
`function`	Reusable declarative behavior composition

These extensions do not break from the DDD philosophy—they extend it into areas that Evans did not address in 2003, such as reactive architectures, event streaming, and explicit user journey modeling. The extensions follow the same principle: they describe *what* the system does in terms that every stakeholder can understand.

The Specification as Source of Truth

One Document, Many Readers

The power of RIDDL lies in the fact that it produces a single artifact that serves every audience on the team:

- **Domain experts** read the model to verify that business rules, processes, and terminology are captured correctly. The `briefly` and `described by` clauses give them a narrative they can follow without technical training.
- **Architects** use the model to validate the system structure, including bounded contexts, entity relationships, message flows between contexts, and saga coordination. The RIDDL compiler validates structural integrity, catching broken references and incomplete definitions.
- **Developers** receive a formal specification that eliminates ambiguity about what to build. Data types, command and event contracts, handler responsibilities, and inter-context communication are all defined explicitly.
- **New team members** can read the RIDDL model as onboarding documentation. Because it is both the ubiquitous language and the system specification, it provides a complete picture of the domain without requiring months of tribal knowledge transfer.

Compiler-Validated Consistency

Informal ubiquitous languages drift because no one enforces consistency. A term used in a Monday meeting may mean something subtly different by Friday. RIDDL's compiler enforces consistency mechanically:

- Every reference must resolve to a defined term. If someone renames an event but forgets to update its handler, the compiler catches it.
- Messages sent between contexts must match declared types. The compiler verifies that producers and consumers agree on data contracts.
- Definitions must be structurally complete. An entity without state, a handler without cases, a context without content—these are flagged as warnings or errors.

This mechanical validation is something no whiteboard, wiki, or Word document can provide. The ubiquitous language, expressed as a RIDDL model, cannot silently become inconsistent.

Progressive Refinement

RIDDL supports the same iterative, modeling-first approach that Evans advocates. Teams can start with broad strokes and add detail as understanding deepens:

1. **Discovery:** Define domains, users, and epics in broad strokes. At this stage, the model reads like a project charter—a shared understanding of scope and stakeholders.
2. **Elaboration:** Add bounded contexts, entities, and message flows. The compiler begins validating structural relationships and catching inconsistencies.
3. **Specification:** Define handlers, state machines, data types, and sagas. The model becomes detailed enough to serve as an unambiguous contract between business and technology.
4. **Documentation:** Generate diagrams, data dictionaries, and architecture views directly from the model. Documentation is always current because it is derived rather than maintained separately.

At every stage, the model remains readable by all stakeholders. Domain experts can review the high-level structure even as architects work on detailed context boundaries. This maintains alignment throughout the project lifecycle—the ubiquitous language evolves, but it never fractures.

Conclusion

Eric Evans showed us that complexity and the communication gap between business and technology are the central challenges of software development. His solution—a ubiquitous language shared by all stakeholders—was right in principle but difficult to

sustain in practice. Languages that live only in conversation and informal documents inevitably drift and fragment.

RIDDL solves this by crystallizing the ubiquitous language into a formal, hierarchically composed specification. Every DDD concept has a direct representation. Every term is a named, described definition. Every relationship is explicit and compiler-validated. The

specification is declarative—it captures what the system is, not how it is built—which keeps it accessible to every audience, from domain expert to software engineer.

The result is a single source of truth that every stakeholder can read, review, and contribute to. The ubiquitous language is no longer an aspiration maintained by team discipline; it is a versioned, validated artifact that evolves with the project and cannot silently become inconsistent.

For teams interested in how RIDDL also serves as a bridge between human architectural vision and AI-assisted code generation, see our companion paper, *RIDDL: The Design Bridge Between Humans and AI.*

References

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional.

Vernon, V. (2013). *Implementing Domain-Driven Design.* Addison-Wesley Professional.

Fowler, M. (2014). “Bounded Context.” martinfowler.com. Retrieved February 2026.

The Standish Group. (2020). *CHAOS Report: Beyond Infinity.* The Standish Group International.

Spencer, R. (2024). “RIDDL Language Reference.” ossum.tech. Retrieved February 2026.

Spencer, R. (2024). “RIDDL EBNF Grammar.” ossum.tech. Retrieved February 2026.