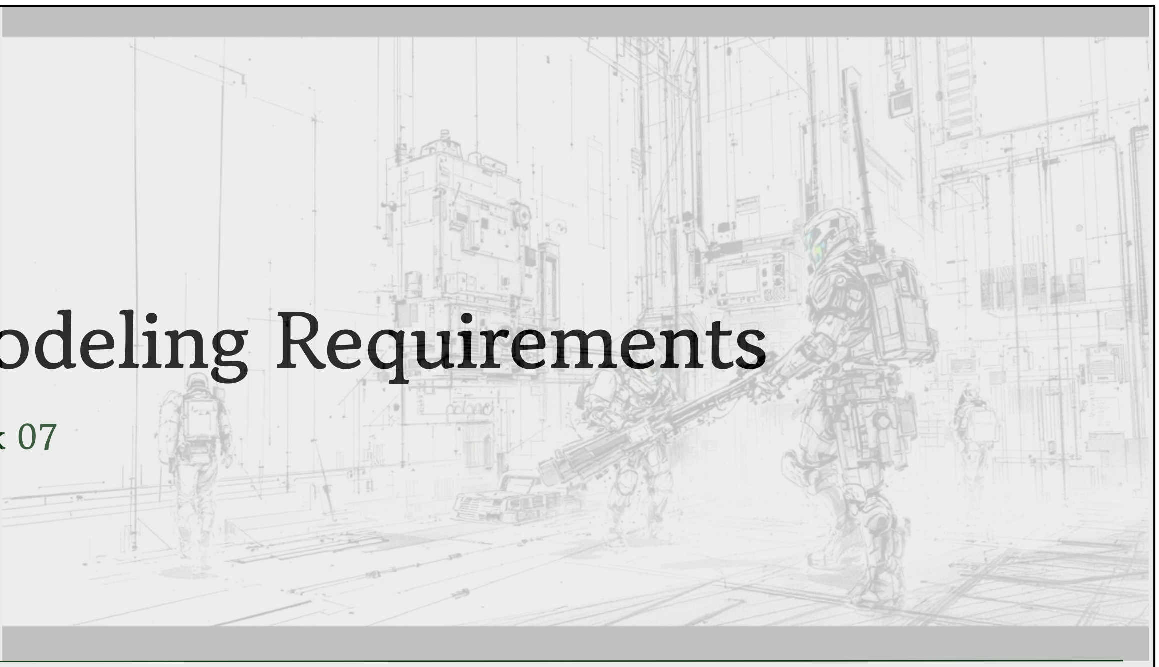
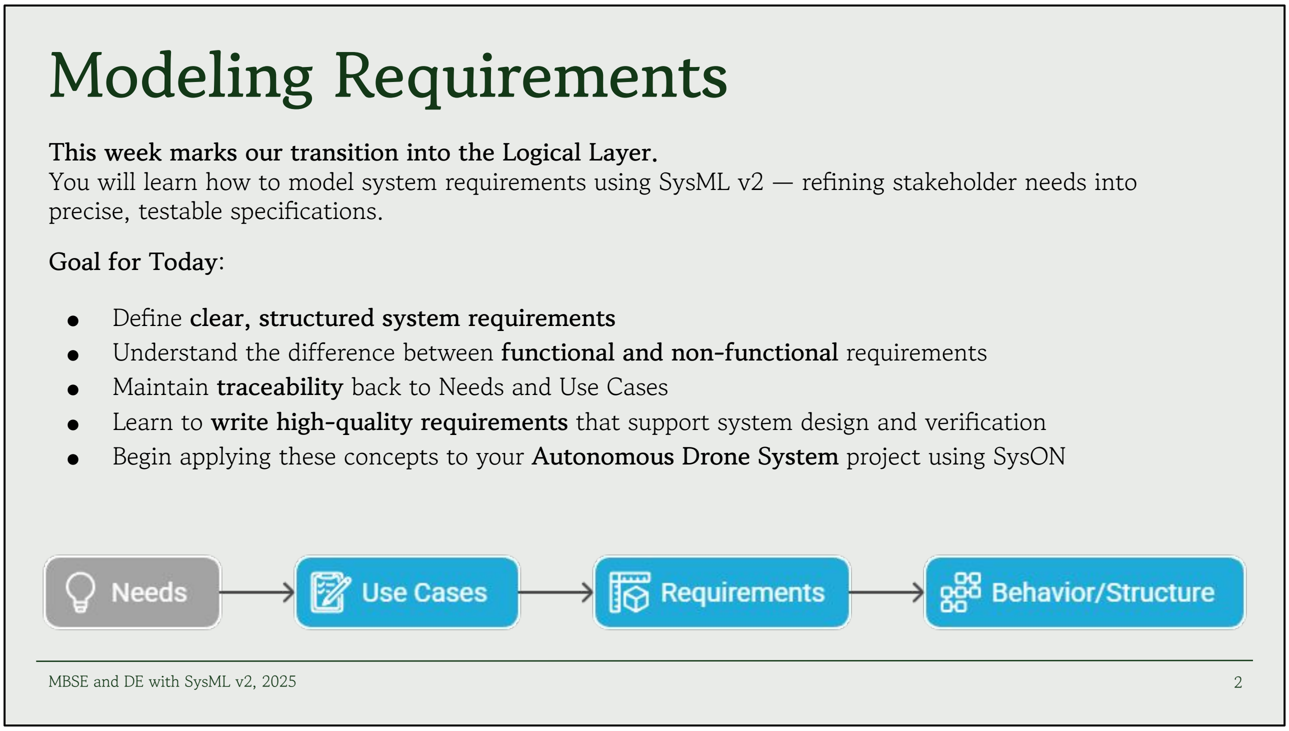


Modeling Requirements

Week 07

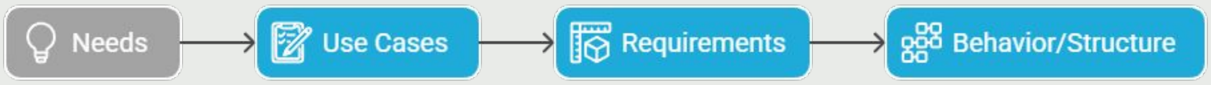


Modeling Requirements

This week marks our transition into the Logical Layer.
You will learn how to model system requirements using SysML v2 — refining stakeholder needs into precise, testable specifications.

Goal for Today:

- Define **clear, structured system requirements**
- Understand the difference between **functional and non-functional** requirements
- Maintain **traceability** back to Needs and Use Cases
- Learn to **write high-quality requirements** that support system design and verification
- Begin applying these concepts to your **Autonomous Drone System** project using SysON



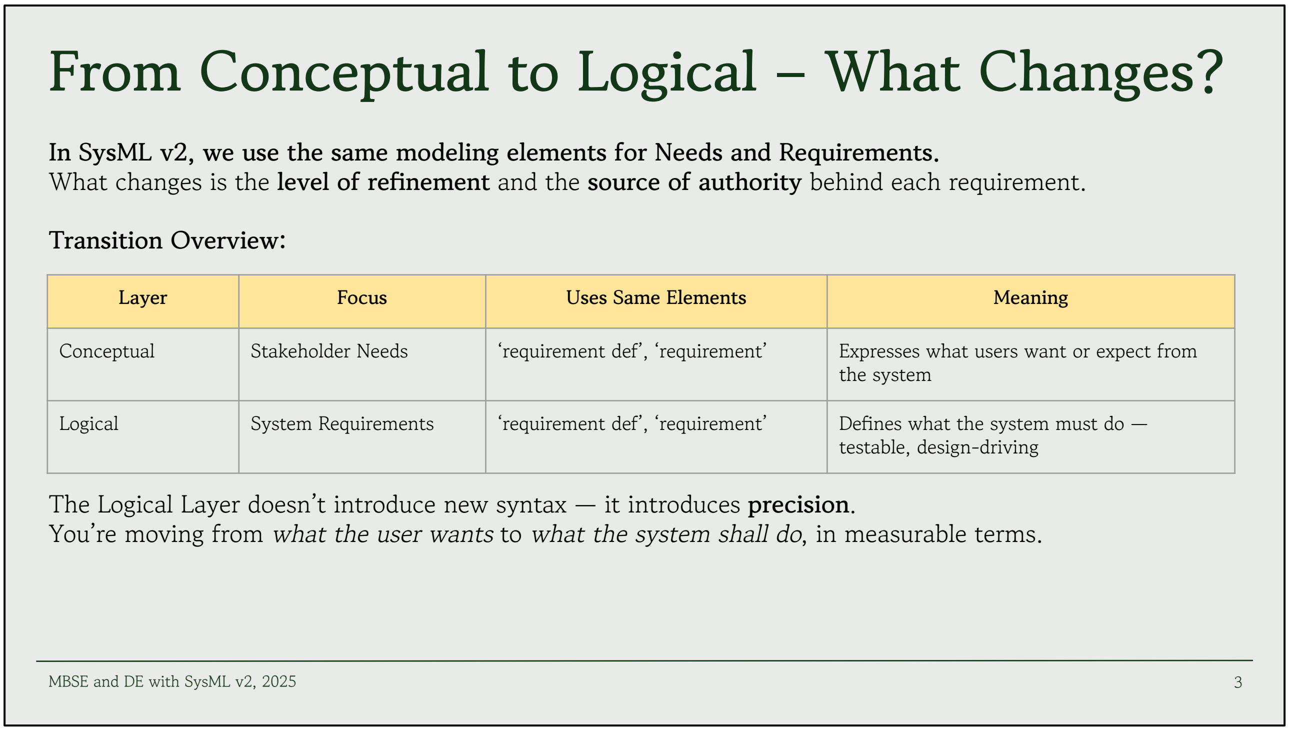
From Needs to Requirements
Welcome to Week 7. This is the point where we begin building the **Logical Layer** of your system model.
In previous weeks, you captured **stakeholder needs**, defined **Use Cases**, and began describing system **scenarios**. Now, we shift focus to answering:

What must the system do — in measurable, verifiable terms?

This is the domain of **Requirements** in SysML v2.
Today, you'll learn:

- How to create formal requirements from stakeholder input
- The different types of requirements — and how to structure them
- How to maintain traceability from Need → Use Case → Requirement
- And how to make sure your requirements are **testable, specific, and clear**

These are foundational skills for any MBSE practitioner — and they directly support what you'll model in your Drone System going forward.



From Conceptual to Logical – What Changes?

In SysML v2, we use the same modeling elements for Needs and Requirements.
What changes is the **level of refinement** and the **source of authority** behind each requirement.

Transition Overview:

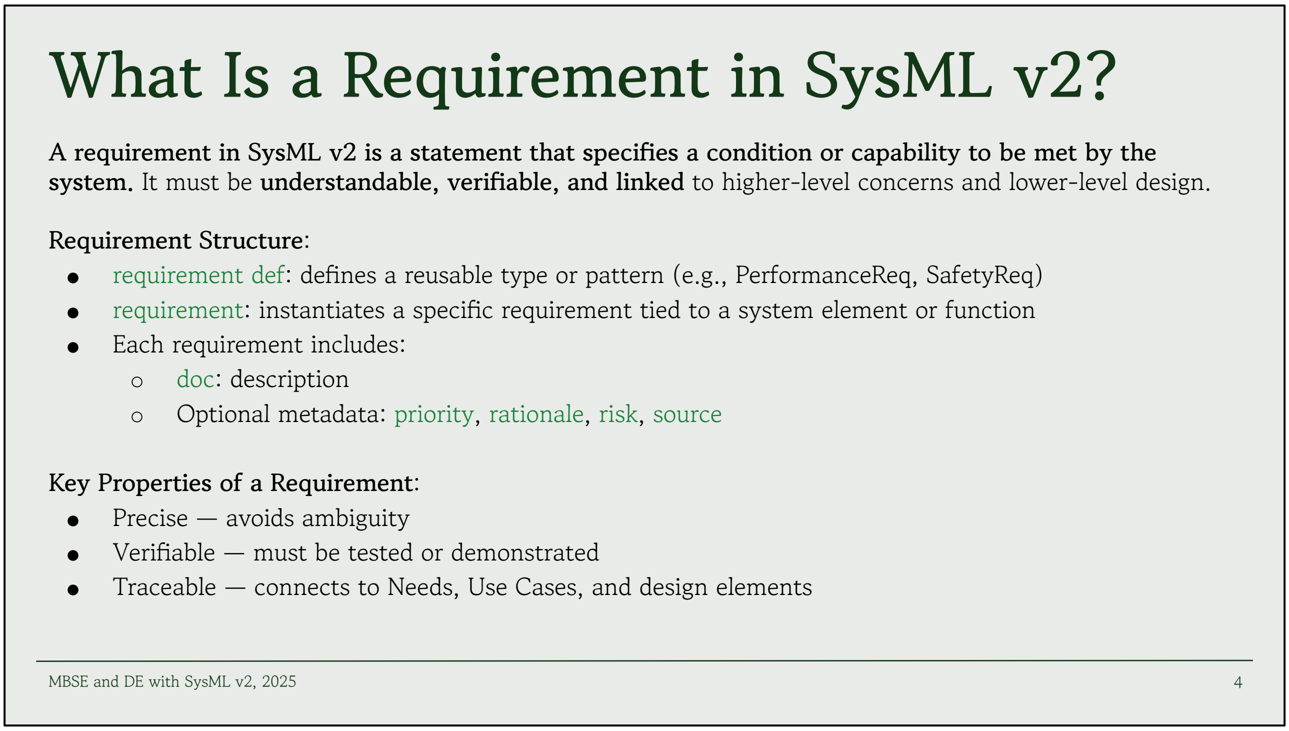
Layer	Focus	Uses Same Elements	Meaning
Conceptual	Stakeholder Needs	‘requirement def’, ‘requirement’	Expresses what users want or expect from the system
Logical	System Requirements	‘requirement def’, ‘requirement’	Defines what the system must do — testable, design-driving

The Logical Layer doesn’t introduce new syntax — it introduces **precision**.
You’re moving from *what the user wants* to *what the system shall do*, in measurable terms.

From Needs to Requirements
In SysML v2, there’s no separate syntax for Needs versus Requirements. Both are expressed using requirement def and requirement.
So what changes as we move from the Conceptual to Logical Layer?

- It’s not the syntax — it’s the **intent and level of refinement**.
- In the Conceptual Layer, you model high-level **stakeholder needs**. These express what the user wants the system to do.
- In the Logical Layer, you **refine** those into formal **system requirements** — statements the system must fulfill, in a way that’s testable and design-ready.

This is a shift from *why* the system exists (stakeholder value) to *what* the system must do (engineering terms).
You maintain the same structural language — but give it more detail, rigor, and traceability.



What Is a Requirement in SysML v2?

A requirement in SysML v2 is a statement that specifies a condition or capability to be met by the system. It must be **understandable, verifiable, and linked** to higher-level concerns and lower-level design.

Requirement Structure:

- requirement def: defines a reusable type or pattern (e.g., PerformanceReq, SafetyReq)
- requirement: instantiates a specific requirement tied to a system element or function
- Each requirement includes:
  - doc: description
  - Optional metadata: priority, rationale, risk, source

Key Properties of a Requirement:

- Precise — avoids ambiguity
- Verifiable — must be tested or demonstrated
- Traceable — connects to Needs, Use Cases, and design elements

From Needs to Requirements

In SysML v2, a requirement is a **structured, model-based element** that expresses what the system must do or what condition it must meet.

Requirements can describe:

- Functions (what the system does)
- Constraints (what it must not do)
- Quality thresholds (how well it must perform)

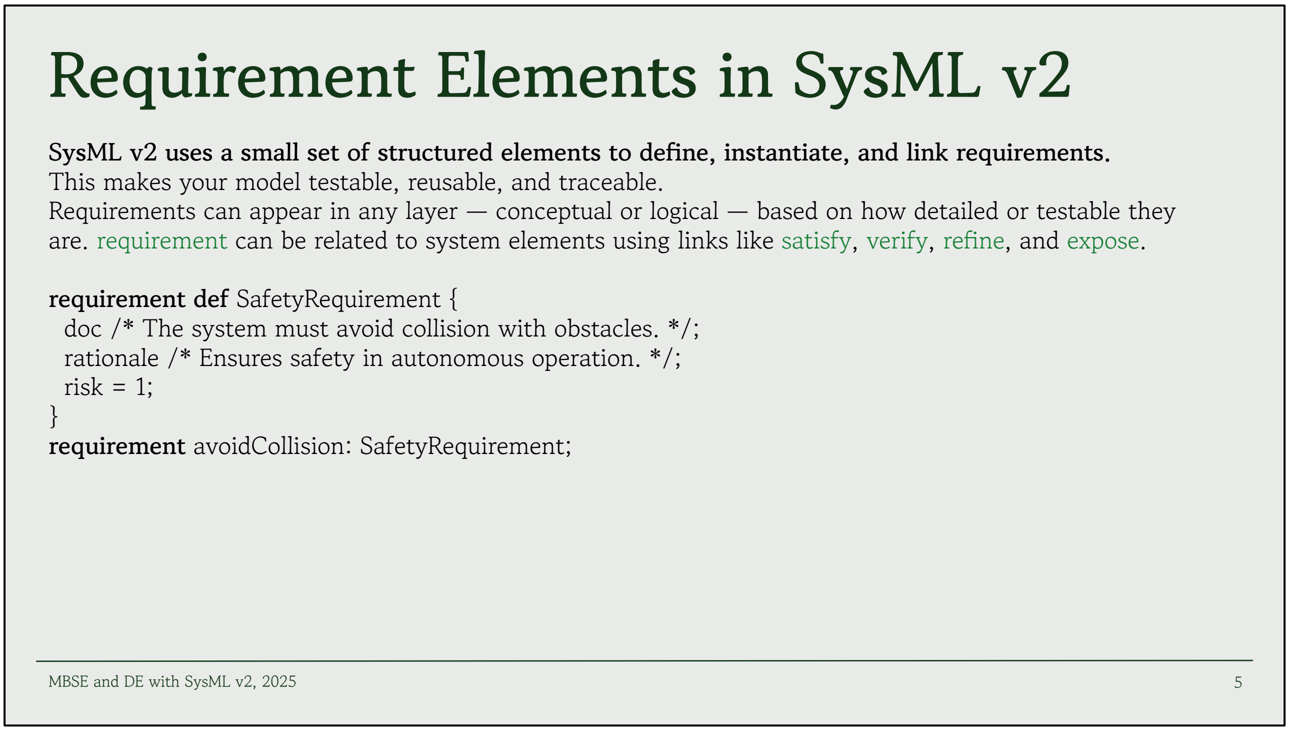
Every requirement is based on a requirement def — this acts like a type or template.

The actual statement is modeled as a requirement, which includes descriptive fields like doc, and optional metadata such as rationale or risk.

The goal is to make each requirement:

- **Understandable** to stakeholders
- **Verifiable** through testing or analysis
- **Traceable** to stakeholder intent and system design

This modeling discipline allows teams to catch inconsistencies early, validate against goals, and maintain alignment throughout the system lifecycle.



Requirement Elements in SysML v2

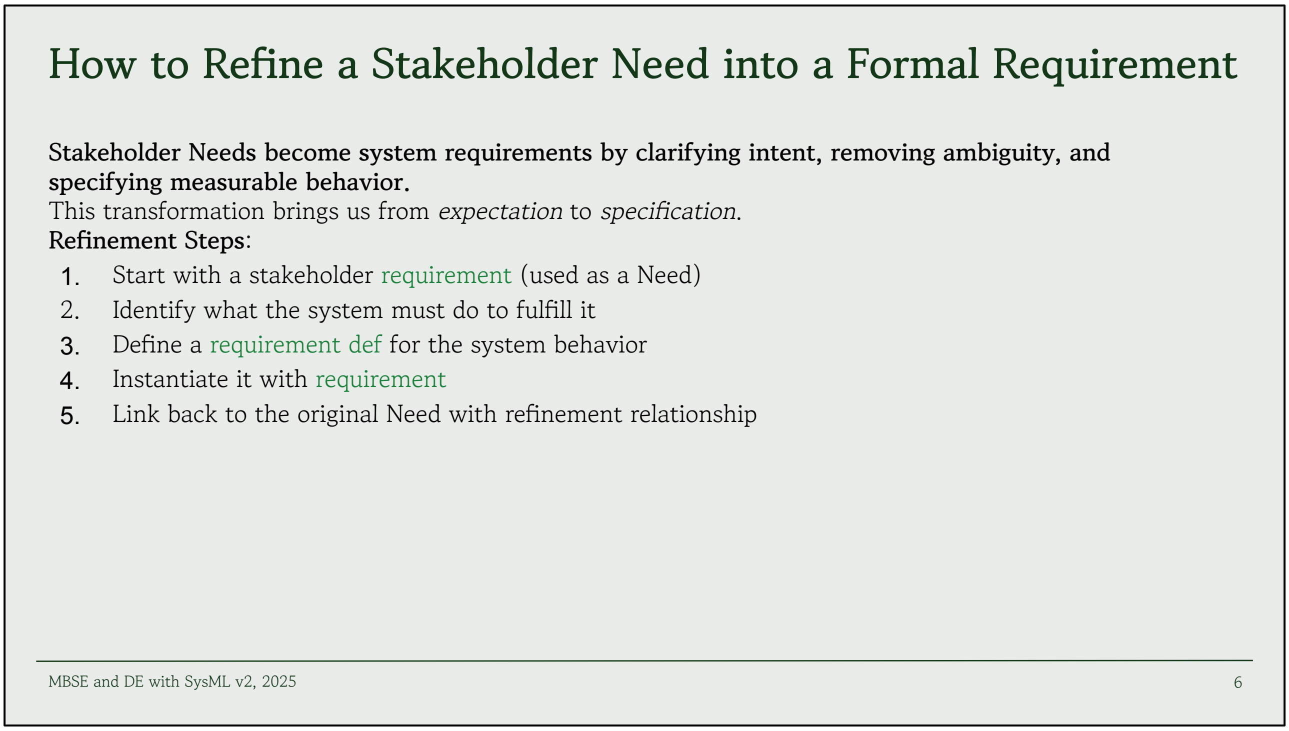
SysML v2 uses a small set of structured elements to define, instantiate, and link requirements.
This makes your model testable, reusable, and traceable.
Requirements can appear in any layer — conceptual or logical — based on how detailed or testable they are. requirement can be related to system elements using links like satisfy, verify, refine, and expose.

```
requirement def SafetyRequirement {
  doc /* The system must avoid collision with obstacles. */;
  rationale /* Ensures safety in autonomous operation. */;
  risk = 1;
}
requirement avoidCollision: SafetyRequirement;
```

From Needs to Requirements
SysML v2 provides a simple but powerful structure for modeling requirements.
At the core, you define a requirement def — which acts as the **type** or category of a requirement.
This can be general, like PerformanceRequirement, or specific, like MaxAltitudeRequirement.
Then you create a requirement instance, which includes:

- A doc field — the main natural language statement of what the system must do
- Optionally, a rationale to explain why it matters
- Tags like priority, risk, or source to help organize and manage the requirement

This keeps your model clean, repeatable, and structured.
In SysON, you'll define both the requirement def and the instance together — and eventually, link it to Use Cases, parts, or tests using traceability links.



How to Refine a Stakeholder Need into a Formal Requirement

Stakeholder Needs become system requirements by clarifying intent, removing ambiguity, and specifying measurable behavior.
This transformation brings us from *expectation* to *specification*.
Refinement Steps:

1. Start with a stakeholder requirement (used as a Need)
2. Identify what the system must do to fulfill it
3. Define a requirement def for the system behavior
4. Instantiate it with requirement
5. Link back to the original Need with refinement relationship

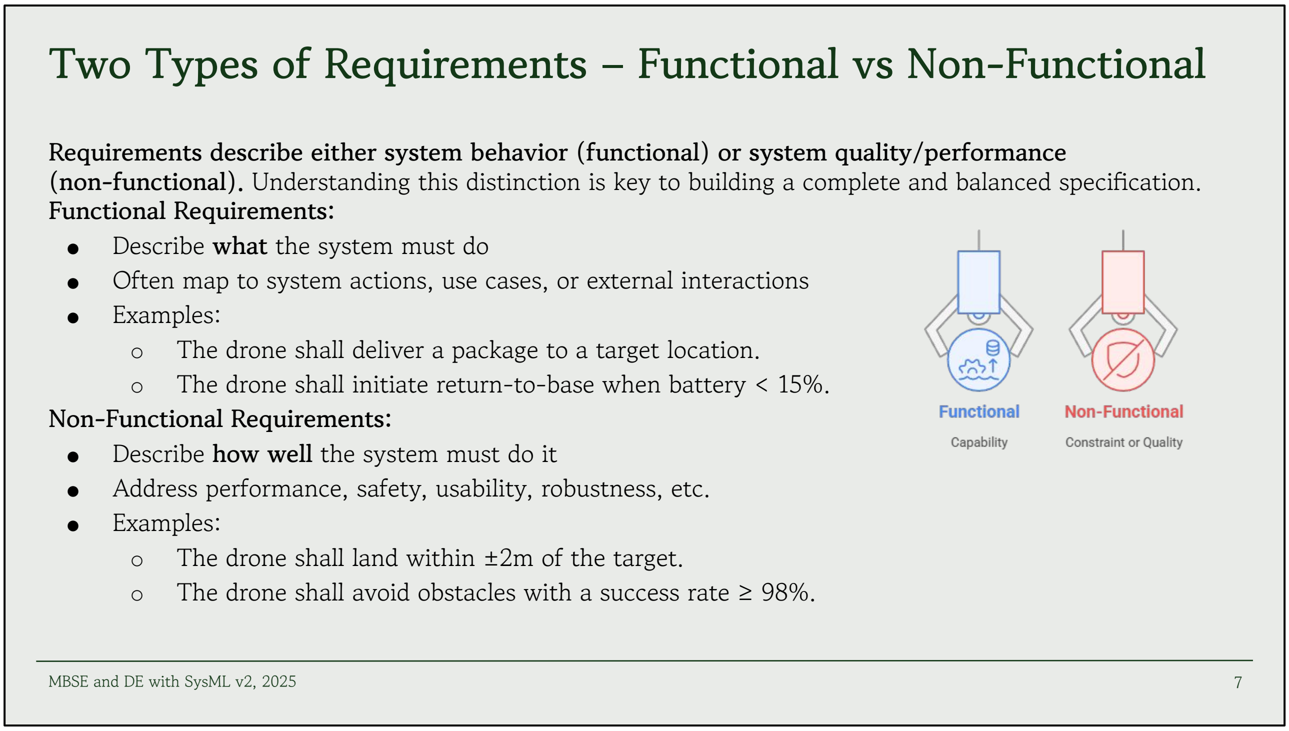
From Needs to Requirements
Let’s walk through how we turn a **stakeholder expectation** into a **structured requirement**.
In your Week 4 models, you captured high-level stakeholder needs using requirement def and requirement. For example:

> “The drone should avoid obstacles during delivery.”

That’s a good start — but it’s not testable yet.
Now, we refine it:

- We clarify what the system must *actually do* — e.g., detect and avoid obstacles within a defined distance
- We write that as a new requirement def and requirement in system terms
- Then we connect the two with a refinement relationship

This link maintains traceability, ensuring the final system still honors what the stakeholder originally asked for — but now in a way that engineers can verify and design toward.



Two Types of Requirements – Functional vs Non-Functional

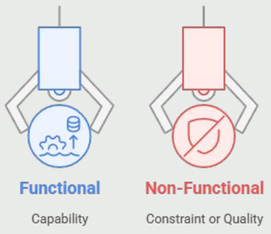
Requirements describe either system behavior (functional) or system quality/performance (non-functional). Understanding this distinction is key to building a complete and balanced specification.

Functional Requirements:

- Describe **what** the system must do
- Often map to system actions, use cases, or external interactions
- Examples:
  - The drone shall deliver a package to a target location.
  - The drone shall initiate return-to-base when battery < 15%.

Non-Functional Requirements:

- Describe **how well** the system must do it
- Address performance, safety, usability, robustness, etc.
- Examples:
  - The drone shall land within ±2m of the target.
  - The drone shall avoid obstacles with a success rate ≥ 98%.



Functional vs Non-Functional Requirements

There are two fundamental types of requirements you'll model: **Functional** and **Non-Functional**.

Functional requirements tell us what the system must do — its operations, behaviors, and interactions.

For example: deliver a package, follow a GPS route, or initiate emergency return.

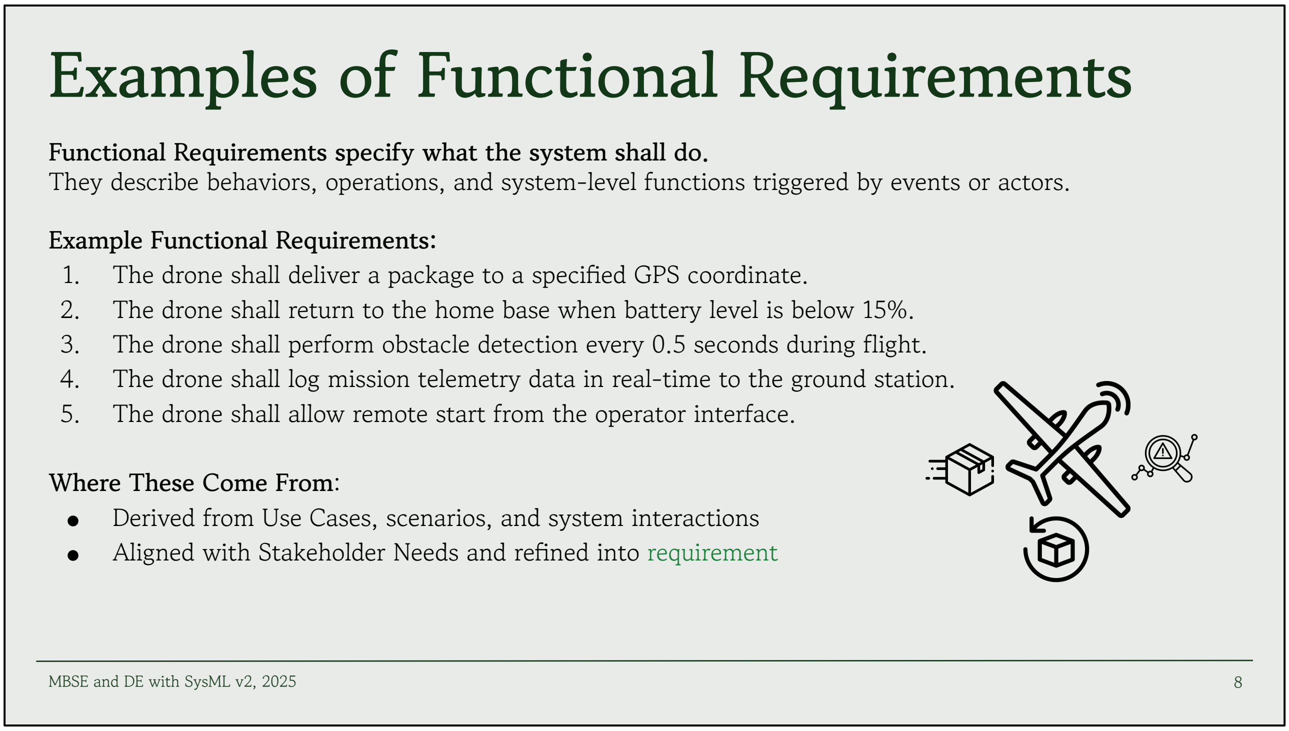
Non-functional requirements, on the other hand, express the **performance or quality constraints**.

They don't describe actions — they describe how *well* the system performs those actions:

- How precise is landing?
- How fast must response time be?
- What level of safety or reliability must be ensured?

This classification helps you organize your system logic and ensures you're covering both *capability* and *quality*.

In your drone system, you'll likely have a mix of both — and organizing them this way makes your model easier to validate, trace, and communicate.



Examples of Functional Requirements

Functional Requirements specify what the system shall do.
They describe behaviors, operations, and system-level functions triggered by events or actors.

Example Functional Requirements:

1. The drone shall deliver a package to a specified GPS coordinate.
2. The drone shall return to the home base when battery level is below 15%.
3. The drone shall perform obstacle detection every 0.5 seconds during flight.
4. The drone shall log mission telemetry data in real-time to the ground station.
5. The drone shall allow remote start from the operator interface.

Where These Come From:

- Derived from Use Cases, scenarios, and system interactions
- Aligned with Stakeholder Needs and refined into requirement

Functional vs Non-Functional Requirements
Here are some **realistic examples** of **Functional Requirements** — the kinds of statements you'll define in your project.
Each one:

- Describes a concrete **system behavior**
- Is phrased in a consistent structure: *"The system shall..."*
- Is traceable back to earlier Use Cases or Stakeholder Needs

For example:

- "The drone shall return to base if battery < 15%" — that likely comes from a safety-related stakeholder concern
- "The drone shall detect obstacles every 0.5 seconds" — this connects to your earlier scenario about avoiding collisions

These requirements will eventually be linked to the drone's structure (sensors, software) and verified through simulation or testing.
As you create your own, try to be specific about **what the system must do**, not how it does it — we'll handle the "how" later when we build structure and behavior layers.

Examples of Non-Functional Requirements

Non-Functional Requirements specify constraints or qualities — how well the system must perform, rather than what it must do.

Example Non-Functional Requirements:

1. The drone shall land within ±2 meters of the designated GPS target.
2. The drone shall complete the delivery mission within 15 minutes.
3. The obstacle detection system shall operate with ≥98% accuracy.
4. The system shall operate in ambient temperatures between −10°C and 50°C.
5. The communication link shall maintain latency below 200ms.

Where These Apply:

- Apply to system as a whole, or to individual capabilities
- Often linked to concern, requirement def, or validation test cases

Functional vs Non-Functional Requirements

Let’s look at some examples of **Non-Functional Requirements**.

These are not about actions — they’re about **constraints** or **performance levels** that those actions must meet.

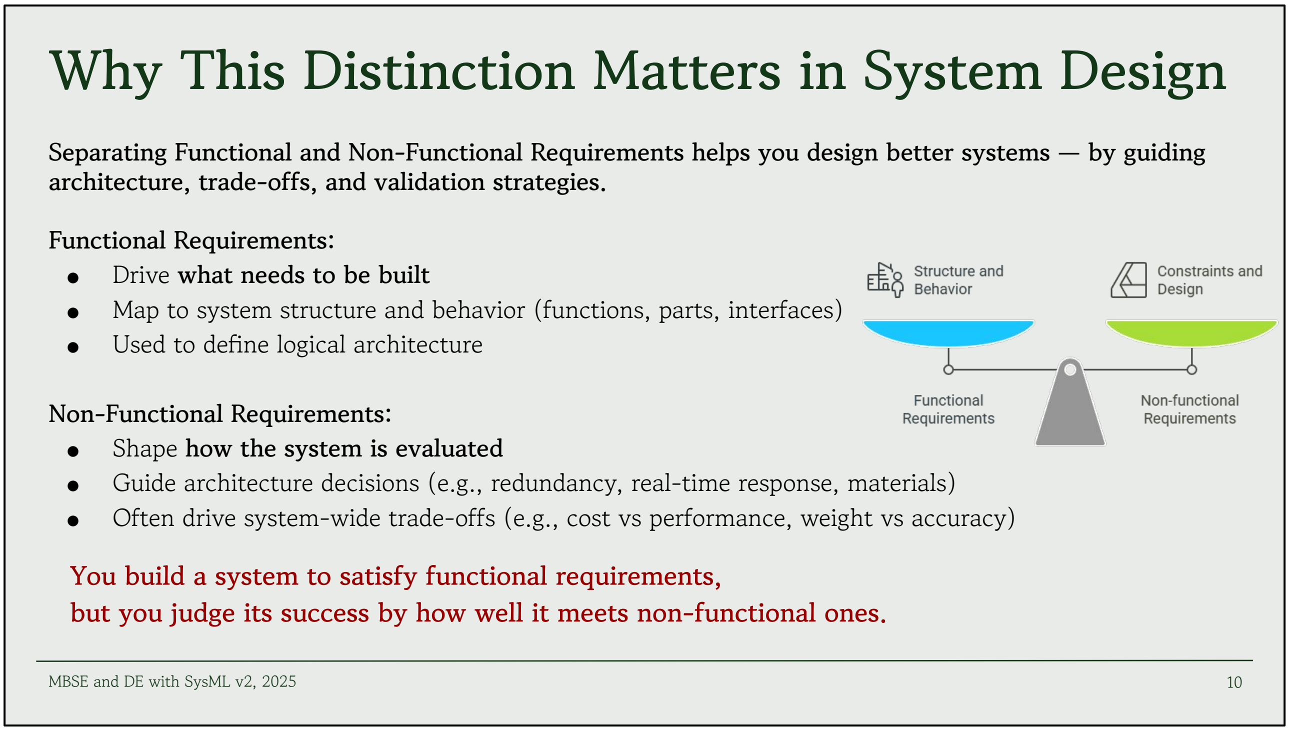
For example:

- “Land within ±2 meters” is about **precision**
- “Complete mission within 15 minutes” is about **timing**
- “Detect with 98% accuracy” is about **reliability**

These types of requirements help ensure that the system doesn’t just function — it functions **well enough** to meet user expectations, safety goals, and regulatory standards.

When you model these, try to include **quantifiable values** wherever possible. Avoid vague words like “fast,” “accurate,” or “efficient” unless you can define **how fast** or **how accurate**.

These requirements are often validated through simulation, test, or analysis — so clarity and specificity are critical.

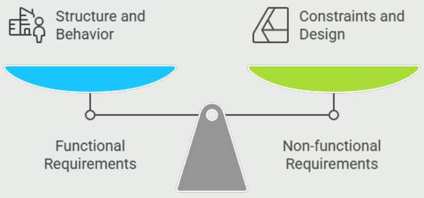


Why This Distinction Matters in System Design

Separating Functional and Non-Functional Requirements helps you design better systems — by guiding architecture, trade-offs, and validation strategies.

Functional Requirements:

- Drive **what needs to be built**
- Map to system structure and behavior (functions, parts, interfaces)
- Used to define logical architecture



Non-Functional Requirements:

- Shape **how the system is evaluated**
- Guide architecture decisions (e.g., redundancy, real-time response, materials)
- Often drive system-wide trade-offs (e.g., cost vs performance, weight vs accuracy)

You build a system to satisfy functional requirements,
but you judge its success by how well it meets non-functional ones.

Functional vs Non-Functional Requirements

Understanding the difference between **functional and non-functional requirements** isn't just academic — it directly influences how your system is designed and validated.

Functional requirements tell you what to build:

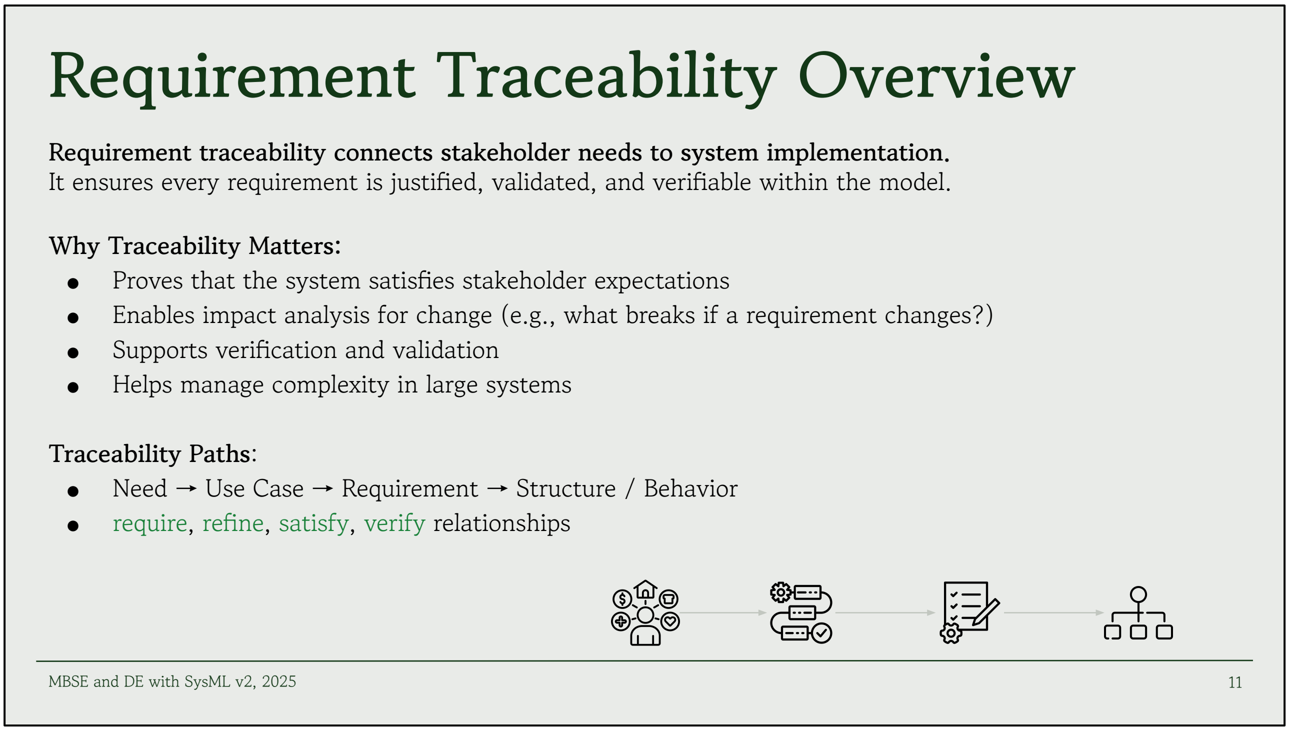
- What components you need
- What actions the system must perform
- How it interacts with users or the environment

Non-functional requirements influence **how you build it**:

- What materials, sensors, or processors are suitable
- Whether you need thermal protection, faster computation, or stronger signal fidelity
- How to balance performance, cost, and safety

So while functional requirements define the **minimum capability**, non-functional requirements often determine **how acceptable the system is** under real-world conditions.

This distinction helps you make smarter design decisions and organize your system model more effectively



Requirement Traceability Overview

Requirement traceability connects stakeholder needs to system implementation.
It ensures every requirement is justified, validated, and verifiable within the model.

Why Traceability Matters:

- Proves that the system satisfies stakeholder expectations
- Enables impact analysis for change (e.g., what breaks if a requirement changes?)
- Supports verification and validation
- Helps manage complexity in large systems

Traceability Paths:

- Need → Use Case → Requirement → Structure / Behavior
- require, refine, satisfy, verify relationships

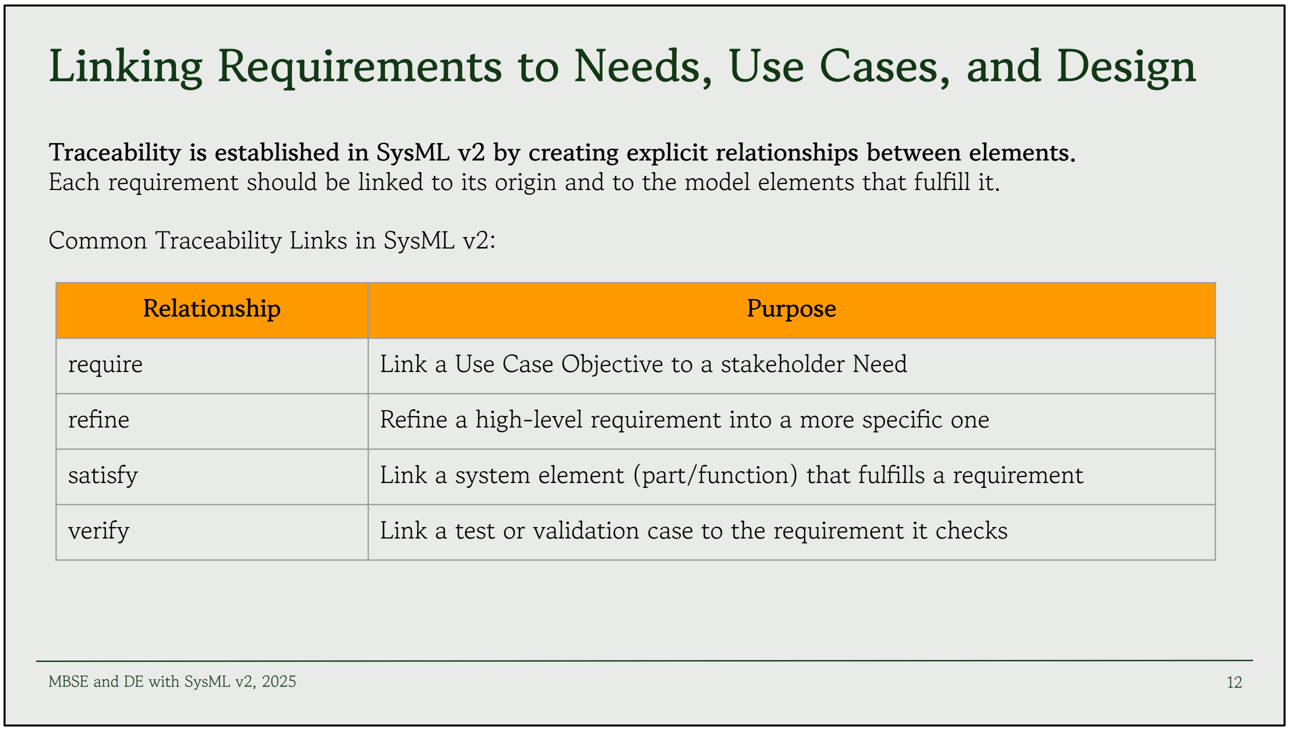
Traceability and Linking Needs
Traceability is one of the most powerful features of SysML modeling.
It gives you a way to **track the origin, justification, and impact** of every requirement in your system — and it ensures that nothing important is left floating or disconnected.
Here's what that means in practice:

- If a stakeholder expresses a need, you model that as a requirement
- You then define a Use Case to fulfill that need
- Next, you write a formal system requirement to implement that Use Case
- Then, you link it to structure or behavior models with satisfy

This chain allows you to:

- Perform **impact analysis** (e.g., if a need changes, what must be redesigned?)
- Validate your model (e.g., can we prove that every requirement is addressed?)
- And maintain full **traceability** across all lifecycle phases

It's a foundational MBSE practice — and one that enables better design decisions, safer systems, and clearer documentation.



Linking Requirements to Needs, Use Cases, and Design

Traceability is established in SysML v2 by creating explicit relationships between elements.
Each requirement should be linked to its origin and to the model elements that fulfill it.

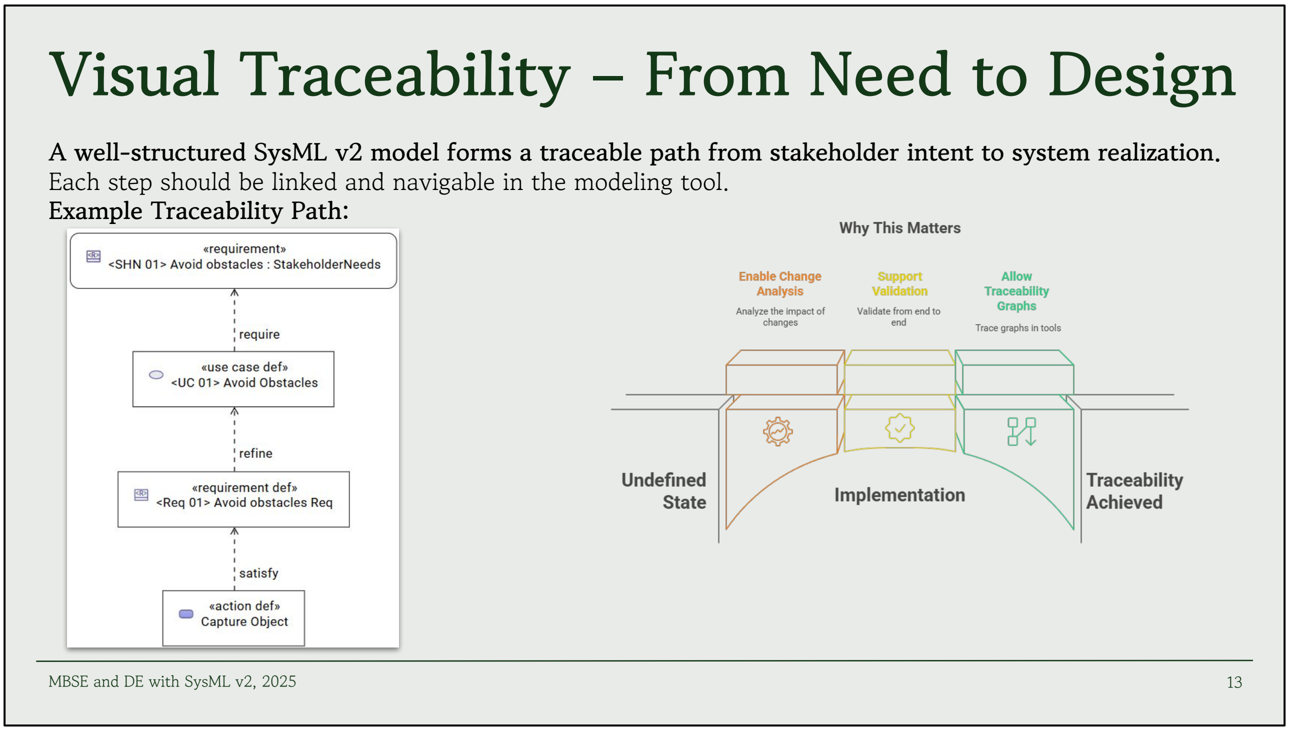
Common Traceability Links in SysML v2:

Relationship	Purpose
require	Link a Use Case Objective to a stakeholder Need
refine	Refine a high-level requirement into a more specific one
satisfy	Link a system element (part/function) that fulfills a requirement
verify	Link a test or validation case to the requirement it checks

Traceability and Linking Needs
Now that we've introduced the concept of traceability, let's look at how it works **in practice** using SysML v2.
SysML gives us a set of **dedicated relationships** to link requirements to the rest of your model:

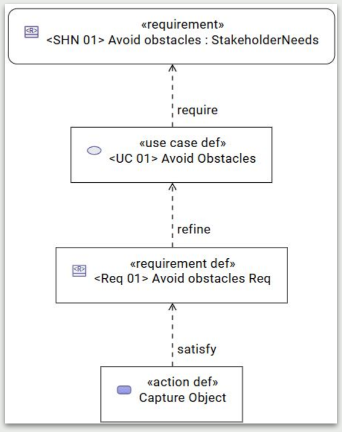
- require shows that a Use Case or Objective is driven by a stakeholder requirement
- refine lets you move from abstract to specific — e.g., going from a Need to a testable spec
- satisfy links a system structure or behavior (like a part or action) to the requirement it fulfills
- verify ties in a test case or simulation model that checks whether the requirement is met

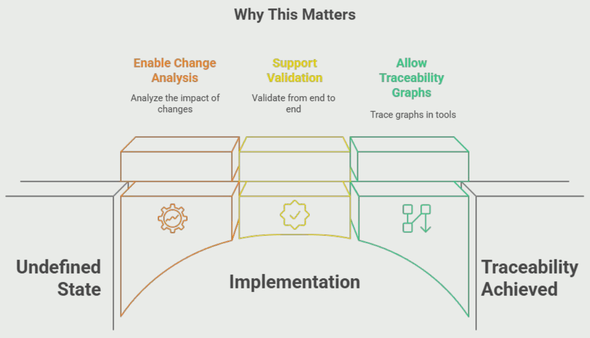
Each of these links strengthens the **digital thread** — turning your model into more than just diagrams. It becomes a fully navigable, auditable trace from **intention to implementation**.
When you build your own drone system, traceability will help ensure that each decision you model supports a real stakeholder goal — and can be verified later.



Visual Traceability – From Need to Design

A well-structured SysML v2 model forms a traceable path from stakeholder intent to system realization.
Each step should be linked and navigable in the modeling tool.
Example Traceability Path:





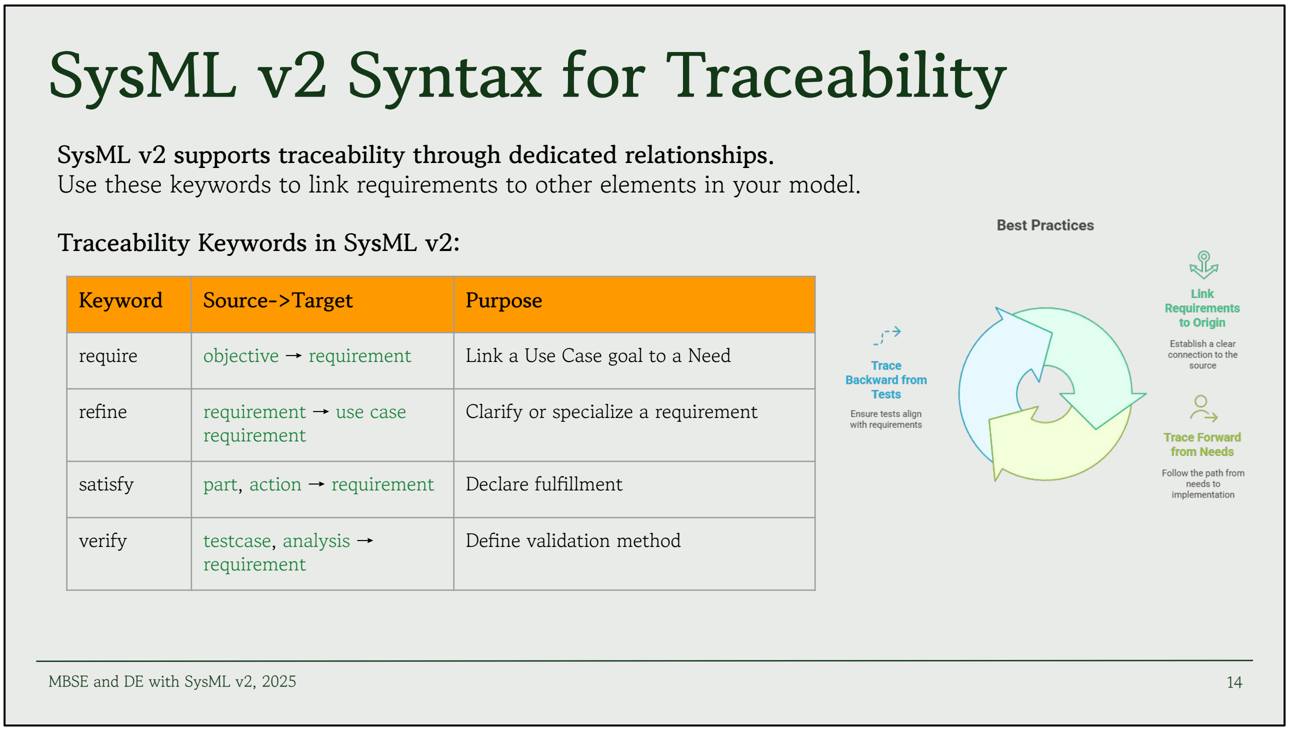
Traceability and Linking Needs
Let's put it all together with a **visual traceability path**.
When you model a system using SysML v2, every modeling layer should **link to the next**:

- A stakeholder **Need** becomes the basis for a **Use Case**
- The Use Case leads to a **formal system requirement**
- That requirement is fulfilled by some part of the **system design**

Each step is modeled using SysML's structured relationships:

- require for objective → need
- refine for requirement → need
- satisfy for system element → requirement

This traceability forms the backbone of your model — enabling auditability, justification for design decisions, and even **automated coverage checks** in tools like SysON.
When changes occur — like a new regulation or updated stakeholder goal — this structure allows you to **trace the impact instantly** across your entire system.

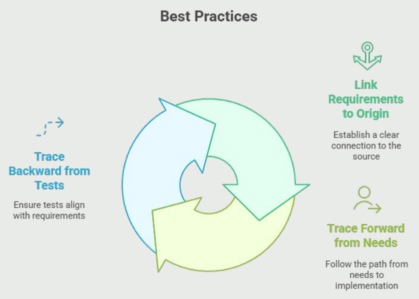


SysML v2 Syntax for Traceability

SysML v2 supports traceability through dedicated relationships.
Use these keywords to link requirements to other elements in your model.

Traceability Keywords in SysML v2:

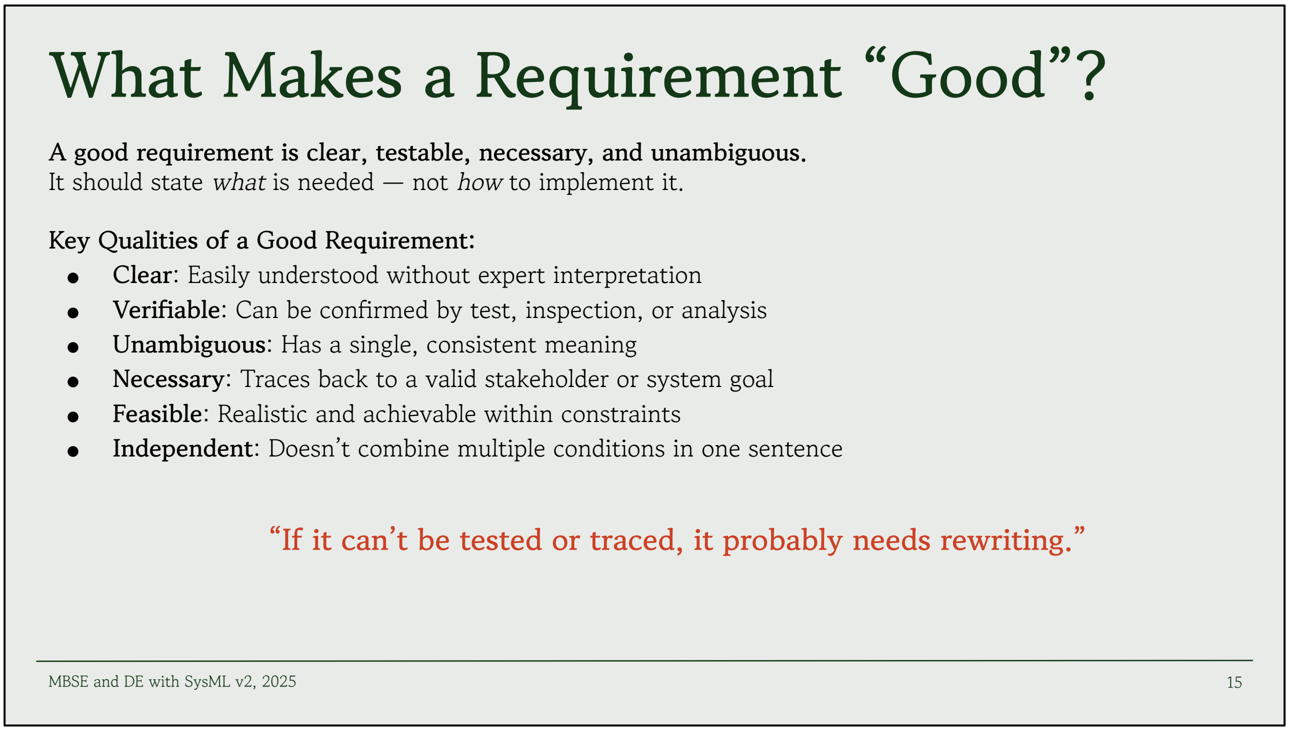
Keyword	Source->Target	Purpose
require	objective → requirement	Link a Use Case goal to a Need
refine	requirement → use case requirement	Clarify or specialize a requirement
satisfy	part, action → requirement	Declare fulfillment
verify	testcase, analysis → requirement	Define validation method


Best Practices
Link Requirements to Origin
Establish a clear connection to the source
Trace Forward from Needs
Follow the path from needs to implementation
Trace Backward from Tests
Ensure tests align with requirements

Traceability and Linking Needs
Let's summarize the key **traceability relationships** you'll use in SysML v2.
Each relationship is **typed** — it's not just a line on a diagram. It means something specific to the model:

- require connects **Use Case objectives** to **Needs**
- refine expresses that one requirement makes another more detailed or testable
- satisfy declares that some model element — a part, an action — fulfills a requirement
- verify ties in a **test or analysis model** that proves a requirement is met

When you model traceability in SysON, use these keywords in the relationship panel or directly in your diagram.
This gives you not just **clarity** — but **automated trace graphs**, **coverage reports**, and **change impact analysis**. It's how real MBSE teams manage complexity and ensure completeness.



What Makes a Requirement “Good”?

A good requirement is clear, testable, necessary, and unambiguous.
It should state *what* is needed — not *how* to implement it.

Key Qualities of a Good Requirement:

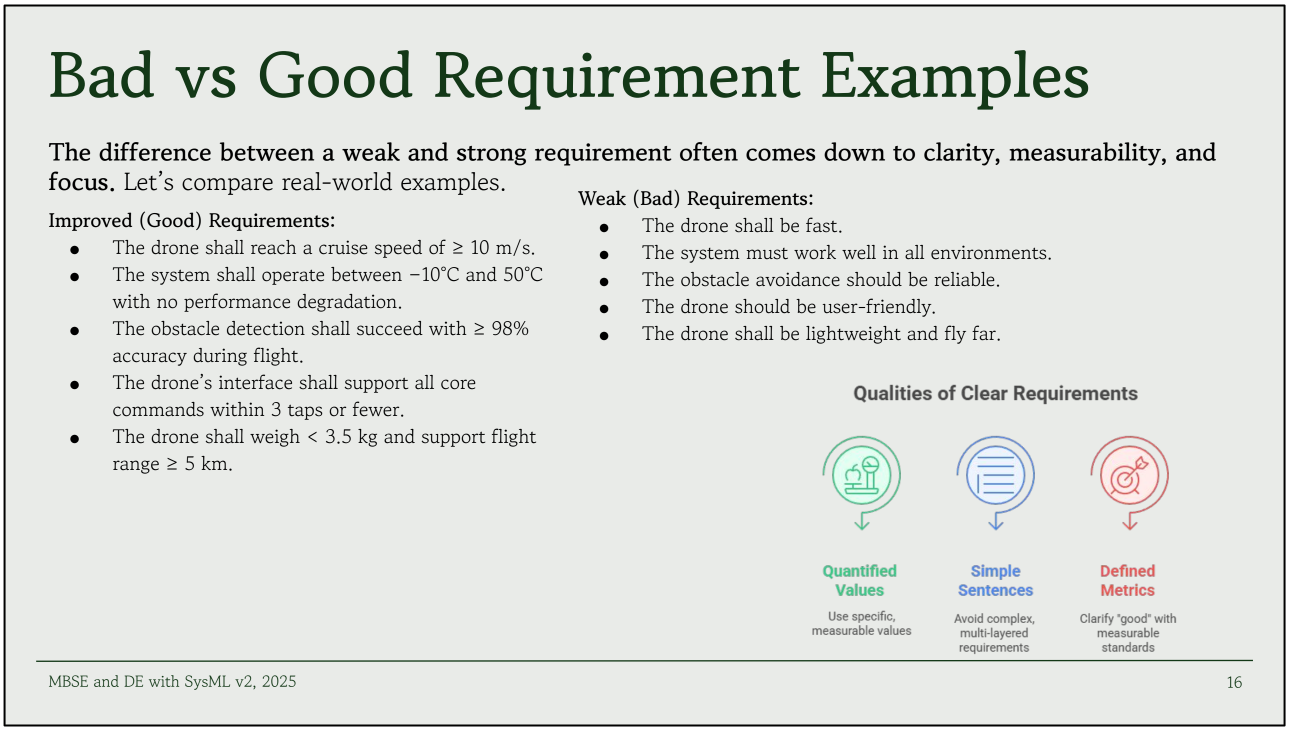
- **Clear**: Easily understood without expert interpretation
- **Verifiable**: Can be confirmed by test, inspection, or analysis
- **Unambiguous**: Has a single, consistent meaning
- **Necessary**: Traces back to a valid stakeholder or system goal
- **Feasible**: Realistic and achievable within constraints
- **Independent**: Doesn’t combine multiple conditions in one sentence

“If it can’t be tested or traced, it probably needs rewriting.”

Writing Good Requirements
Before you write any requirement, ask yourself: “Is this statement clear, necessary, and testable?”
A **good requirement** is not just about language — it’s about being **useful to the model and team**.
It must be:

- **Clear** — so all readers understand it the same way
- **Verifiable** — so you can prove whether the system meets it
- **Unambiguous** — so there’s no room for multiple interpretations
- **Necessary** — so it connects to a real stakeholder value or system function
- **Feasible** — so the team can actually implement and test it
- **Independent** — so each requirement can be managed and tested on its own

You’ll evaluate your own Drone System requirements against these criteria. If a statement is too vague or too complex — it’s a sign to refine.



Bad vs Good Requirement Examples

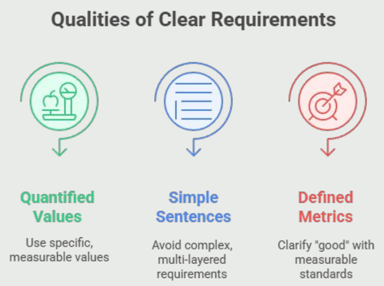
The difference between a weak and strong requirement often comes down to clarity, measurability, and focus. Let's compare real-world examples.

Improved (Good) Requirements:

- The drone shall reach a cruise speed of ≥ 10 m/s.
- The system shall operate between −10°C and 50°C with no performance degradation.
- The obstacle detection shall succeed with ≥ 98% accuracy during flight.
- The drone's interface shall support all core commands within 3 taps or fewer.
- The drone shall weigh < 3.5 kg and support flight range ≥ 5 km.

Weak (Bad) Requirements:

- The drone shall be fast.
- The system must work well in all environments.
- The obstacle avoidance should be reliable.
- The drone should be user-friendly.
- The drone shall be lightweight and fly far.


Qualities of Clear Requirements
Quantified Values
Use specific, measurable values
Simple Sentences
Avoid complex, multi-layered requirements
Defined Metrics
Clarify "good" with measurable standards

Writing Good Requirements

Let's look at some **bad vs good requirement examples.**

Bad requirements are often:

- Vague ("fast," "reliable," "user-friendly")
- Not measurable
- Combining multiple ideas in one sentence

Good requirements, in contrast:

- Use **specific, testable values**
- Describe one thing per statement
- Match what you can implement or verify

For example:

"The drone shall be fast."

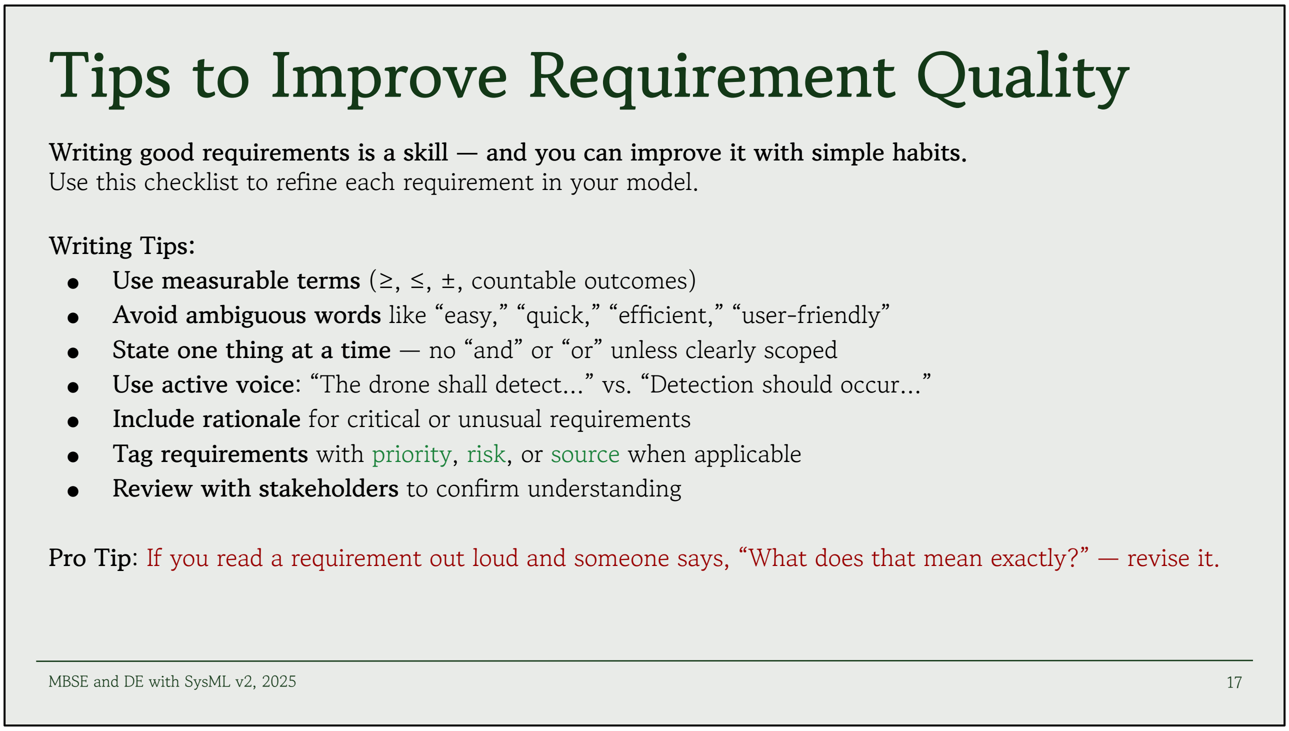
✕ What is fast? To whom? In what context?

We revise it:

"The drone shall cruise at ≥ 10 meters per second."

Now it's measurable and unambiguous.

Use this mindset when writing your own Drone System requirements — and apply it to every line.



Tips to Improve Requirement Quality

Writing good requirements is a skill — and you can improve it with simple habits.
Use this checklist to refine each requirement in your model.

Writing Tips:

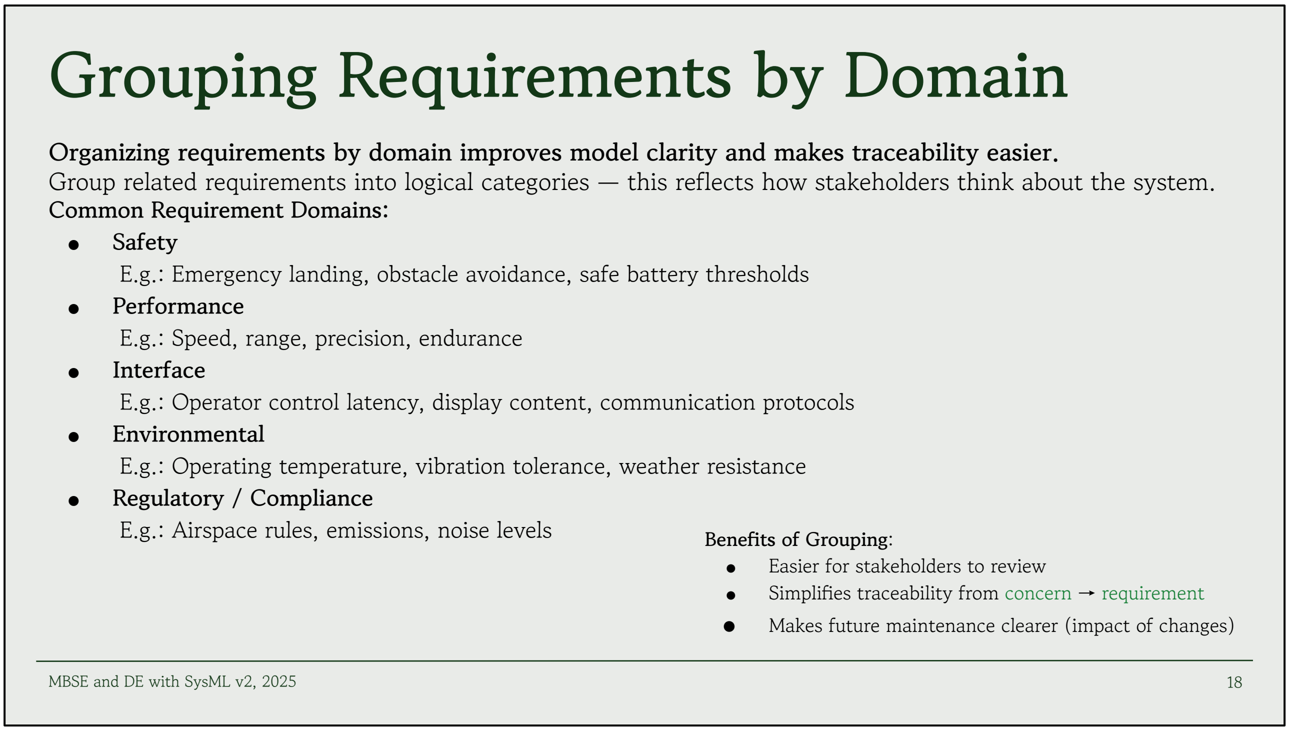
- **Use measurable terms** (≥, ≤, ±, countable outcomes)
- **Avoid ambiguous words** like "easy," "quick," "efficient," "user-friendly"
- **State one thing at a time** — no "and" or "or" unless clearly scoped
- **Use active voice**: "The drone shall detect..." vs. "Detection should occur..."
- **Include rationale** for critical or unusual requirements
- **Tag requirements** with priority, risk, or source when applicable
- **Review with stakeholders** to confirm understanding

Pro Tip: If you read a requirement out loud and someone says, "What does that mean exactly?" — revise it.

Writing Good Requirements
As you begin to write and refine your Drone System requirements, keep these simple but powerful tips in mind.

- Use **measurable values** — this makes requirements testable and objective.
- Avoid vague or subjective language — "easy," "fast," "safe" all sound nice but mean different things to different people.
- Make sure each requirement is **focused** on just one expectation. If you see an "and" or "or" — ask whether it should be split.
- Use **active voice** and specific actions.
- Add rationale when something might not be obvious — this helps downstream engineers or testers understand *why* a requirement matters.
- Finally, if possible, review requirements with the stakeholder who originated the goal. Ask: "Does this say what you intended?"

If the answer is "not quite" — revise until it does.



Grouping Requirements by Domain

Organizing requirements by domain improves model clarity and makes traceability easier.
Group related requirements into logical categories — this reflects how stakeholders think about the system.
Common Requirement Domains:

- **Safety**
  E.g.: Emergency landing, obstacle avoidance, safe battery thresholds
- **Performance**
  E.g.: Speed, range, precision, endurance
- **Interface**
  E.g.: Operator control latency, display content, communication protocols
- **Environmental**
  E.g.: Operating temperature, vibration tolerance, weather resistance
- **Regulatory / Compliance**
  E.g.: Airspace rules, emissions, noise levels

Benefits of Grouping:

- Easier for stakeholders to review
- Simplifies traceability from concern → requirement
- Makes future maintenance clearer (impact of changes)

Organizing and Structuring Requirements
As your system model grows, so will your list of requirements.
If you leave them in one flat list — it quickly becomes hard to navigate, trace, and review.
That's why we **group requirements into logical domains** — like Safety, Performance, Interface, Environmental.
This:

- Matches how stakeholders think about the system
- Makes **traceability clearer** (especially to stakeholder concerns)
- Helps when doing **impact analysis** later — for example, if a regulation changes, you know which compliance requirements to revisit

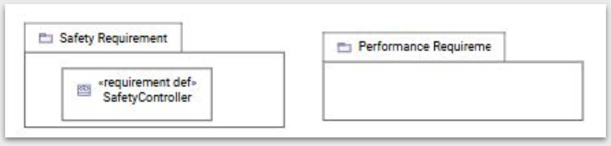
In SysML v2, you can organize requirements in **packages** or sub-packages — and use model queries or views to filter by domain when needed.
For your Drone System project, think about which domains are most critical, and start grouping your requirements accordingly.

Structuring Requirements in the Model

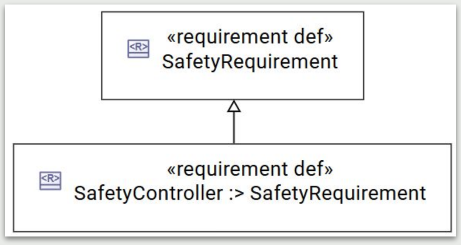
You can structure requirements using hierarchy, packages, or both.
This helps maintain clarity and traceability as your system evolves.

Two Common Approaches:

- Package-based Organization:



- **Hierarchy using** requirement def **specialization**:

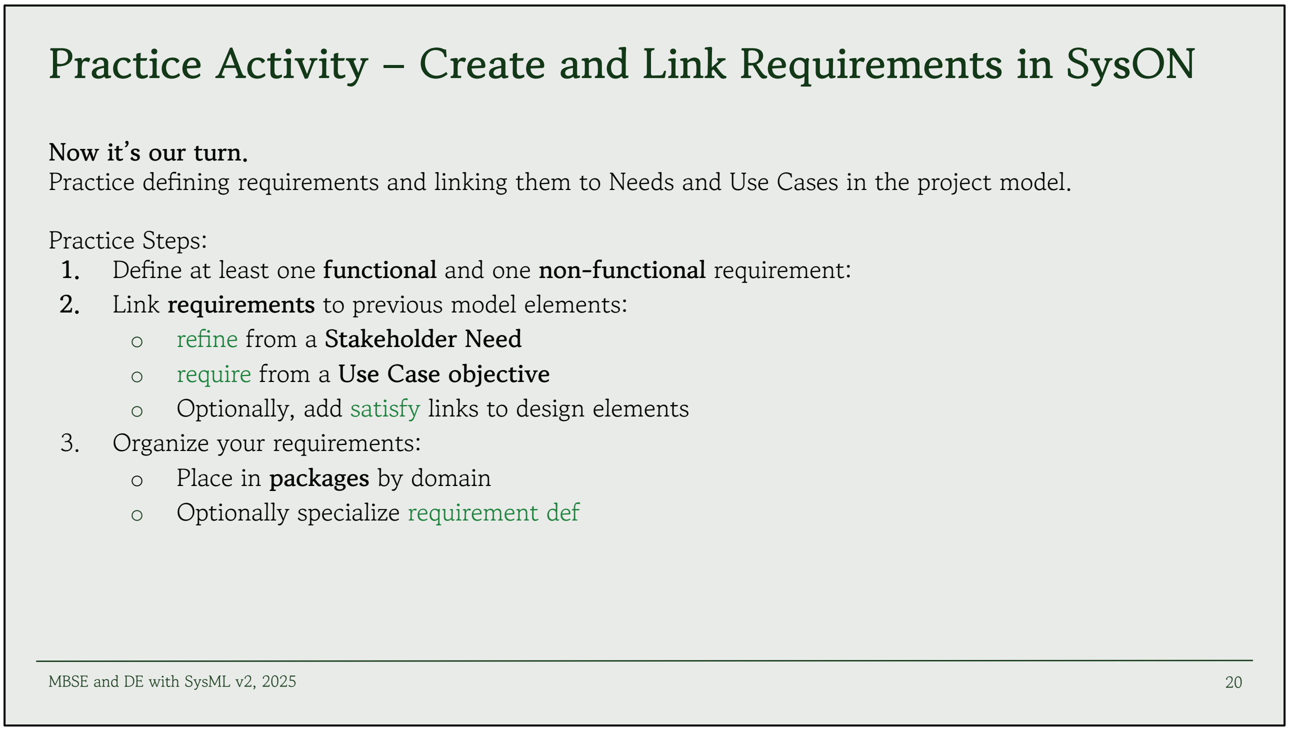


Best Practice: Combine **package organization** (for navigation) with **type specialization** (for semantic clarity).

Organizing and Structuring Requirements
Once you've grouped your requirements conceptually, you need to **structure them in the model**.
There are two main ways to do this:
1 Use **packages** — simple folders in the model to group related requirements. This is great for navigation and reporting.
2 Use **specialization hierarchy** — define a generic Requirement, then specialize into SafetyRequirement, PerformanceRequirement, and so on. This adds **semantic meaning** and helps enforce consistency.
In most MBSE practice — including for your Drone System — we use **both**:

- Packages for visibility and modularity
- Specialized requirement def types to make the model smarter and easier to query

This combination gives you the best of both worlds: clarity for human readers, and power for the modeling tool to analyze relationships.



Practice Activity – Create and Link Requirements in SysON

Now it's our turn.
Practice defining requirements and linking them to Needs and Use Cases in the project model.

Practice Steps:

1. Define at least one **functional** and one **non-functional** requirement:
2. Link **requirements** to previous model elements:
   - refine from a **Stakeholder Need**
   - require from a **Use Case objective**
   - Optionally, add satisfy links to design elements
3. Organize your requirements:
   - Place in **packages** by domain
   - Optionally specialize requirement def

Practice and Project Integration
Now it's time to practice what you've learned — for your Drone System project model.
Here's your goal for this exercise:

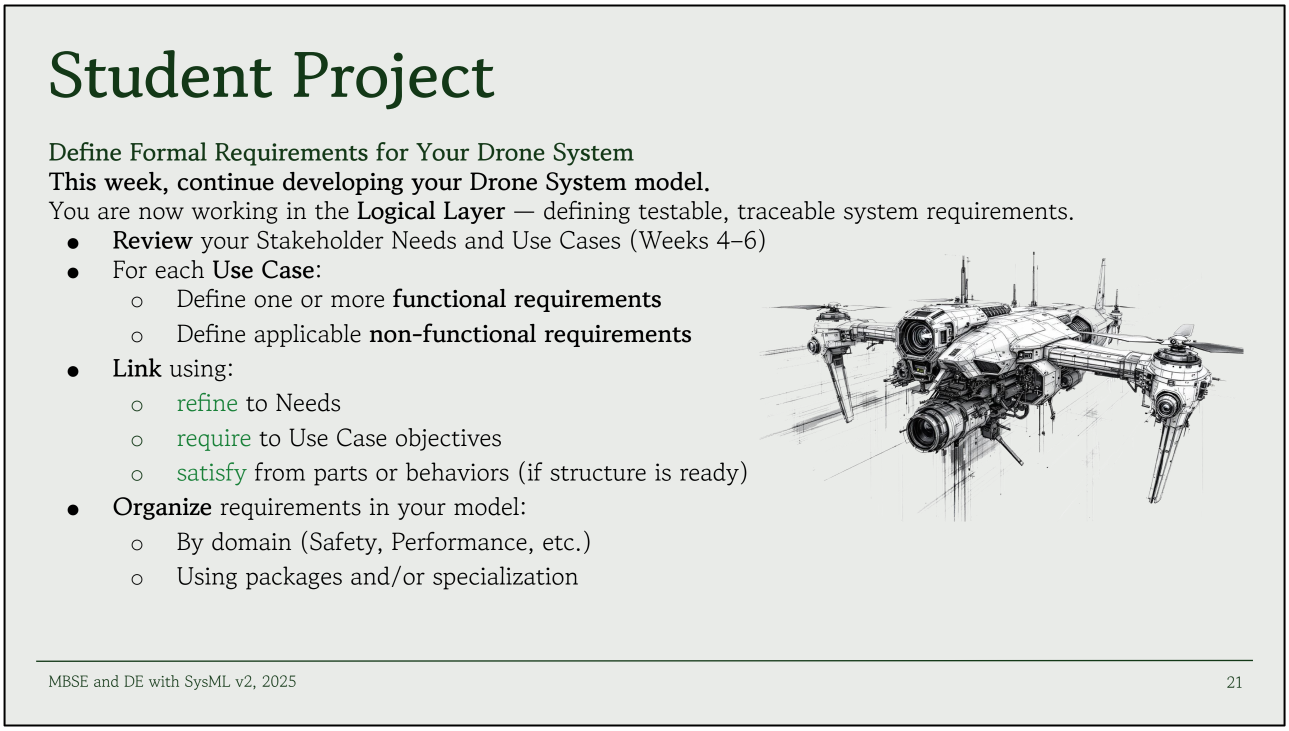
- Define at least one **functional requirement** — something the system must do.
- Define at least one **non-functional requirement** — a performance or quality constraint.

Next, establish traceability:

- Link your requirement back to the original **Stakeholder Need** using refine.
- Link from a **Use Case objective** using require.
- Optionally, link to parts or behaviors using satisfy.

Finally, structure your model:

- Organize requirements into **packages** by domain — Safety, Performance, Interface, etc.
- Optionally specialize your requirement def types for semantic clarity.



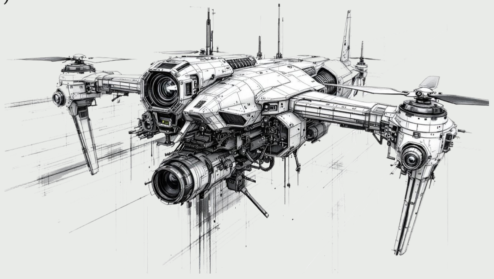
Student Project

Define Formal Requirements for Your Drone System

This week, continue developing your Drone System model.

You are now working in the **Logical Layer** — defining testable, traceable system requirements.

- **Review** your Stakeholder Needs and Use Cases (Weeks 4–6)
- For each **Use Case**:
  - Define one or more **functional requirements**
  - Define applicable **non-functional requirements**
- **Link** using:
  - refine to Needs
  - require to Use Case objectives
  - satisfy from parts or behaviors (if structure is ready)
- **Organize** requirements in your model:
  - By domain (Safety, Performance, etc.)
  - Using packages and/or specialization



Practice and Project Integration

This week in your Drone System project, you are formally entering the **Logical Layer**.

That means:

- Moving beyond general stakeholder intent
- Defining **specific, testable system requirements**

Here's how to proceed:

- Start by reviewing your existing Stakeholder Needs and Use Cases.
- For each Use Case, ask:
  - What **functional requirements** does this drive?
  - What **non-functional requirements** apply to this scenario?

Define your requirements using requirement def and requirement.

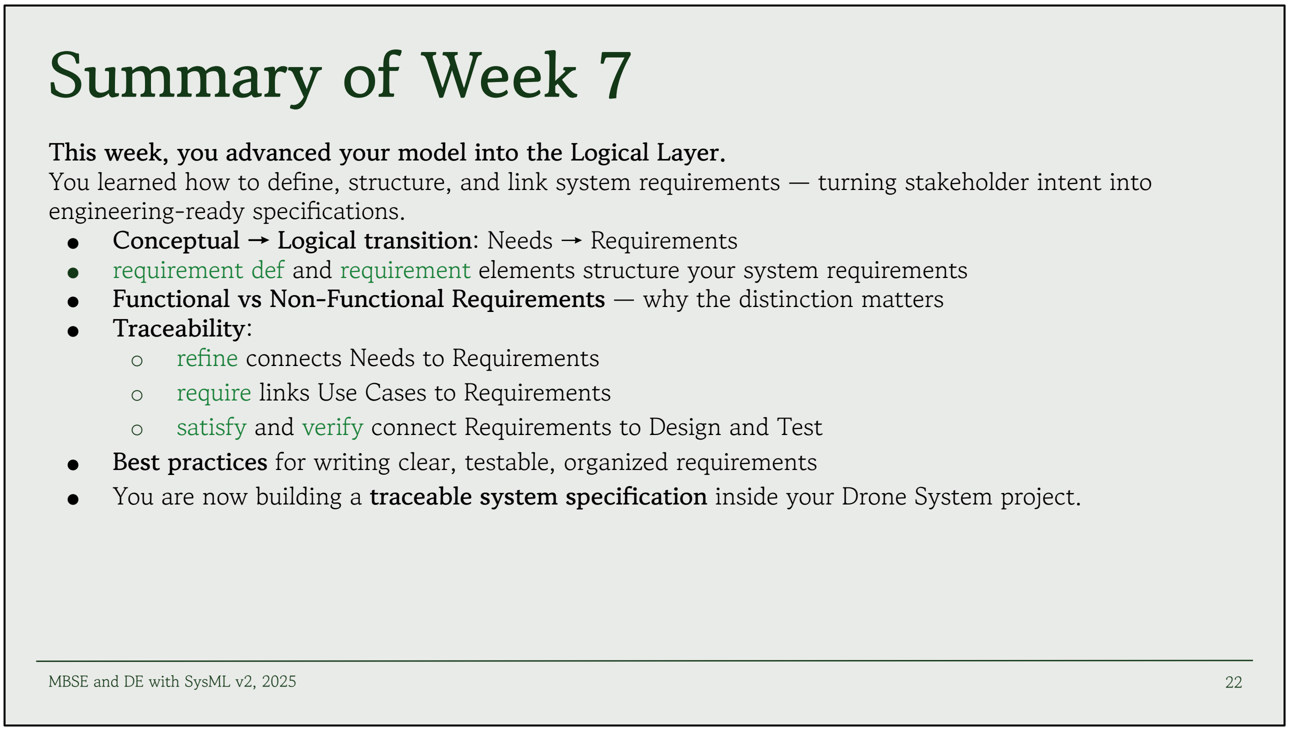
Link them:

- Back to the Need with refine
- From the Use Case objective with require
- To system parts/behaviors (if modeled) using satisfy

Finally, organize your requirements:

- By **domain** (Safety, Performance, etc.)
- Using **packages** and/or specialized requirement types

You are now building the foundation that supports design, validation, and certification — this is the heart of the MBSE approach.



Summary of Week 7

This week, you advanced your model into the Logical Layer.
You learned how to define, structure, and link system requirements — turning stakeholder intent into engineering-ready specifications.

- **Conceptual → Logical transition**: Needs → Requirements
- requirement def and requirement elements structure your system requirements
- **Functional vs Non-Functional Requirements** — why the distinction matters
- **Traceability**:
    - refine connects Needs to Requirements
    - require links Use Cases to Requirements
    - satisfy and verify connect Requirements to Design and Test
- **Best practices** for writing clear, testable, organized requirements
- You are now building a **traceable system specification** inside your Drone System project.

Practice and Project Integration
This week you've reached a major milestone in your system model — you are now working in the **Logical Layer**.
You've learned how to:

- Transform Stakeholder Needs into **formal Requirements**
- Distinguish between **functional** and **non-functional** requirements
- Use **traceability links** to maintain model integrity
- Apply **best practices** to write clear, testable statements
- Organize your requirements for clarity and scalability

Your Drone System model now contains a traceable, engineering-ready specification — and this Logical Layer will drive your design and validation going forward.
Next week, we'll continue this progression by modeling **Interfaces and Ports** — the connections that make your system elements interact.
For now, focus on refining and linking your requirements — and enjoy this key achievement in your MBSE journey.

QUESTION!