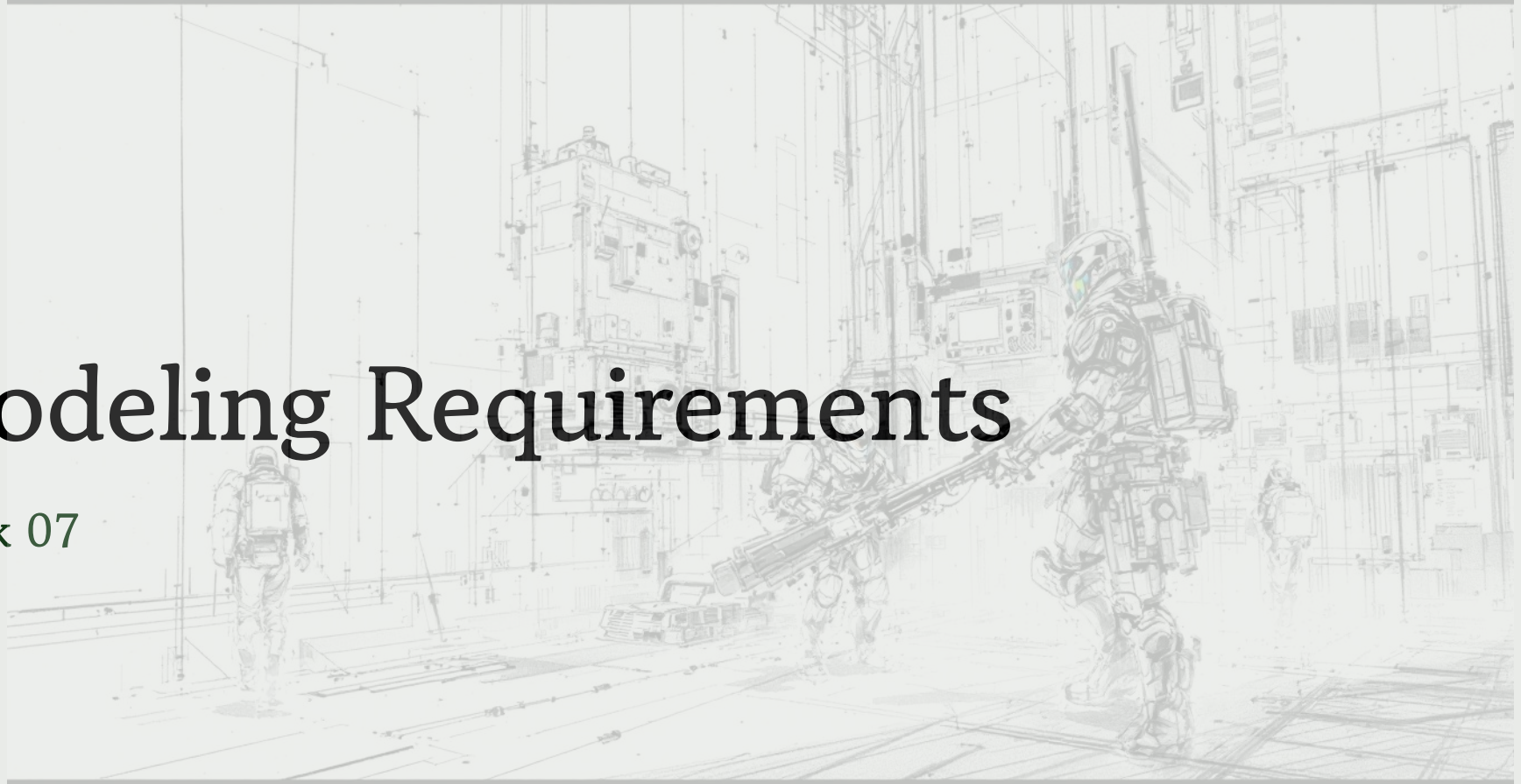
Modeling Requirements

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 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.

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

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;
```

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

Two Types of Requirements – Functional vs Non-Functional

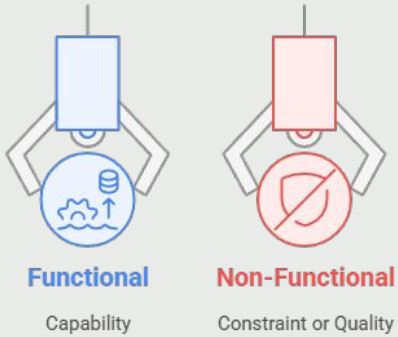
Requirements describe either system behavior (functional) or system quality/performance (non-functional). Understanding this distinction is key to building a complete and balanced specification.

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%.

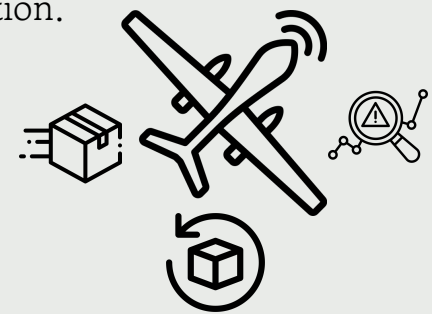


Examples of Functional Requirements

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:

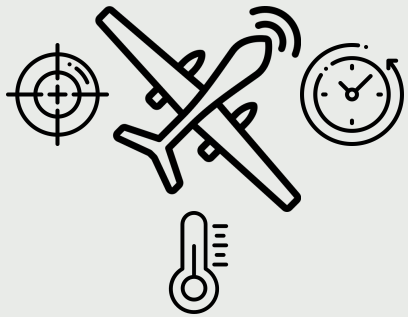
- Derived from Use Cases, scenarios, and system interactions
- Aligned with Stakeholder Needs and refined into requirement

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

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.

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

Linking Requirements to Needs, Use Cases, and Design

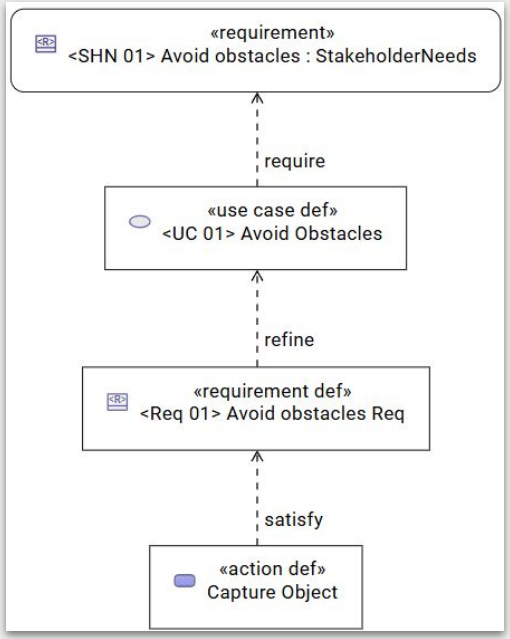
Traceability is established in SysML v2 by creating explicit relationships between elements.
Each requirement should be linked to its origin and to the model elements that fulfill it.

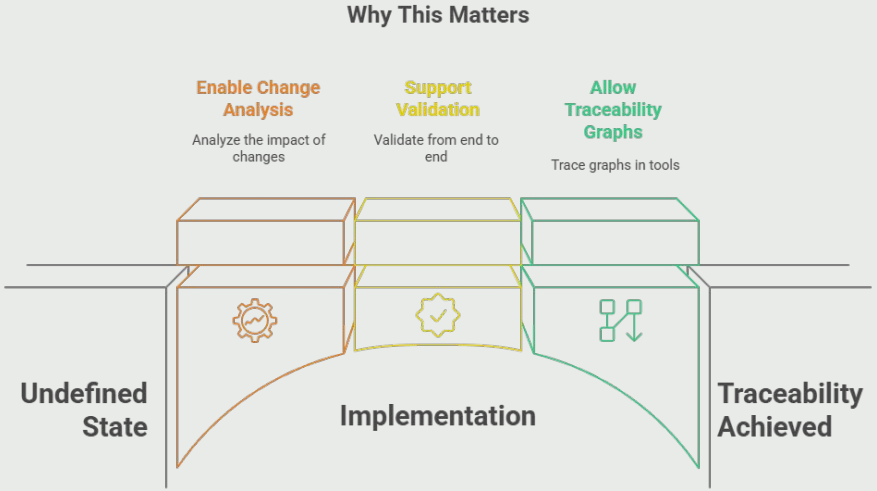
Common Traceability Links in SysML v2:

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

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:



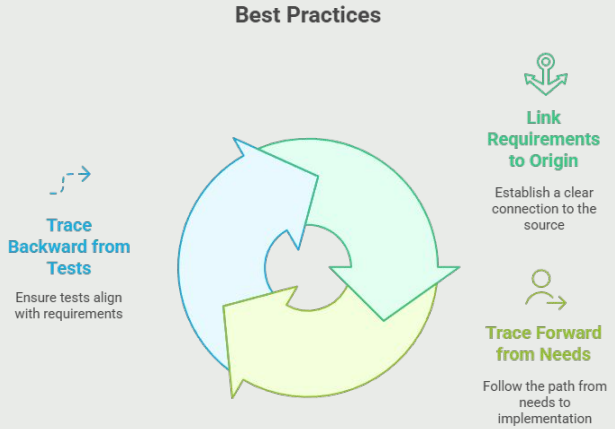


SysML v2 Syntax for Traceability

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



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.”

Bad vs Good Requirement Examples

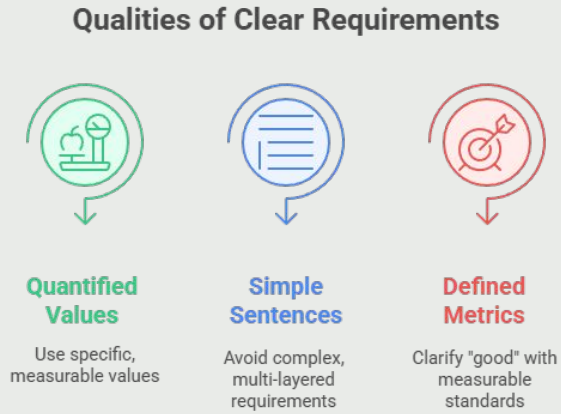
The difference between a weak and strong requirement often comes down to clarity, measurability, and focus. Let's compare real-world examples.

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.



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.

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)

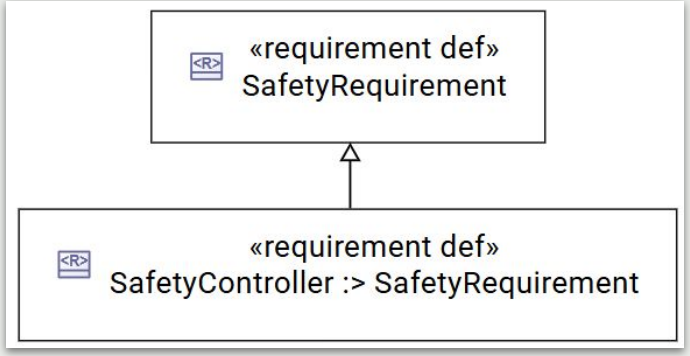
Structuring Requirements in the Model

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).

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

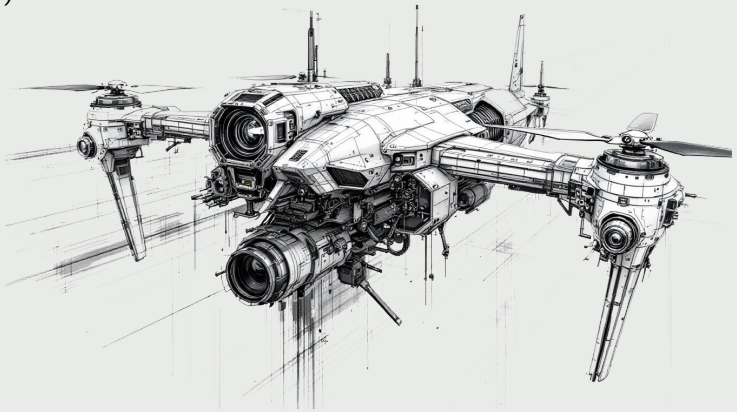
Student Project

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



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.

QUESTION!