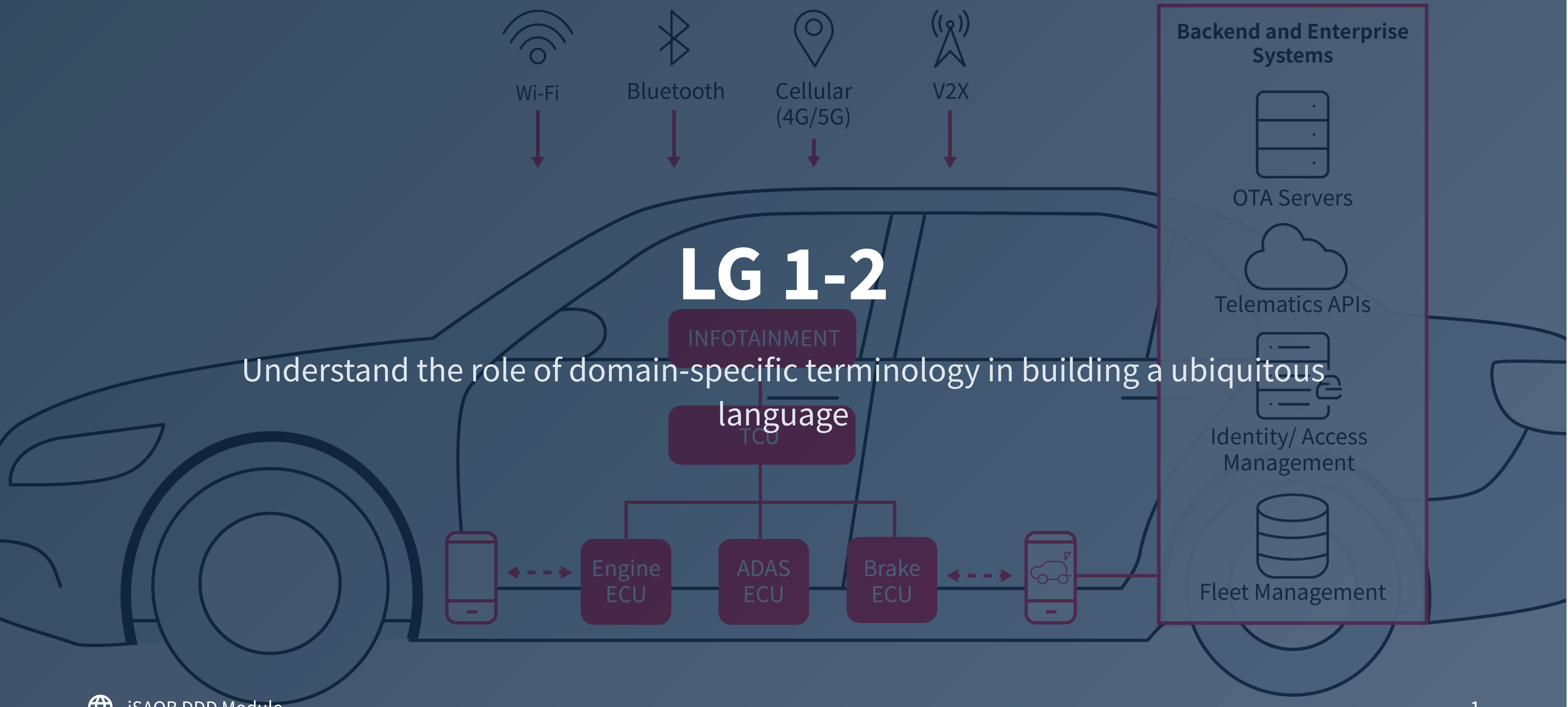


LG 1-2

Understand the role of domain-specific terminology in building a ubiquitous language

The illusion of "we all understand it"

- Teams talk **all day**
- Documents **exist**
- Code **compiles**
- And yet… misunderstandings **explode** at integration time

> *Start with a trap everyone has fallen into: "We thought we were aligned… until other team implemented it." Do not introduce "Ubiquitous Language" yet. Create discomfort first.*

The Communication Gap

Assumed Understanding
Everyone believes they're using the same meaning

Hidden Divergence
Different interpretations of the same terms

Integration Shock
Misunderstandings surface when systems connect

The most dangerous assumption in software

"When I say X, you mean the same thing"

Vehicle

Feature

Activation

Eligibility

Customer

Pause here. Ask participants to mentally pick one term they know is overloaded in their org.

Why this problem scales brutally in automotive

- **Multiple domains**
- **Long system lifetimes**
- **Regulatory language** vs engineering language
- **Hardware–software–cloud** intersections
- **Vendors, suppliers, integrators**

> *Key message: Automotive doesn't just have complexity — it has semantic drift over time.*

Scaling Challenge

Time Amplification
Semantic drift compounds over years

Intersection Complexity
Boundaries multiply where systems meet

Hidden Dependencies
Language creates invisible coupling

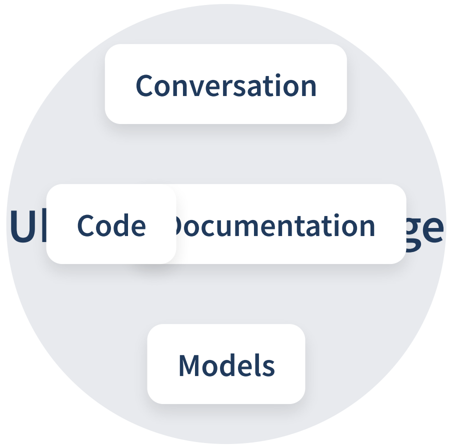
The formal idea (Eric Evans)

Ubiquitous Language (UL) is:

- A **shared language**
- Used by **domain experts** and developers
- Used **consistently**
- Within a **defined boundary**
- In **conversation**, documentation, models, and code

Stress word consistently. UL is not a glossary PDF that no one uses.

Consistency is Key



What UL is NOT

Ubiquitous Language is not:

- Just naming conventions
- Just "business-friendly class names"
- A one-time workshop output
- Global across whole company

This slide prevents common misapplications later.

The purpose of Ubiquitous Language

- Make models **communicable**
- Keep code **aligned** with domain meaning
- Surface **misunderstandings** early
- Reduce **translation layers** in human communication

> *Key insight: UL is an architectural control mechanism, not documentation.*

UL as Architectural Control

Semantic Consistency

Ensures shared understanding across boundaries

Explicit Boundaries

Defines where meaning changes

Evolution Support

Enables controlled language growth

Key Insight

UL is not about perfect documentation — it's about creating shared understanding that survives translation between humans and code.

Language shapes thought

- Humans reason **through language**
- **Ambiguous language** → ambiguous reasoning
- **Precise language** → precise models

> *Connect to real engineering: If you can't say it precisely, you can't implement it safely.*

Language Impact on Models

Ambiguous Language	Precise Language
"Feature activation"	"LaneAssistActivation"
"Customer data"	"OwnerProfile"
"Vehicle status"	"BatteryChargingState"

Key Insight

The precision of your language directly determines the precision of your models and implementation.

The "semantic debt" analogy

- **Technical debt** → code rot
- **Semantic debt** → meaning rot
- **Harder to detect**
- **More expensive to fix**

> *Semantic debt compounds quietly. UL is how you pay it down continuously.*

Debt Comparison

Technical Debt

Visible in code quality
Measured by tools
Impacts performance

Semantic Debt

Hidden in terminology
Detected in conversations
Impacts understanding

Key Insight

While technical debt affects implementation, semantic debt affects **understanding** and can silently undermine entire systems.

Why UML diagrams alone fail

- Diagrams **freeze meaning** at a point in time
- Language **evolves daily**
- Code **outlives diagrams**

Say this clearly: In DDD, code is the most important language artifact.

Documentation vs. Code

Static Documentation

Freezes language at creation
Rarely updated
Quickly becomes outdated

Living Code

Evolves with understanding
Executed and tested daily
Reflects current domain knowledge

Key Insight

In DDD, **code is the most important language artifact** — it's the only one that stays in sync with the domain.

UL is scoped, not global

- UL exists **within a bounded context**
- Same word may mean **different things** elsewhere
- ✓ That is **acceptable** — even healthy

> *Foreshadow bounded contexts without teaching them fully yet.*

Context Boundaries

Logistics Context

"Vehicle" = Transport asset

Autonomy Context

"Vehicle" = Sensing platform

Key Insight

Boundaries don't just prevent confusion — they **enable clarity** by allowing precise language within each context.

Example: "Vehicle" means different things

The same word, different meanings across contexts

Sales Context

Vehicle = **sellable configuration**

Manufacturing Context

Vehicle = **assembly instance**

Service Context

Vehicle = **warranty-bearing asset**

Should we force one "Vehicle" definition?

Answer: Absolutely not.

UL requires explicit boundaries

- Without boundaries:
- Language becomes **political**
- One team **dominates** semantics
- Models **collapse** under compromise

> *Important cultural point for senior engineers.*

Boundary Problems

Semantic Escalation
Disagreements become political battles

Power Imbalance
Dominant team imposes terminology

Model Dilution
Concepts lose precise meaning

Key Insight
Explicit boundaries prevent **semantic conflicts** by allowing each context to maintain its own precise language.

UL emerges from collaboration

- Conversations
- Scenarios
- Modeling
- Conflict resolution

> *UL is not "defined by architects". It is negotiated.*

Collaborative Discovery

Raw Terms
Unclear, overlapping meanings

Conflicts Surface
Different interpretations emerge

Definitions Sharpen
Through negotiation and modeling

Key Insight
Ubiquitous Language is **discovered**, not invented. It emerges from shared understanding.

Knowledge Crunching preview

- Interviews
- Event Storming
- Domain storytelling
- Observation

This connects LG 1-2 to Day-2 content.

Extracting Language

Collaborative
Multiple perspectives
Shared discovery

Scenario-based
Real workflows
Context-rich

Key Insight
Knowledge crunching is how we **discover** domain language, not how we **invent** it.

The UL lifecycle

Evolution of Ubiquitous Language

1. Raw terms
2. Conflicts surface
3. Definitions sharpen
4. Terms enter models
5. Terms enter code
6. Feedback refines language

Discovery

Raw terms emerge from conversations
Conflicts surface

Modeling

Definitions sharpen
Terms enter models

Refinement

Terms enter code
Feedback refines language

Key Insight

UL is **alive**. If it's static, it's dead.

> *Make it explicit: UL is alive. If it's static, it's dead.*

What belongs in a UL glossary

For each term:

Name
The term itself

Definition
One sentence explanation

Example
Concrete usage

Synonyms to avoid
Words that cause confusion

Boundary notes
Where this applies

Key Insight

Glossaries are **working tools**, not dictionaries.
They evolve with understanding.

Sample Entry

Term: BatteryChargingSession

Definition: A period where a vehicle battery is actively receiving power

Example: "Initiate BatteryChargingSession when connector is inserted"

Avoid: PowerTransfer, Refueling, EnergyIntake

Boundary: Energy Management Context

Example: "Activation"

UL Glossary Entry

Term
Activation

Definition
Making a purchased feature operational on a specific vehicle instance

Example
Activating lane assist after OTA update

Avoid
Enable, TurnOn, Configure

Boundary
Feature Management context

Key Insight

Explain why **banning synonyms** is essential:
Prevents semantic drift
Forces precision in thinking
Creates shared understanding

Code Impact

Class Name: FeatureActivation
Method: activateFeature(featureId, vehicleId)
Event: FeatureActivated
Test: shouldActivateLaneAssistAfterUpdate()

Ask: If glossary says "Activation", why does code say `enableFeature()`?

Why banning words matters

Problems Without Banning

Synonyms Hide Disagreement
Different words mask different meanings

Different Words Hide Same Concept
Fragmented understanding of the same idea

Same Word Hides Different Concepts
Overloaded terminology creates confusion

Key Insight

> UL is about **forcing clarity through discomfort**. Banning synonyms forces precise thinking.

Cognitive Impact

Forces Precision
Eliminates ambiguity in thinking

Creates Shared Vocabulary
Everyone uses same terms

Surfaces Hidden Conflicts
Makes disagreements visible

> *UL is about forcing clarity through discomfort.*

UL and code

Where UL Appears in Code

Class Names
FeatureActivation, BatteryChargingSession

Method Names
activateFeature(), beginCharging()

Domain Events
FeatureActivated, ChargingStarted

Tests
shouldActivateLaneAssist()

Key Question

If glossary says **"Activation"**, why does code say `**enableFeature()**`?

Code Comparison

Without UL

```
void process(int id, int type) {
  if (type == 1) {
    enableFeature(id);
  } else if (type == 2) {
    turnOnFeature(id);
  }
}
```

With UL

```
void activateFeature(FeatureId id)
  // Activation logic here
}
```

UL ensures code speaks the same language as domain experts.

Common UL anti-patterns

Warning Signs

Generic Names
Data, Manager, Handler

Technical Terms in Domain
Repository, DTO, Entity

Business Terms in Infrastructure
Customer in database layer

"We'll Clean Names Later"
Technical debt accumulation

Key Insight
"Later" almost never comes. Anti-patterns signal UL is not being actively maintained.

Impact on Code

Cognitive Load
Developers must translate intent

Communication Gap
Domain experts can't validate code

Semantic Debt
Meaning compounds incorrectly

Integration Failures
Misunderstandings surface at boundaries

> *"Later" almost never comes.*

The translation hell anti-pattern

Multiple Translation Layers

Domain Expert
Business knowledge

BA
Requirements translation

Spec
Technical specification

Dev
Implementation

Key Problem

Each step **translates language**, creating:

- Meaning drift
- Lost nuance
- Implementation gaps

Translation vs. Direct

Translation Hell

Domain Expert → BA → Spec → Dev → Code

Multiple interpretation points

Semantic drift at each step

Ubiquitous Language Solution

Domain Expert ↔ Dev

Direct communication

Shared terminology

UL collapses translation chains.

Code that hides domain

Domain-Expressive vs. Domain-Hiding Code

Domain-Hiding

```
class Battery {
  int status; // 0=off, 1=on, 2=charging
  int temp;  // in celsius
  int rate;  // kW
}
```

Domain-Expressive

```
class BatteryChargingSession {
  BatteryState state;
  Temperature temperature;
  ChargingRate rate;
}
```

Key Insight

Domain-expressive code **communicates intent** while domain-hiding code requires **translation**

Impact on Understa

Flags & Magic Value

```
if (status == 2) {
  // What does 2 mear
  // Why is temp in C
  // What units is ra
}
```

Reuse earlier charging example i

Tesla: "Disengagement" as language

Disengagement as Domain Concept

Not Just Telemetry
A business-significant event

Drives Analysis
Enables learning & improvement

Regulation Support
Enables compliance reporting

Key Insight
Language choice **enables analysis** and creates actionable business concepts.

Code Implementation

Domain Event
AutopilotDisengaged

Method
recordDisengagement(reason, context)

Test
shouldRecordDisengagementWhenSafetyTriggered()

Language choice enables analysis.

Toyota: platform language evolution

Platform Language Distinctions

Platform ≠ Vehicle
Software architecture vs physical asset

Capability ≠ Feature
Technical ability vs marketable option

Service ≠ ECU
Logical function vs hardware component

Function ≠ Software
Business capability vs implementation

Key Insight

UL evolves as **architecture evolves.** Precise language prevents confusion between related concepts.

Language Evolution

Traditional Automotive
ECU-centric, hardware-focused

Software-Defined Vehicle
Service-oriented, capability-focused

UL Evolution
New terms emerge as concepts change

UL evolves as architecture evolves.

Puzzle 1

If two teams use the same word but never interact, is that a problem?

Team A

"Vehicle" = Transport asset

Team B

"Vehicle" = Autonomous platform

Answer

Only when **models must integrate**

Puzzle 2

When should a term be split into two?

- Different **rules** apply
- Different **lifecycles** exist
- Different **experts** own it

Diagnostic scenario

Code with Missing Domain Language

process()

handle()

execute()

Key Question

What **domain language** is missing?

Impact of Missing Language

Hidden Intent

Lost Communication

Maintenance Burden

Key Insight

Force participants to **see absence as a smell** — missing domain language is a design flaw.

UL → Tactical patterns

Foundation for Tactical Patterns

Entities
Need names

Value Objects
Need meaning

Events
Need past-tense language

Aggregates
Need consistency

Services
Need clear verbs

Repositories
Need domain focus

Key Insight

UL provides the **naming foundation** for tactical patterns. Without precise language, patterns become meaningless.

Code Examples

Entity
class Vehicle

Value Object
class BatteryState

Domain Event
class FeatureActivated

Service
class ChargingService

Language Impact

Tactical patterns **amplify UL** — poor naming makes patterns ineffective.

UL → Strategic design

Language as Strategic Signal

Divergence

Signals new contexts

Convergence

Signals cohesion

Key Insight

Language patterns reveal **strategic boundaries** before architecture does.

UL as Discovery Tool

Naming Conflicts

Reveal hidden boundaries

Translation Gaps

Identify integration points

Strategic Value

UL is an **early-warning system** for architectural decisions.

UL as an architectural early-warning system

Early Warning Signals

Conflicting Definitions
Hidden coupling

Naming Arguments
Boundary discovery

Awkward Names
Missing concepts

Domain Expert Confusion
Model misalignment

Strategic Value
UL issues are **leading indicators** of architectural problems. Addressing language prevents larger failures.

Prevention Strategies

Active Glossary
Living documentation

Regular Review
Continuous alignment

Domain Expert Involvement
Validation by owners

Language First
Code expresses domain

Architectural Control
UL is not documentation — it's **architectural control** through shared understanding.

Self-study before Day 2

Before Day 2

- Identify **3 overloaded terms** in your project
- Write **competing definitions**
- Note **who owns each meaning**

Tomorrow: Language extraction techniques