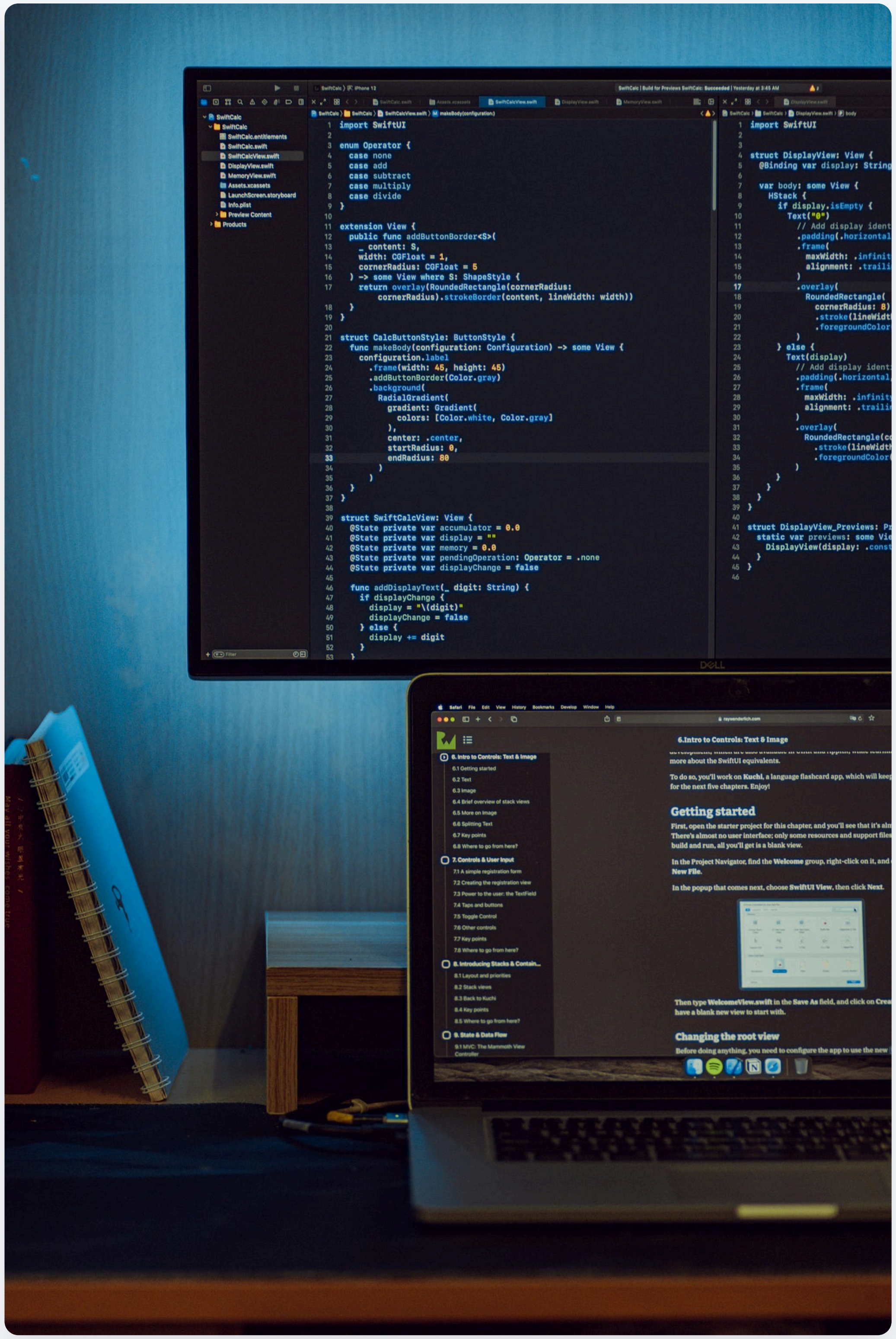
LG 1-1

Explain the connections between domain, software, and models

The uncomfortable truth about most enterprise software

- Most software systems do **not fail because of technology**
- They fail because the software does **not mean** what the business thinks it means
- Code runs correctly — but the system behaves wrongly

> *Ask: "How many of you have seen a system where requirements were 'implemented correctly' but business still said it's wrong?"*

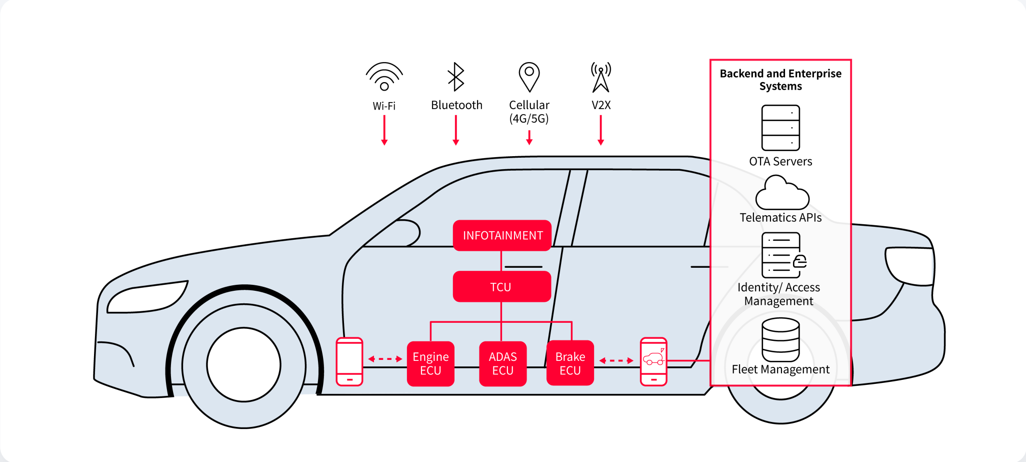


A familiar automotive pain

Vehicle Order Feature Activation Eligibility

- Same word, **different meanings**
- Different teams, **different assumptions**
- Integration bugs that are **not technical bugs**

> *Make the point: Misalignment is not a tooling problem. It's a meaning problem.*



Why traditional software thinking fails here

- Databases capture **structure**, not meaning
- APIs expose **operations**, not intent
- Microservices split **deployment**, not understanding
- UML documents describe **shape**, not behavior

> *Important: Do NOT bash microservices or UML. Frame them as insufficient, not wrong.*

Database tables **store** data

Domain models **represent** meaning

Traditional code **implements** features

DDD code **expresses** business concepts

The core problem we are solving

How do we make software **faithfully represent** complex automotive reality — and keep it that way as reality changes?

Domain - The messy reality

Model - The disciplined abstraction

Software - The precise execution

Why iSAQB starts DDD here

- Before **patterns**
- Before **architecture**
- Before **bounded contexts**
- We must understand:

> *Explicitly say: "If this slide doesn't land, the rest of the week becomes pattern memorization."*

1. What is **reality**?
2. What is a **model**?
3. What is **software's role**?
4. How do they **connect**?

LG 1-1 learning goal (explicit)

LG 1-1

Explain the connections between domains, software, and models

✗ Not: "Define domain"

✗ Not: "Draw a diagram"

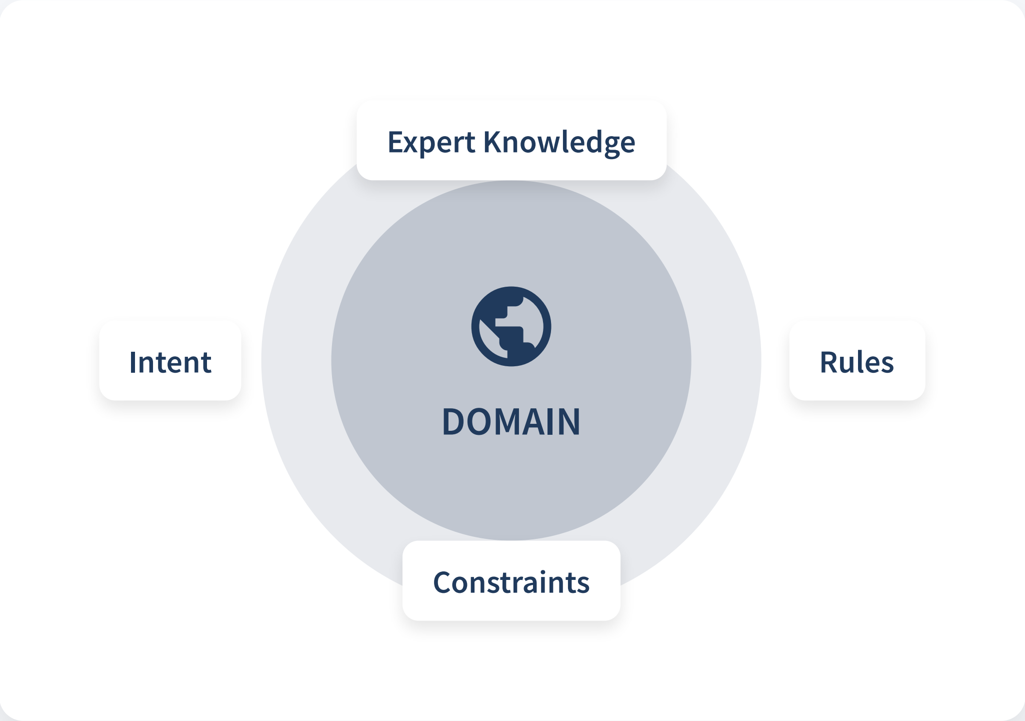
✓ But: **Explain the connection**

What is a Domain? (not the dictionary answer)

A domain is:

- A sphere of **expert knowledge**
- A space of **rules, constraints, and intent**
- **Independent** of software existence

> *Emphasize: The domain existed *before* the software and will exist *after* it.*



Automotive domain example: Battery charging

- Charging limits are **not technical decisions**
- Temperature constraints exist due to **chemistry**
- Regulations influence **allowed behavior**
- Engineers **already know this**

> *Key insight: Domain rules are *discovered*, not invented by developers.*

Charging Domain Rules

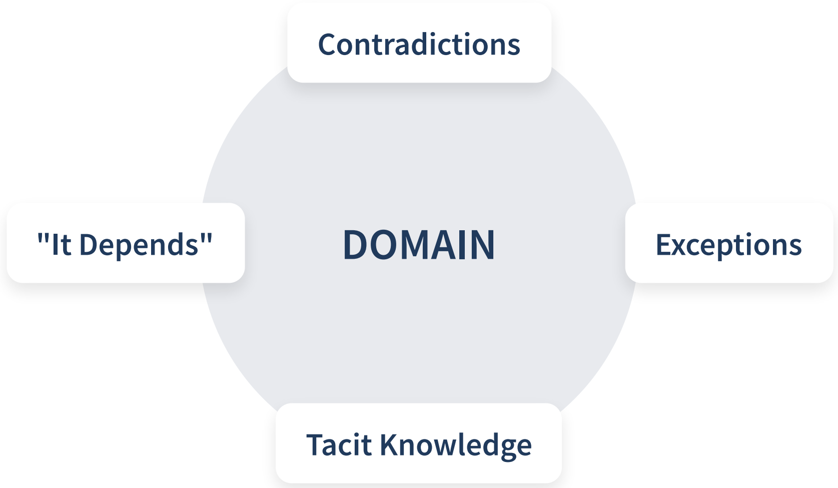
- Battery chemistry determines **maximum charging rate**
- Temperature ranges **limit charging** for safety
- Battery aging **affects capacity** over time

Domains are messy by nature

- Contradictions
- Exceptions
- Tacit knowledge
- "It depends" answers

> *Make this explicit: Humans tolerate ambiguity. Software cannot.*

Domain Reality



Humans

- ✓ Handle ambiguity
- ✓ Adapt to exceptions
- ✓ Use intuition

Software

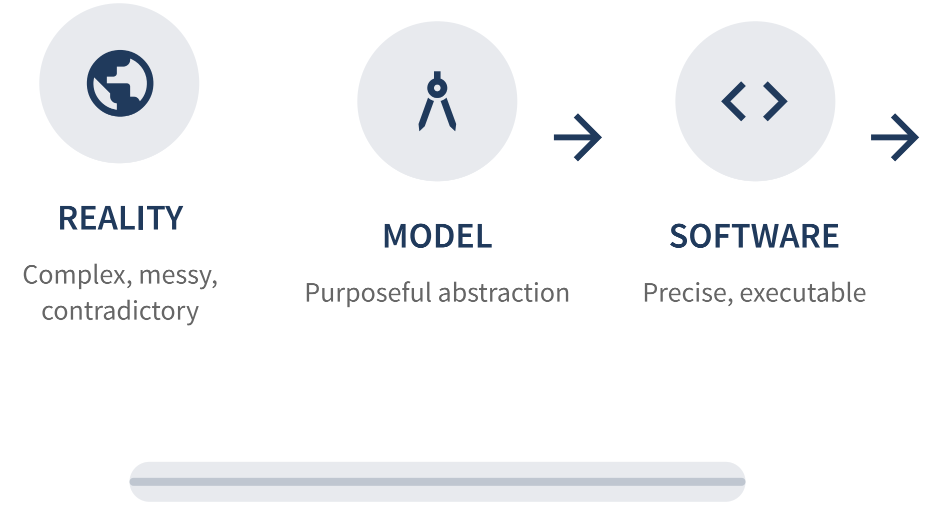
- ✗ Needs precision
- ✗ Requires explicit rules
- ✗ Cannot guess intent

Why we need models at all

- Reality is too **complex**
- Software needs **precision**
- Models are the **bridge**

> *Introduce Evans' idea without quoting yet.*

The Modeling Bridge



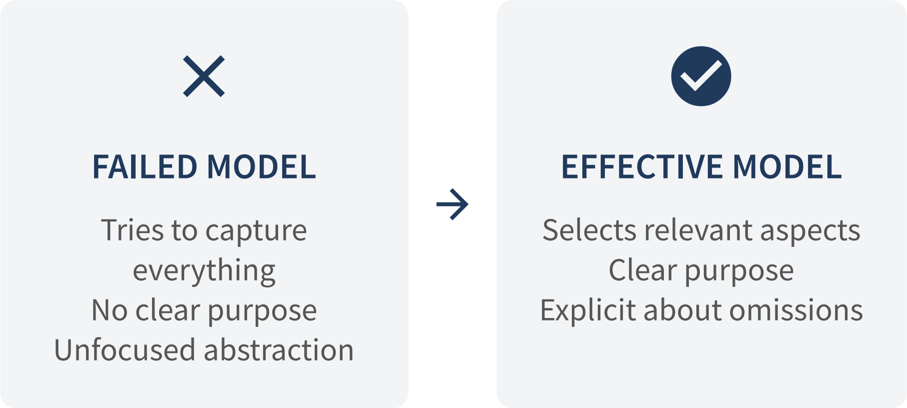
What a model really is

A model is:

- A **selective abstraction**
- Built for a **specific purpose**
- Explicit about what it **ignores**

> *Stress: A model that tries to capture everything is a failed model.*

Models are purposeful simplifications



FAILED MODEL

Tries to capture everything
No clear purpose
Unfocused abstraction

→

EFFECTIVE MODEL

Selects relevant aspects
Clear purpose
Explicit about omissions

Map analogy (critical)

- Road map ≠ satellite image
- Metro map **distorts** geography intentionally
- Accuracy is **not realism**; it is **usefulness**

> *This analogy will be reused throughout the course.*

Key Insight

Like maps, models must serve their purpose. A London Underground map is "wrong" geographically but "right" for navigation.

Model as Map

REALITY

Complete, complex, overwhelming
All details present
Not purpose-fit

MODEL

Simplified, focused, purposeful
Selective abstraction
Useful for specific needs

Mental Model

When we model a domain, we're creating a "map" that navigates complexity for a specific purpose.

Automotive modeling example

Battery Modeling Example

Physical Battery

- Cells
- Ions
- Heat
- Aging
- Chemistry

Charging Model

- State of Charge
- Temperature
- Eligibility
- Charging Rate

Key Question

Is the charging model "wrong" because it ignores chemistry?

No — it is purpose-fit.

Model Purpose Fit

Research Purpose

- ✓ Detailed chemistry
- ✓ Ion behavior
- ✓ Material properties

Software Purpose

- ✓ Charging decisions
- ✓ Safety constraints
- ✓ User experience

Key Insight

Models are not "more complete" or "less complete" — they are either purpose-fit or not.

What software means in DDD

- Software is **executable domain knowledge**
- Not all code is **domain software**
- **Infrastructure** ≠ domain logic

> *This is subtle and important for senior devs.*

Key Insight

In DDD, software is the bridge between domain knowledge and execution. It's not just code that runs, but code that means something.

Types of Software in DDD

Domain Software

Encodes business meaning
Hard to replace safely
Core value of the system

Infrastructure

Frameworks
Databases
Protocols

DDD Protection

DDD protects domain code from infrastructure churn by separating concerns.

Sacred vs replaceable code

Code Classification

Domain Code

- Encodes business meaning
- Hard to replace safely
- Core business value

Infrastructure

- Frameworks
- Databases
- Protocols

DDD Protection

DDD protects domain code from infrastructure churn by separating concerns.

Impact of Change

Domain Changes

- High risk
- Business impact
- Requires expertise

Infrastructure Changes

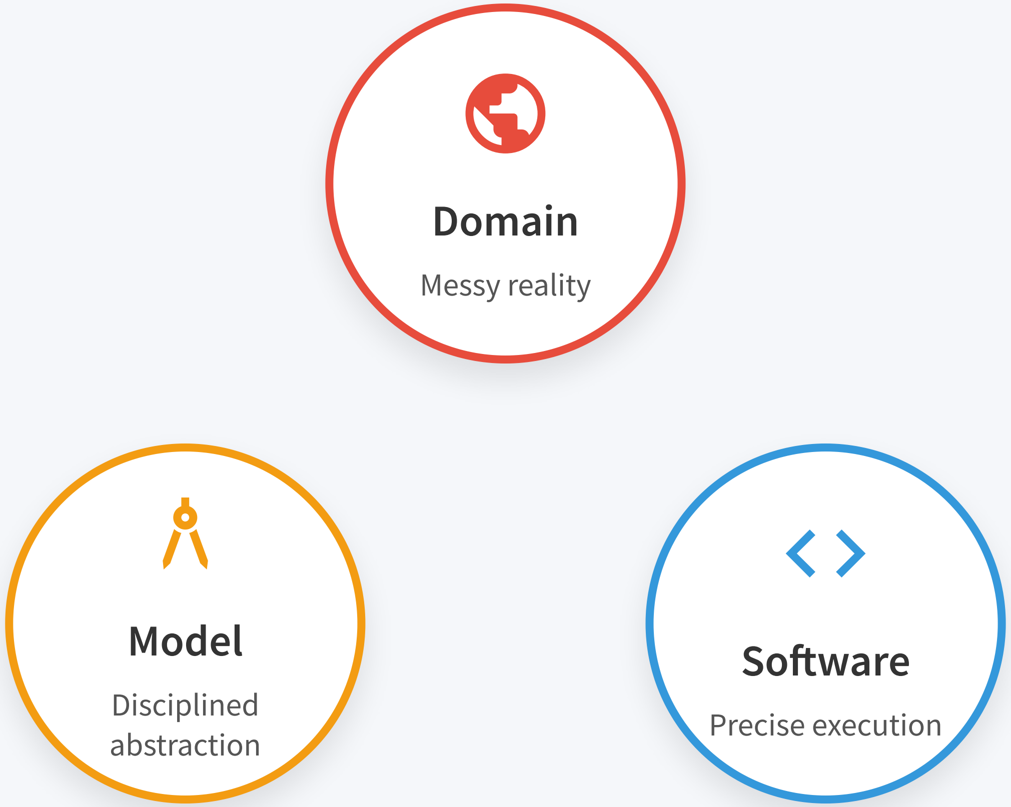
- Lower risk
- Technical impact
- Developer expertise

Key Insight

Say explicitly: DDD protects domain code from infrastructure churn.

The triangle

Domain → Model → Software



Domain

Messy reality

Model

Disciplined abstraction

Software

Precise execution

The Anchor Connection

This triangle represents the **core relationship** in DDD: how we translate complex reality into executable software through purposeful modeling

Flow of knowledge

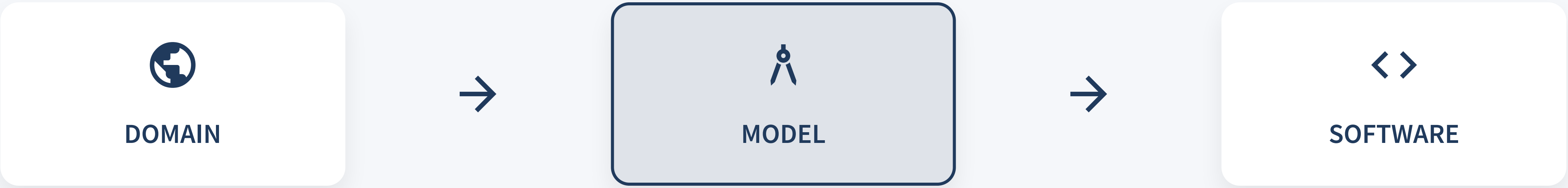
1. **Experts describe**
   Domain experts describe reality

2. **Teams extract**
   Teams extract meaning

3. **Models stabilize**
   Models stabilize understanding

4. **Code implements**
   Code implements the model

5. **Feedback refines**
   Feedback refines the model



Key Insight

This process is **iterative, not linear**. Knowledge flows in both directions, constantly refining our understanding.

Where systems usually break

Model frozen too early

Models created before full understanding
Rigid abstractions that can't evolve

Code diverges from model

Implementation drifts from design
Technical shortcuts accumulate

Language drifts between teams

Same words, different meanings
Context lost in translation

New rules patched instead of modeled

Quick fixes over proper abstractions
Technical debt accumulates

⚠ Break Pattern

These breaks happen when we **disconnect** the triangle: Domain → Model → Software

Tesla: disengagement as a domain concept

Disengagement as Domain Concept

Domain Event

Not just a log entry
First-class concept in the model

Enables Analysis

Patterns of disengagement
Performance metrics

Facilitates Learning

System improvement
Edge case identification

Regulation Compliance

Safety reporting
Regulatory requirements

Key Teaching

When something becomes a domain concept, it becomes **visible** and **actionable**.

From Technical to Domain

Technical View

System state change
Control transfer event

Domain View

Safety boundary crossing
Autonomy capability limit

BMW: cloud vs vehicle models

Model Separation

Cloud Systems

- Modern APIs
- Data lakes
- High bandwidth
- Frequent updates

Vehicle Systems

- Legacy protocols
- Embedded systems
- Limited bandwidth
- Safety-critical

Key Insight

BMW maintains **separate models** for cloud and vehicle systems, with **explicit translation** between them.

Model Translation

Cloud Model

User experience focus
Business logic
Data analytics

⇄

Vehicle Model

Real-time constraints
Safety protocols
Hardware interface

Foreshadowing

This approach foreshadows the **Anticorruption Layer** pattern we'll explore later.

Domain-Driven Approach

Each model serves its **specific context** with clear boundaries and purpose.

Two code snippets (revisited)

Generic Data-Driven Code

```
function processVehicleData(data) {
if (data.type === 'charging') {
return {
status: data.status,
rate: calculateRate(data.temp, data.soc)
};
}
// More if statements for other types
}
```

Domain-Expressive Code

```
class BatteryChargingSession {
constructor(temperature, stateOfCharge) {
this.temperature = temperature;
this.stateOfCharge = stateOfCharge;
}

calculateChargingRate() {
return new ChargingRate(
this.temperature, this.stateOfCharge
);
}
}
```

Ask participants to read it aloud

Notice how the domain-expressive code speaks the language of the problem domain, while the generic code speaks the language of the implementation. Which would a domain expert understand?

Diagnostic questions

Can a domain expert validate this?

Where are business rules visible?

What breaks when rules change?

Key Insight

These questions help identify whether your code truly **expresses the domain** or merely implements technical requirements

Guided construction flow

1

Domain Conversation

Talk to domain experts
Extract key concepts

2

Key Concepts

Identify core vocabulary
Define relationships

3

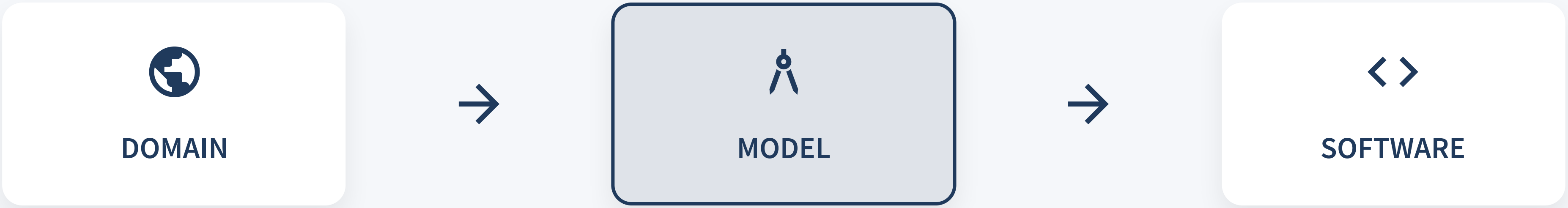
Model Sketch

Create purposeful abstraction
Focus on behavior

4

Domain-Centric Code

Express business concepts
Implement domain rules


DOMAIN → MODEL → SOFTWARE

Exercise Setup

This flow sets up the hands-on exercise where you'll apply the concepts we've discussed

Exercise framing

Not Designing Software

Focus on extracting meaning
Not on technical implementation

Extracting Meaning

Identify core domain concepts
Model business behavior

Precision > Completeness

Focus on what matters
Not on capturing everything

Exercise Focus

You are not designing software — you are **extracting meaning**. Precision beats completeness.

Puzzle 1

Can two models of "Vehicle" both be correct at the same time?

Model A: Logistics

- Cargo capacity
- Dimensions
- Transportation type
- Delivery timeline

Model B: Autonomous Driving

- Sensor configuration
- Processing capability
- Navigation system
- Safety protocols

Think about...

Models are **purpose-bound**. The same real-world concept can have multiple valid models depending on context and use case.

Puzzle 2

When is a database table a domain model — and when is it not?

Domain Model Table

- Expresses business concepts
- Enforces domain rules
- Rich with behavior
- Uses ubiquitous language

Infrastructure Table

- Primarily for data storage
- Normalized for efficiency
- Technical constraints
- Lacks business meaning

Think about...

Tables are **storage mechanisms**, not meaning. Domain models live in **behavior & constraints**, not just data structures.

Answers & discussion

Models are purpose-bound

- ✓ Multiple models of same concept can be valid
- ✓ Context determines model's correctness
- ✓ Purpose guides abstraction choices
- ✓ No single "right" model exists

Tables are storage, not meaning

- ✓ Tables are data structures, not concepts
- ✓ Meaning lives in behavior & constraints
- ✓ Domain model transcends persistence
- ✓ Database tables are infrastructure

Key Takeaway

- The **distinction** between model and implementation is critical
- Domain meaning exists **independent** of technical implementation

Why LG 1-1 enables everything else

Without LG 1-1 foundation

Ubiquitous Language

Collapses without shared understanding

Bounded Contexts

Become arbitrary boundaries

Architecture

Becomes accidental complexity

DOMAIN

MODEL

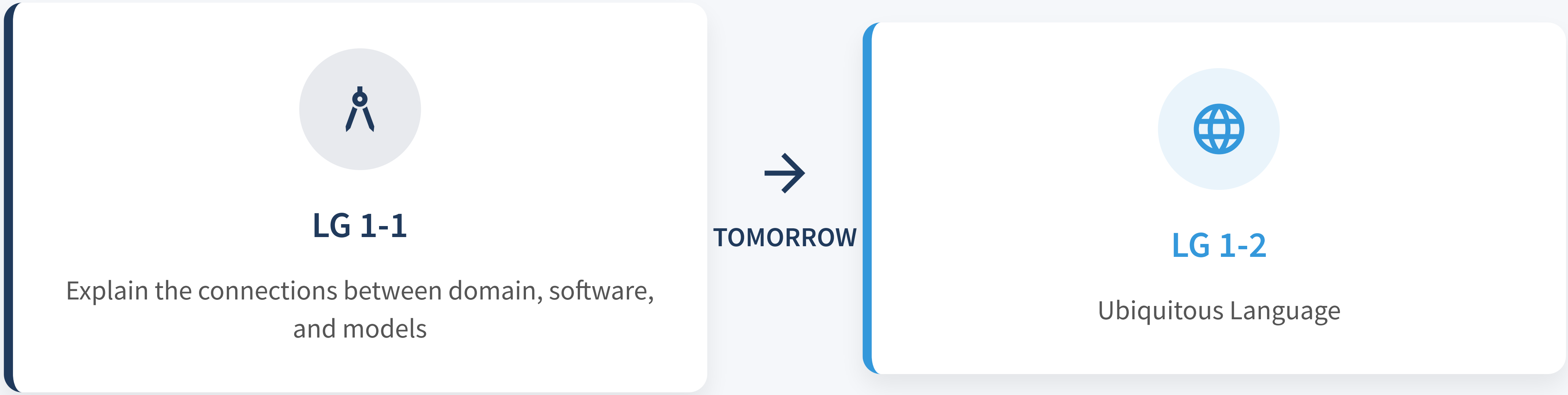
SOFTWARE

Foundation Principle

Understanding the connection between **domain, model, and software** is essential before applying any DDD patterns

Connection to LG 1-2 (next)

If models are shared understanding...



LG 1-1

Explain the connections between domain, software, and models

→ TOMORROW

LG 1-2

Ubiquitous Language

Language is the glue that keeps models shared

- Prevent semantic drift
- Language as architectural control
- Team communication
- Shared understanding