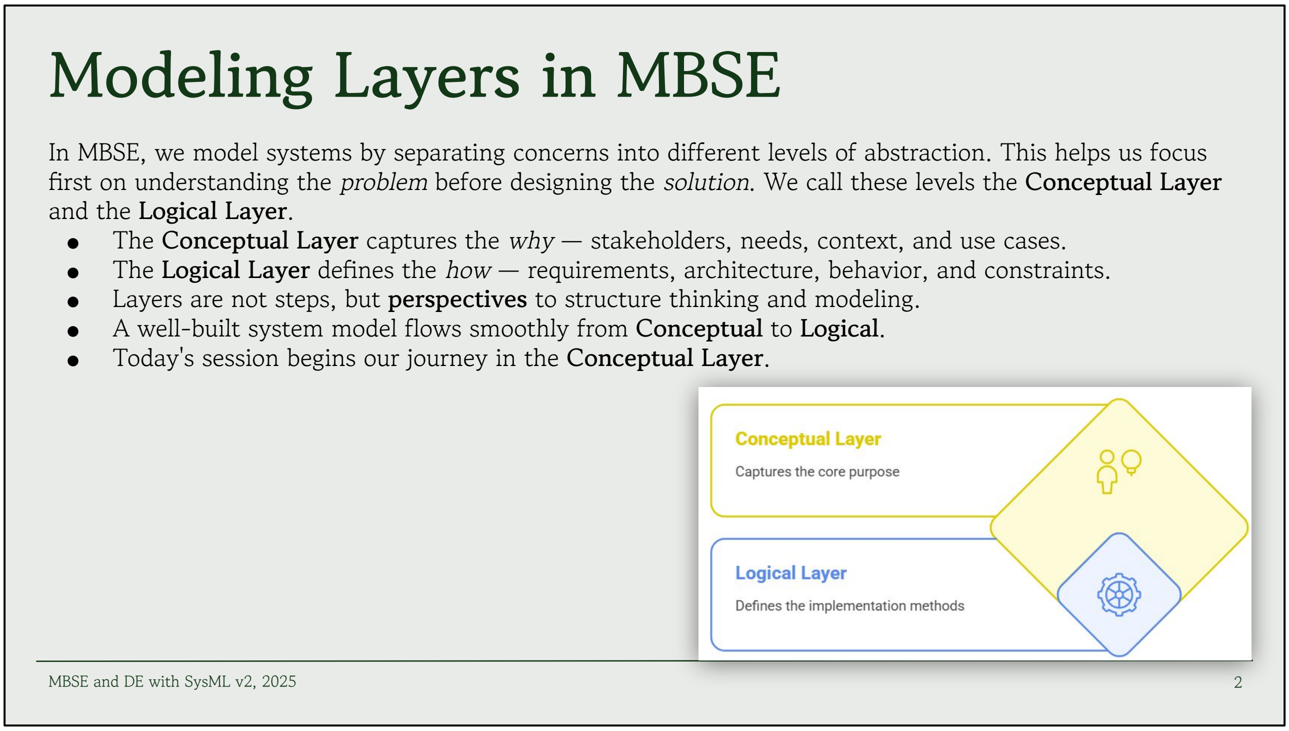
Modeling Stakeholder Needs and Concerns

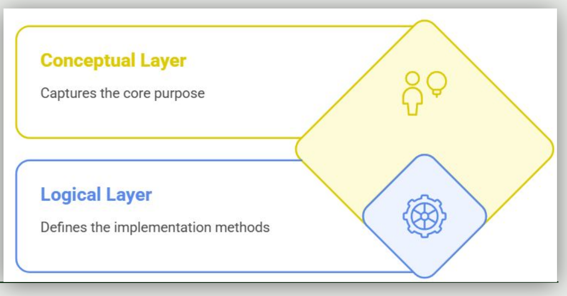
Week 04



Modeling Layers in MBSE

In MBSE, we model systems by separating concerns into different levels of abstraction. This helps us focus first on understanding the *problem* before designing the *solution*. We call these levels the **Conceptual Layer** and the **Logical Layer**.

- The **Conceptual Layer** captures the *why* — stakeholders, needs, context, and use cases.
- The **Logical Layer** defines the *how* — requirements, architecture, behavior, and constraints.
- Layers are not steps, but **perspectives** to structure thinking and modeling.
- A well-built system model flows smoothly from **Conceptual** to **Logical**.
- Today's session begins our journey in the **Conceptual Layer**.



Modeling Layers in MBSE – A System Thinking Perspective

In Model-Based Systems Engineering, one of the most important concepts is the idea of **layers of abstraction** — levels of thinking that help us separate *what the system is meant to achieve* from *how the system will actually achieve it*.

We call these two levels the **Conceptual Layer** and the **Logical Layer**.

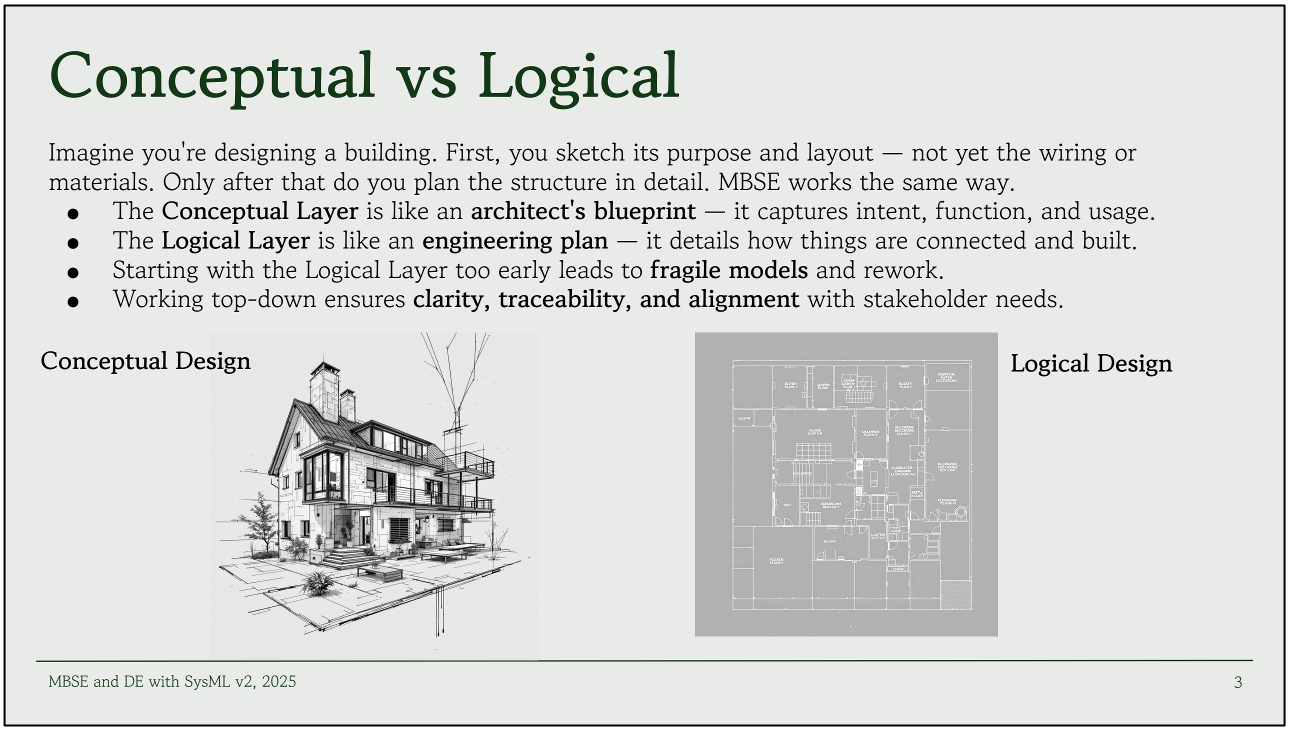
The **Conceptual Layer** is about **understanding the problem space** — this is where we identify *who* the stakeholders are, *what* they care about, and *what goals* the system must fulfill. We also explore *how the system will interact with its environment* and *what kinds of situations it will face*. This includes capturing **stakeholder needs**, defining **use cases**, and understanding the **system context**. At this layer, we're not concerned with architecture, behavior models, or interfaces. We're simply trying to deeply understand *why the system exists* and *what outcomes it must support*.

On the other hand, the **Logical Layer** is where we begin **engineering the solution**. Once we have a clear understanding of needs and usage context, we can translate those into **formal system requirements**. From there, we design the system's **structure**, define its **behavior**, and model how it handles **performance, constraints, and analysis**. This is where architecture, behavior models, and parametric models live.

Throughout this course, you'll be working in **both layers**. Sometimes we'll be focusing entirely on the Conceptual Layer. Other times we'll switch to Logical. And often, we'll bridge between them — for example, when we trace a stakeholder need to a requirement.

The key thing to remember is this: these layers are not steps in a checklist. They're **perspectives** — ways of looking at a system from different levels of abstraction. If you mix them too early, your model will become confusing and fragile. But if you work carefully through each layer, your system model will be grounded, traceable, and reliable.

So today — as we begin our **first real modeling session** — we start at the top: the **Conceptual Layer**. This is where every good system model begins.

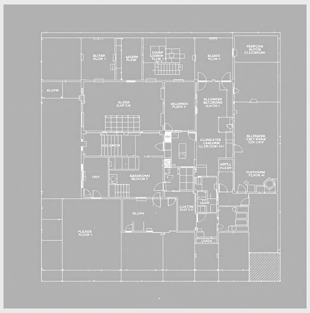


Conceptual vs Logical

Imagine you're designing a building. First, you sketch its purpose and layout — not yet the wiring or materials. Only after that do you plan the structure in detail. MBSE works the same way.

- The **Conceptual Layer** is like an **architect's blueprint** — it captures intent, function, and usage.
- The **Logical Layer** is like an **engineering plan** — it details how things are connected and built.
- Starting with the Logical Layer too early leads to **fragile models** and rework.
- Working top-down ensures **clarity, traceability, and alignment** with stakeholder needs.

Conceptual Design



Logical Design

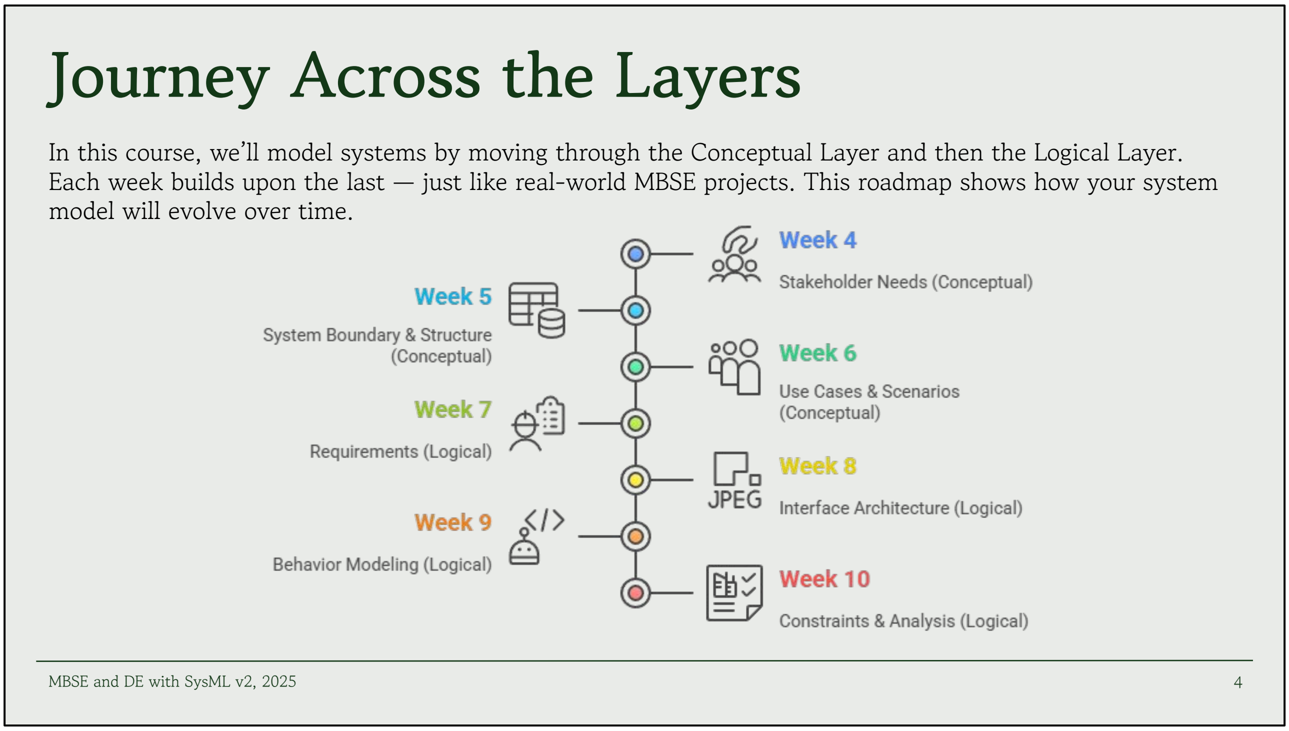
Conceptual vs Logical – Think Like a Systems Architect

Let’s think like system architects. When an architect starts designing a building, they don’t immediately model the plumbing or wiring. They begin with the **concept** — What is the building for? How will people move through it? What functions must it support? This is the **Conceptual Layer** — focused on understanding purpose and use.

Once that’s defined, engineers can work out the **details**: dimensions, support structures, utilities. This is the **Logical Layer** — the blueprint becomes an actionable construction plan.

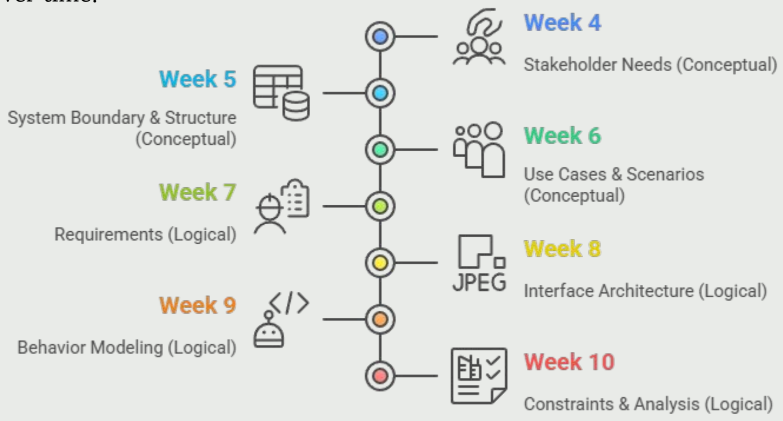
In MBSE, we do the same. If you start with structure and behavior before understanding needs and usage context, you’ll end up building the wrong thing, or worse, something unbuildable.

That’s why we model in layers — to keep our thinking clean, structured, and grounded in real-world purpose.



Journey Across the Layers

In this course, we'll model systems by moving through the Conceptual Layer and then the Logical Layer. Each week builds upon the last — just like real-world MBSE projects. This roadmap shows how your system model will evolve over time.


Week 4
Stakeholder Needs (Conceptual)
Week 5
System Boundary & Structure (Conceptual)
Week 6
Use Cases & Scenarios (Conceptual)
Week 7
Requirements (Logical)
Week 8
JPEG
Interface Architecture (Logical)
Week 9
</>
Behavior Modeling (Logical)
Week 10
Constraints & Analysis (Logical)

Our Journey Across the Layers

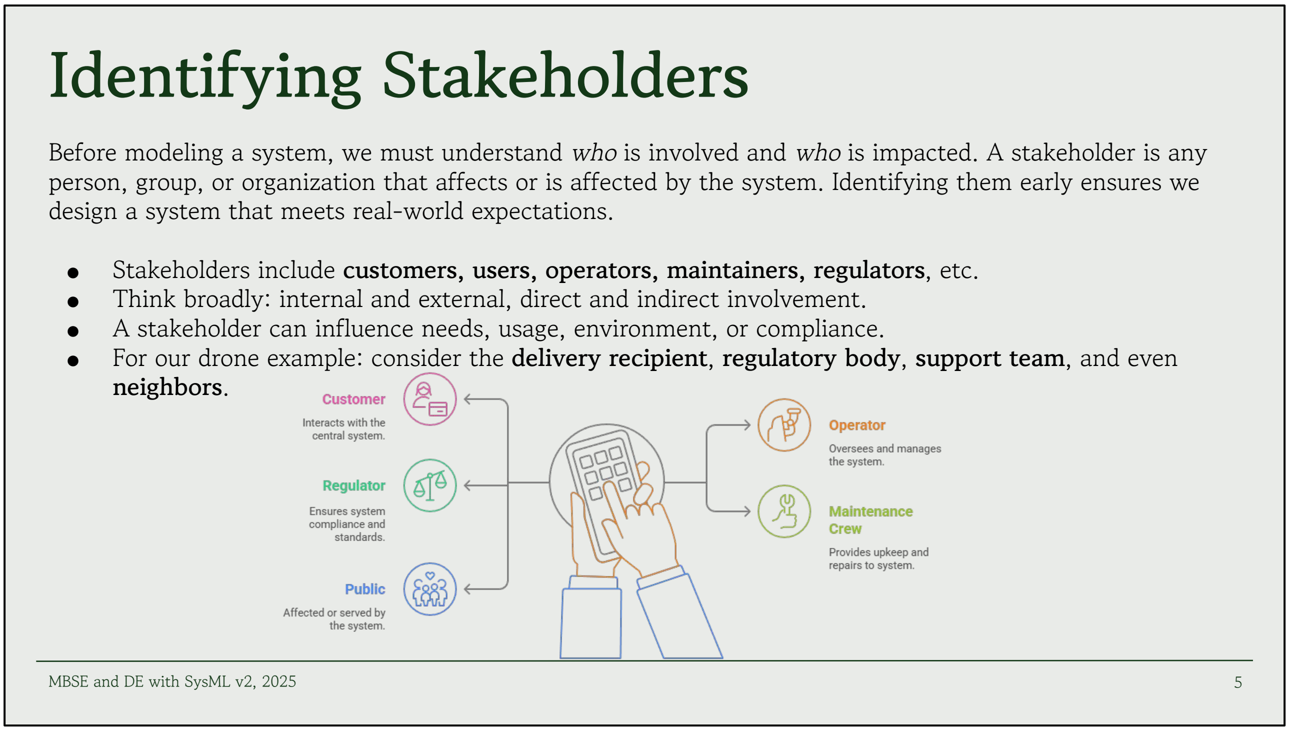
This slide is your roadmap. It shows how each week fits into the larger picture of system modeling.

Weeks 4 to 6 are fully focused on the **Conceptual Layer** — this is where we define the system from the outside-in: Who needs it? Why? In what environment? What situations will it face?

Once we have a clear picture of the problem space, we transition in Week 7 to the **Logical Layer**. From that point forward, we design how the system should behave, how it will be structured internally, and how we analyze it for performance or safety.

Think of each week as building on the one before. By the end of Week 10, you'll have a complete multi-layered model of your system — grounded in stakeholder needs, shaped by use cases, and validated through structure, behavior, and constraints.

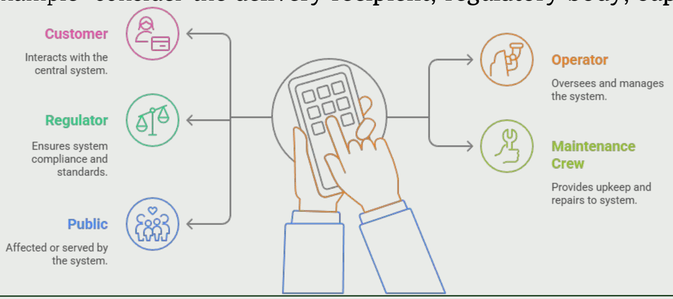
Let's begin that journey — today — with **Stakeholder Needs**.



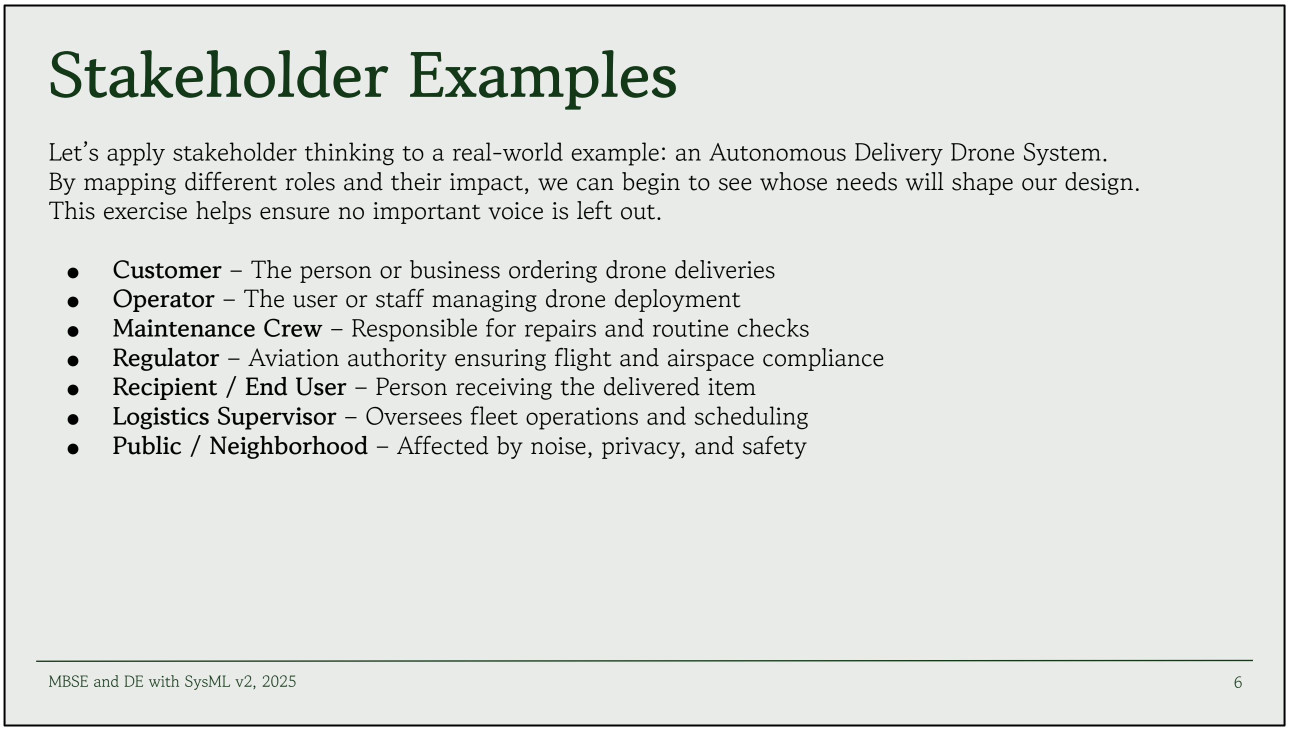
Identifying Stakeholders

Before modeling a system, we must understand *who* is involved and *who* is impacted. A stakeholder is any person, group, or organization that affects or is affected by the system. Identifying them early ensures we design a system that meets real-world expectations.

- Stakeholders include **customers, users, operators, maintainers, regulators**, etc.
- Think broadly: internal and external, direct and indirect involvement.
- A stakeholder can influence needs, usage, environment, or compliance.
- For our drone example: consider the **delivery recipient**, **regulatory body**, **support team**, and even **neighbors**.


Customer
Interacts with the central system.
Regulator
Ensures system compliance and standards.
Public
Affected or served by the system.
Operator
Oversees and manages the system.
Maintenance Crew
Provides upkeep and repairs to system.

Identifying Stakeholders
Our first step in building any system model is identifying the **stakeholders**.
These are people, groups, or institutions who either interact with the system or are influenced by its performance, behavior, or even its existence.
You'll often think of obvious ones like the **customer** or **end user** — but in MBSE we go further.
For our **Autonomous Delivery Drone System**, we must also consider the **operator or pilot**, the **maintenance crew** who will service the drones, the **regulatory authority** that governs airspace, and even **citizens living under the drone's flight path**.Why is this important? Because each of these stakeholders will have **different concerns** and **different needs**. If we miss one now, we might end up designing a system that fails later — not because the technology is wrong, but because we didn't consider *who we were building for*.
In the next step, we'll categorize these stakeholders to make their roles even clearer.

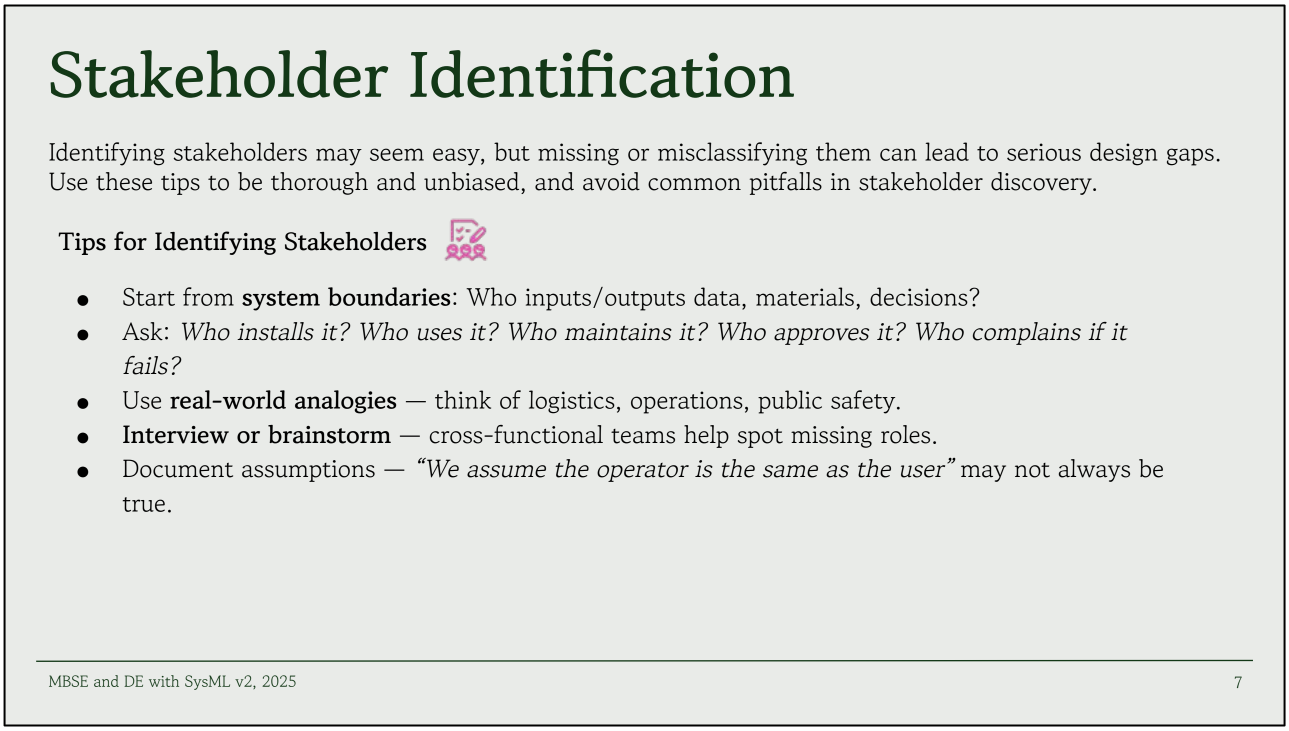


Stakeholder Examples

Let’s apply stakeholder thinking to a real-world example: an Autonomous Delivery Drone System.
By mapping different roles and their impact, we can begin to see whose needs will shape our design.
This exercise helps ensure no important voice is left out.

- **Customer** – The person or business ordering drone deliveries
- **Operator** – The user or staff managing drone deployment
- **Maintenance Crew** – Responsible for repairs and routine checks
- **Regulator** – Aviation authority ensuring flight and airspace compliance
- **Recipient / End User** – Person receiving the delivered item
- **Logistics Supervisor** – Oversees fleet operations and scheduling
- **Public / Neighborhood** – Affected by noise, privacy, and safety

Stakeholder Examples – Autonomous Delivery Drone System
Let’s now take our theoretical concept and apply it to our class-wide system: the **Autonomous Delivery Drone**.
Who are the stakeholders? At first, it may seem like it’s just the person placing the order — but as we dig deeper, we see many more:
The **Customer** expects fast and accurate delivery.
The **Operator** needs intuitive controls and reliability.
The **Maintenance Team** requires access and diagnostic tools.
The **Regulator** ensures airspace laws are followed.
The **Recipient** might not be the same person who ordered the drone — their experience matters too.
And finally, the **general public** — people who never touch the drone but live underneath its flight path — they have concerns about noise, privacy, and safety.
These roles will help us model stakeholder needs in a structured and traceable way. Missing just one of these could lead to a design gap or risk later in development.



Stakeholder Identification

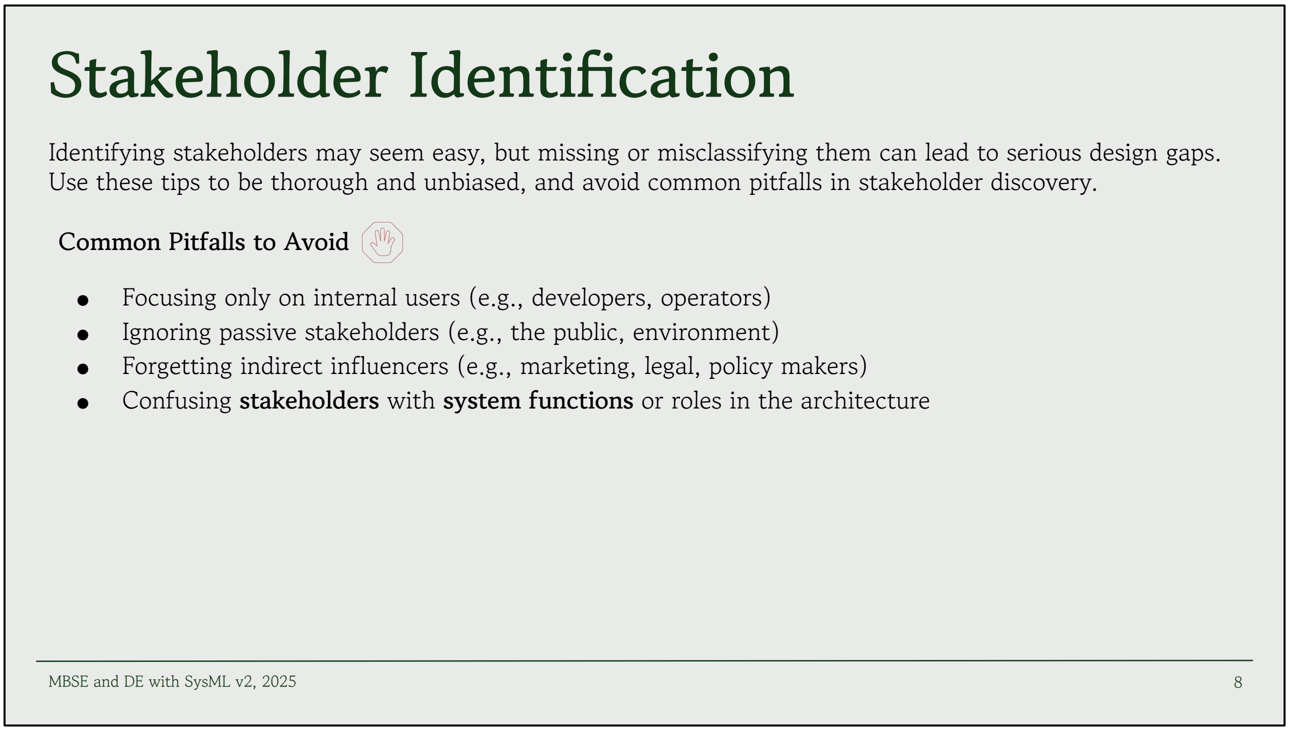
Identifying stakeholders may seem easy, but missing or misclassifying them can lead to serious design gaps. Use these tips to be thorough and unbiased, and avoid common pitfalls in stakeholder discovery.

Tips for Identifying Stakeholders

- Start from **system boundaries**: Who inputs/outputs data, materials, decisions?
- Ask: *Who installs it? Who uses it? Who maintains it? Who approves it? Who complains if it fails?*
- Use **real-world analogies** — think of logistics, operations, public safety.
- **Interview or brainstorm** — cross-functional teams help spot missing roles.
- Document assumptions — *"We assume the operator is the same as the user"* may not always be true.

Stakeholder Identification

It's easy to fall into the trap of thinking your stakeholders are just the people who directly touch the system — but systems are embedded in environments. Stakeholders include **anyone who interacts with, benefits from, enforces, or is affected by** the system, directly or indirectly.A good technique is to start from the **system boundary**. Who interacts with it? Who feeds it information? Who receives output? Then go further: Who needs to approve this system? Who could block it? Who fixes it when it fails?
In real projects, stakeholder blind spots lead to *requirements churn*, *non-compliance*, or even *public backlash*. MBSE helps by making stakeholder modeling explicit — but only if we capture the **right list of people** from the start.



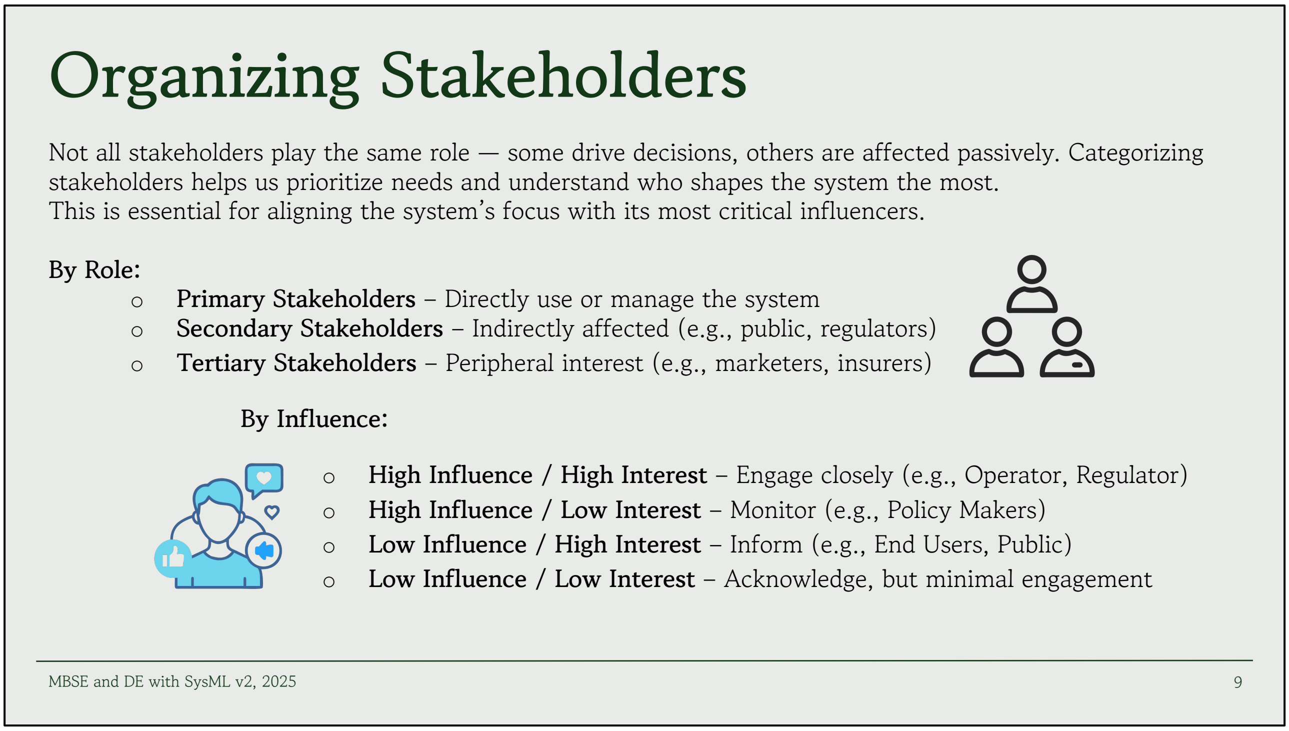
Stakeholder Identification

Identifying stakeholders may seem easy, but missing or misclassifying them can lead to serious design gaps. Use these tips to be thorough and unbiased, and avoid common pitfalls in stakeholder discovery.

Common Pitfalls to Avoid

- Focusing only on internal users (e.g., developers, operators)
- Ignoring passive stakeholders (e.g., the public, environment)
- Forgetting indirect influencers (e.g., marketing, legal, policy makers)
- Confusing **stakeholders** with **system functions** or roles in the architecture

Stakeholder Identification
It's easy to fall into the trap of thinking your stakeholders are just the people who directly touch the system — but systems are embedded in environments. Stakeholders include **anyone who interacts with, benefits from, enforces, or is affected by** the system, directly or indirectly.A good technique is to start from the **system boundary**. Who interacts with it? Who feeds it information? Who receives output? Then go further: Who needs to approve this system? Who could block it? Who fixes it when it fails?
In real projects, stakeholder blind spots lead to *requirements churn*, *non-compliance*, or even *public backlash*. MBSE helps by making stakeholder modeling explicit — but only if we capture the **right list of people** from the start.



Organizing Stakeholders

Not all stakeholders play the same role — some drive decisions, others are affected passively. Categorizing stakeholders helps us prioritize needs and understand who shapes the system the most.
This is essential for aligning the system's focus with its most critical influencers.

By Role:

- **Primary Stakeholders** – Directly use or manage the system
- **Secondary Stakeholders** – Indirectly affected (e.g., public, regulators)
- **Tertiary Stakeholders** – Peripheral interest (e.g., marketers, insurers)

By Influence:

- **High Influence / High Interest** – Engage closely (e.g., Operator, Regulator)
- **High Influence / Low Interest** – Monitor (e.g., Policy Makers)
- **Low Influence / High Interest** – Inform (e.g., End Users, Public)
- **Low Influence / Low Interest** – Acknowledge, but minimal engagement

Organizing Stakeholders by Role and Influence
After identifying our stakeholders, we need to organize them in a meaningful way. This helps us manage complexity and allocate attention where it matters most.
First, we classify stakeholders by **role**. **Primary stakeholders** are usually users, operators, or owners. **Secondary stakeholders** may never touch the system, but they're affected by it — like neighbors under a drone's flight path. **Tertiary stakeholders** are even more removed — they may have a business or policy interest, but no direct interaction.
Then we classify them by **influence**. A powerful method is to use a **2x2 matrix**: one axis for **influence** and one for **interest**.

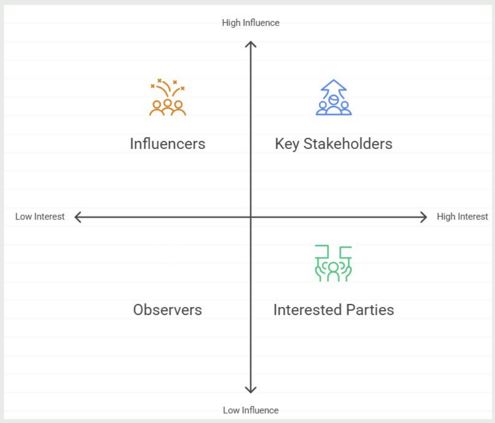
- Stakeholders with **high influence and high interest** are your top priority — think of the drone operator or airspace regulator.
- Those with **low interest but high influence**, like legal or policy advisors, can block your progress if ignored.
- And don't forget the **public** — low influence, maybe, but high interest if drones buzz their neighborhoods.

This kind of mapping is critical when we later model needs and prioritize design decisions. It's not about ignoring people — it's about managing trade-offs wisely.

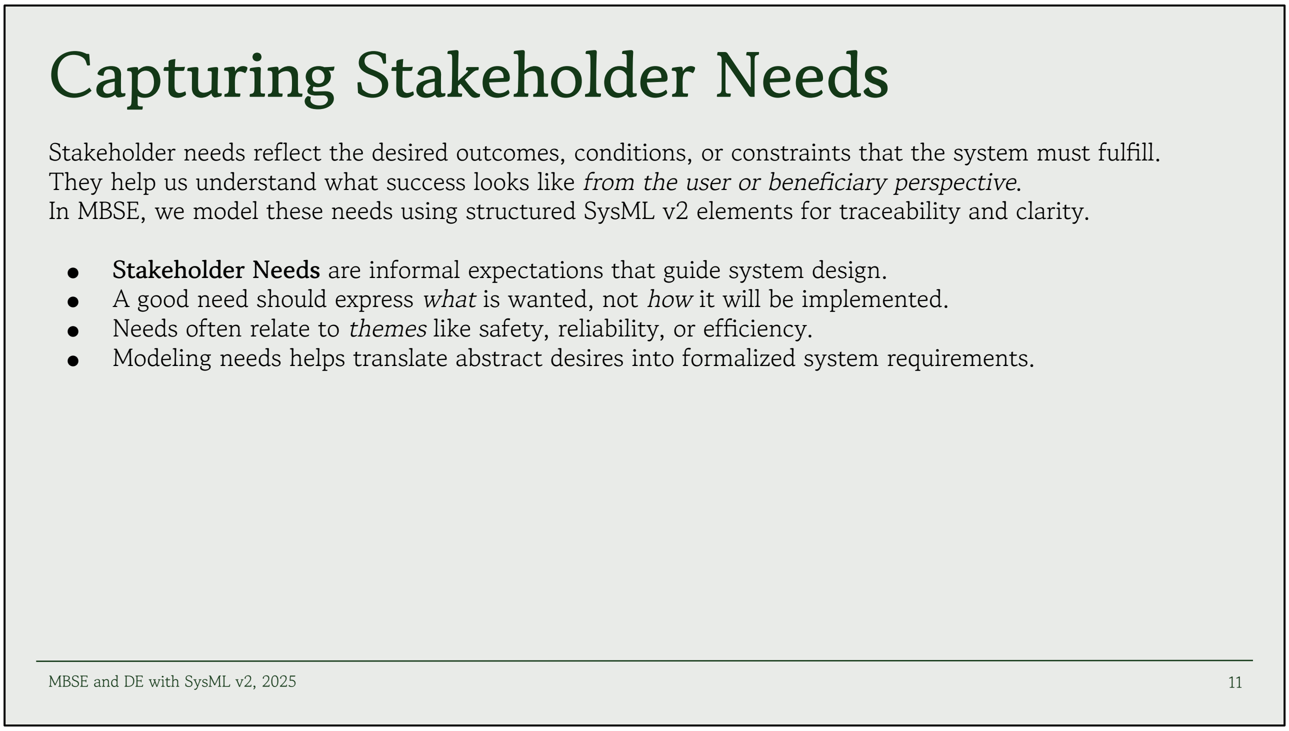
Stakeholder Influence vs Interest

Let's categorize drone stakeholders using a common systems approach: the Influence vs Interest Matrix. This tool helps you visualize where to focus your engagement, communication, and design attention.

Quadrant	Stakeholder	Notes
🔵 High Influence / High Interest	**Drone Operator**, **Airspace Regulator**	Involved in operations and compliance — top priority
🟠 High Influence / Low Interest	**Logistics Supervisor**, **Policy Maker**	Can shape policy or direction — keep informed
🟢 Low Influence / High Interest	**Customer**, **Recipient**, **Public**	Impacted directly — ensure transparency and trust
⚪ Low Influence / Low Interest	**Insurance Agent**, **Local Vendors**	Low risk, but should still be acknowledged



Stakeholder Influence vs Interest – Drone System Example
Let's apply the influence-interest matrix to our Autonomous Delivery Drone System.
In the **High-High quadrant**, we place the **Drone Operator** and the **Airspace Regulator**. These two have direct power over the system and care deeply about how it functions. They will be your most engaged stakeholders — requiring ongoing validation and involvement.
High influence but low interest groups, like **logistics managers** or **policy makers**, may not care about day-to-day operations but could block or redirect the system at any time — treat them with strategic updates.
In the **Low influence but high interest** zone, we see **Customers**, **Recipients**, and the **Public**. They don't shape the system, but they experience it. If we ignore their concerns (like noise, accuracy, or safety), the system could lose public trust or fail adoption.
Finally, the **Low-Low quadrant** may include supporting stakeholders like **vendors** or **insurers**. Their needs are minor, but we acknowledge them to ensure completeness.
This kind of stakeholder mapping will guide how we prioritize needs and trace requirements later on. It's an essential early modeling step.

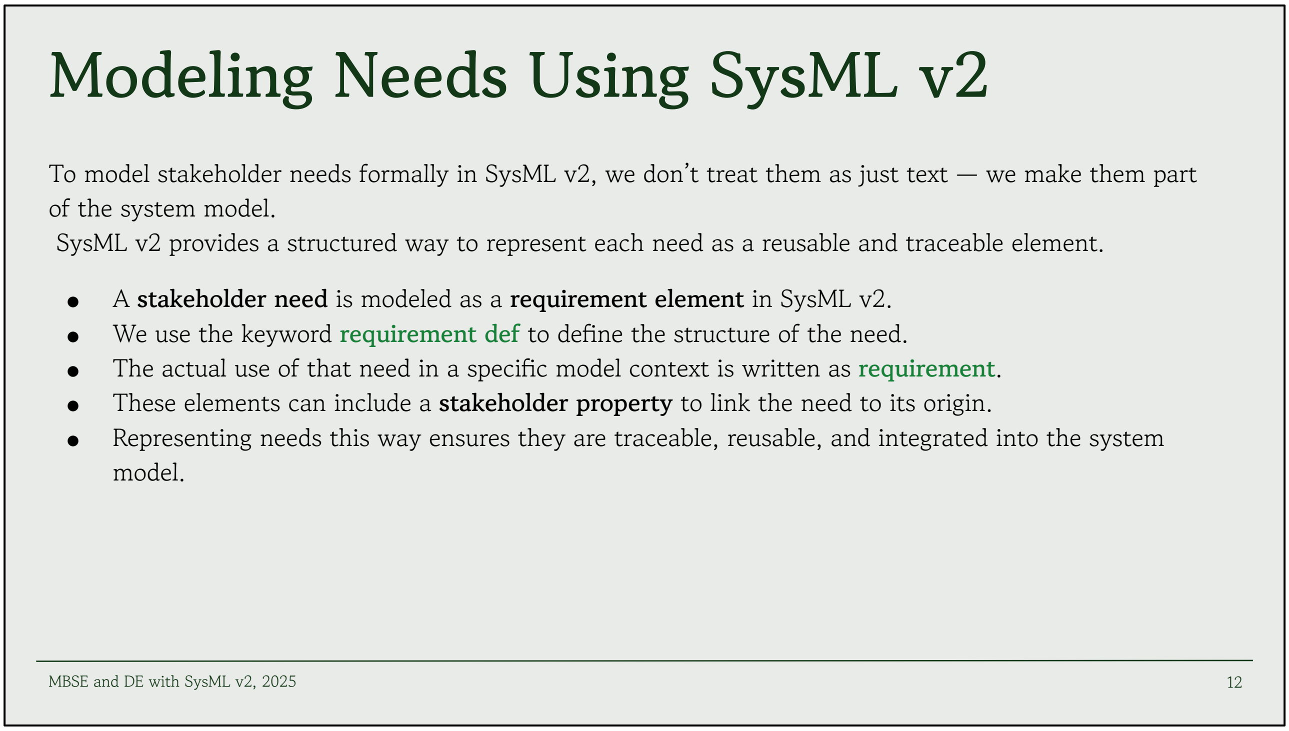


Capturing Stakeholder Needs

Stakeholder needs reflect the desired outcomes, conditions, or constraints that the system must fulfill.
They help us understand what success looks like *from the user or beneficiary perspective*.
In MBSE, we model these needs using structured SysML v2 elements for traceability and clarity.

- **Stakeholder Needs** are informal expectations that guide system design.
- A good need should express *what* is wanted, not *how* it will be implemented.
- Needs often relate to *themes* like safety, reliability, or efficiency.
- Modeling needs helps translate abstract desires into formalized system requirements.

Capturing Stakeholder Needs – Core Concepts
Before we talk about diagrams or SysML elements, we need to deeply understand what a **stakeholder need** really is.
A need is not yet a requirement. It doesn't say how something will be done or even specify exact numbers. It reflects what a stakeholder wants the system to **achieve** or **enable** — it expresses **purpose and expectations**.
For example, in our drone system, a customer might want "fast delivery," a regulator might care about "airspace safety," and a maintenance team might need "easy access for diagnostics." These are all valid stakeholder needs.
One of the biggest mistakes engineers make is skipping this step — going straight to system structure or behavior without really understanding the people behind the system.
That leads to rework, missed expectations, and failure to meet stakeholder goals.
Modeling needs helps us ground the system in **real-world value**. As we move forward in this course, we'll refine these needs into technical requirements, architectures, and behaviors. But it all starts right here — with the voice of the stakeholder.



Modeling Needs Using SysML v2

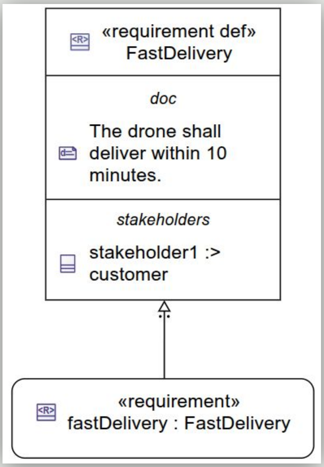
To model stakeholder needs formally in SysML v2, we don't treat them as just text — we make them part of the system model.
SysML v2 provides a structured way to represent each need as a reusable and traceable element.

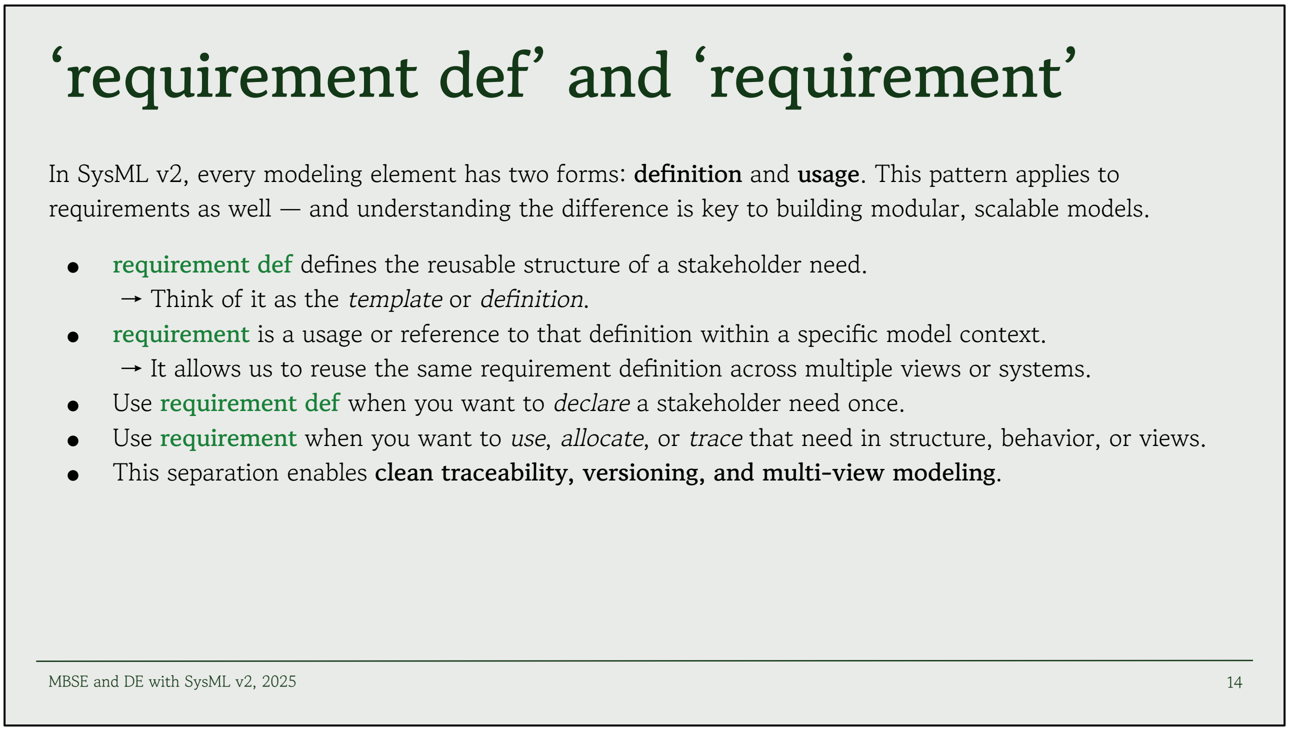
- A **stakeholder need** is modeled as a **requirement element** in SysML v2.
- We use the keyword **requirement def** to define the structure of the need.
- The actual use of that need in a specific model context is written as **requirement**.
- These elements can include a **stakeholder property** to link the need to its origin.
- Representing needs this way ensures they are traceable, reusable, and integrated into the system model.

Modeling Needs Using SysML v2 Elements
When we talk about stakeholder needs in MBSE, we don't want to just keep them in a spreadsheet or Word document — we want to make them **first-class citizens in the model**.
In SysML v2, we do this by defining a **requirement element**. We use the keyword **requirement def** to define the need — this is like creating a reusable template or definition. Then, when we apply that need in the model, we create a **requirement**, which is a usage of the definition.
For now, we don't need to worry about the difference between these two keywords — we'll focus on that in the next slide. But it's important to understand that this structure allows us to manage complexity and scale.
Every stakeholder need we model can also reference the person or group it came from by setting the **stakeholder property**. This gives us automatic traceability — we always know *who* asked for *what*.
We'll keep building on this pattern as we connect needs to concerns, and later to system requirements, structure, and behavior.

Modeling Needs Using SysML v2

Example of Graphic Notation for ‘requirement def’ and ‘requirement with their relationship.





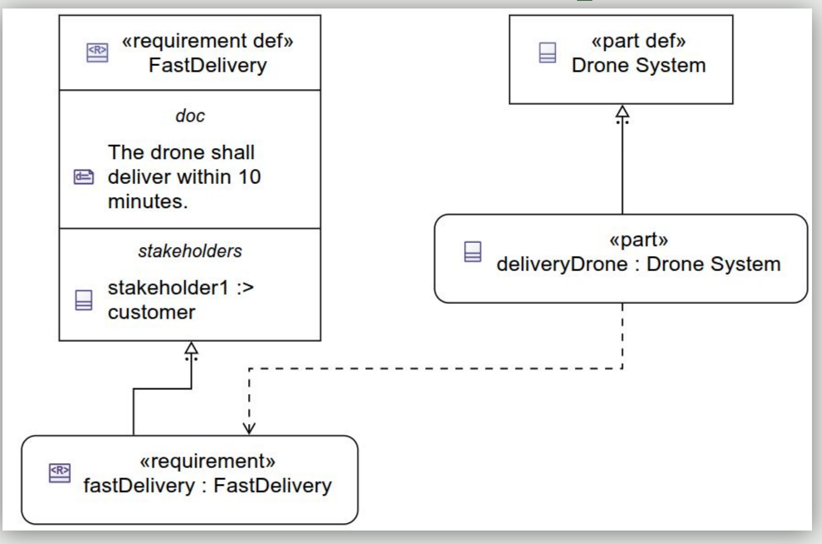
‘requirement def’ and ‘requirement’

In SysML v2, every modeling element has two forms: **definition** and **usage**. This pattern applies to requirements as well — and understanding the difference is key to building modular, scalable models.

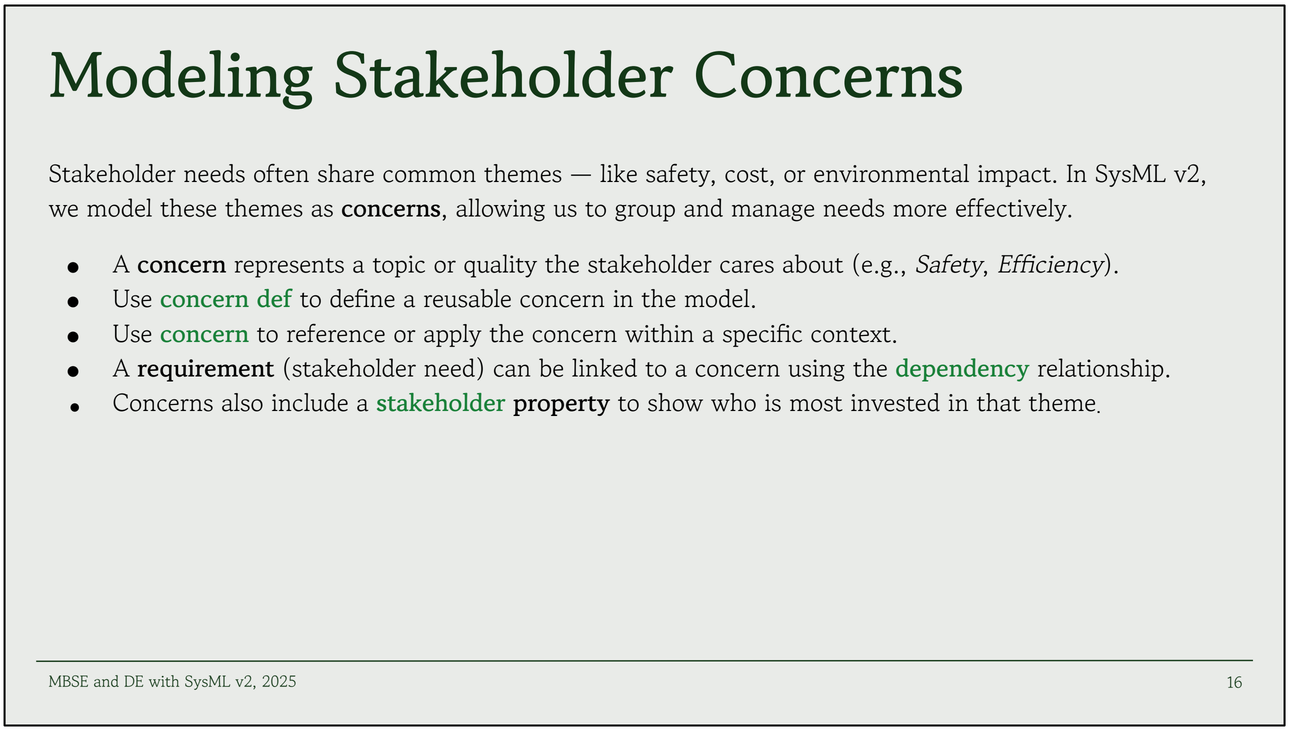
- **requirement def** defines the reusable structure of a stakeholder need.
  → Think of it as the *template* or *definition*.
- **requirement** is a usage or reference to that definition within a specific model context.
  → It allows us to reuse the same requirement definition across multiple views or systems.
- Use **requirement def** when you want to *declare* a stakeholder need once.
- Use **requirement** when you want to *use*, *allocate*, or *trace* that need in structure, behavior, or views.
- This separation enables **clean traceability, versioning, and multi-view modeling**.

requirement def vs requirement in SysML v2
SysML v2 follows a powerful and consistent pattern: **everything has a definition and a usage**. This applies to parts, connectors, behaviors — and also to requirements.So what’s the difference between requirement def and requirement?
A requirement def is where you **define** the need — give it a name, a description, and a stakeholder reference. Think of it as the master copy of the need. You usually create it only once.
A requirement is a **usage** of that definition. You use it whenever you need to reference that requirement in your system structure, allocate it to a function, or satisfy it with a design element. This means you can reuse the same defined requirement across **multiple parts of the model** without duplicating it.

‘requirement def’ and ‘requirement’



For example, let’s say our drone needs to deliver a package within 10 minutes. We define that once using requirement def FastDelivery. Later, we can use requirement fastDelivery: FastDelivery in the delivery system model, in the behavior model, and in the performance analysis — all pointing back to the same source.
This keeps our model clean, traceable, and aligned — exactly what MBSE is meant to do.



Modeling Stakeholder Concerns

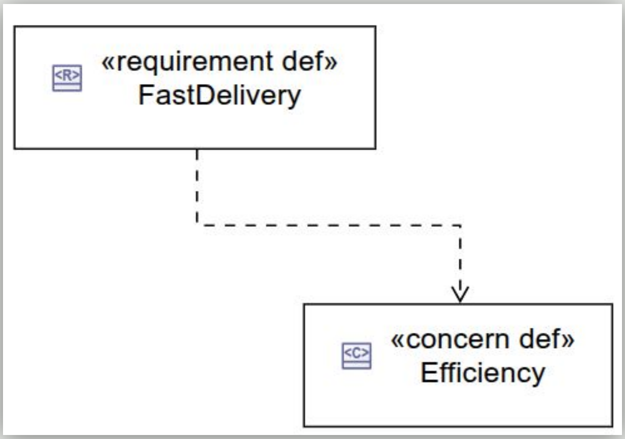
Stakeholder needs often share common themes — like safety, cost, or environmental impact. In SysML v2, we model these themes as **concerns**, allowing us to group and manage needs more effectively.

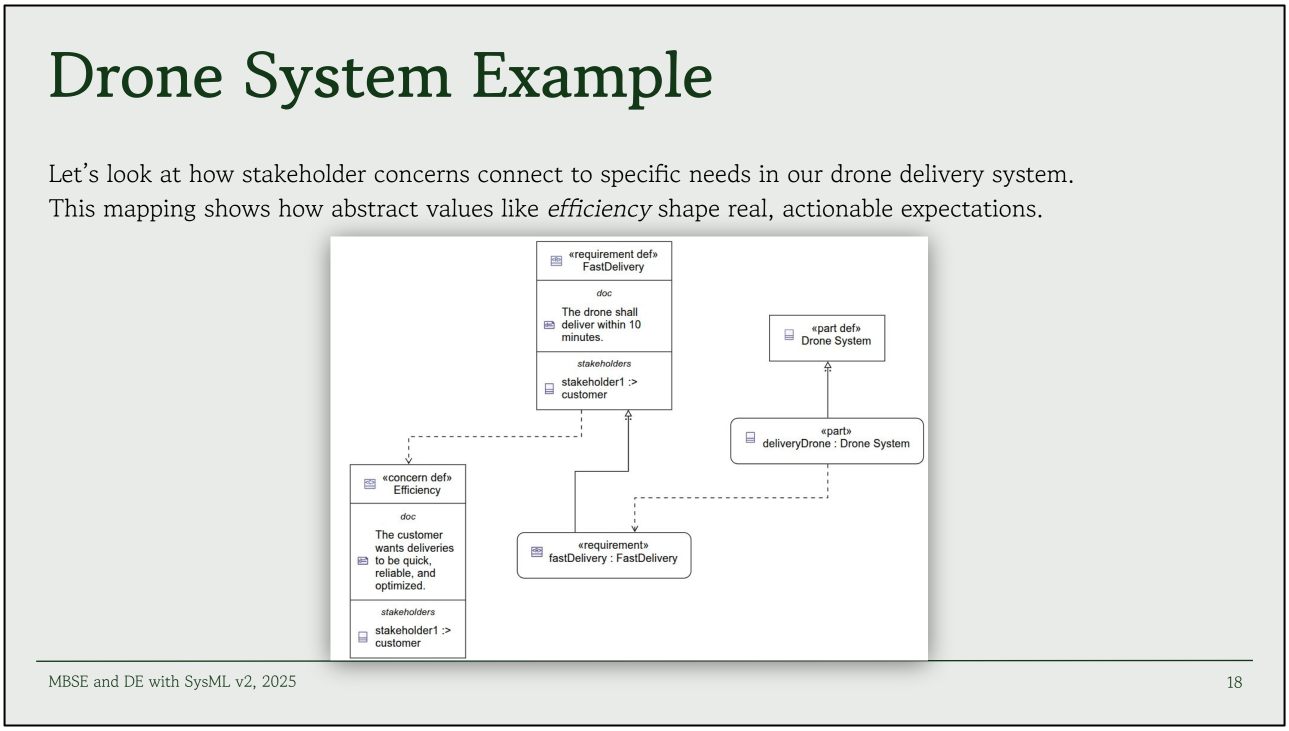
- A **concern** represents a topic or quality the stakeholder cares about (e.g., *Safety*, *Efficiency*).
- Use **concern def** to define a reusable concern in the model.
- Use **concern** to reference or apply the concern within a specific context.
- A **requirement** (stakeholder need) can be linked to a concern using the **dependency** relationship.
- Concerns also include a **stakeholder** **property** to show who is most invested in that theme.

Modeling Stakeholder Concerns with concern def
Sometimes stakeholder needs don’t stand alone — they relate to bigger ideas or values that shape many decisions. These are called **concerns**.
In SysML v2, we model them using concern def. Think of it like creating a label or theme — such as *Safety*, *User Experience*, or *Efficiency*. These concerns help us group requirements that share the same intent or priority.
For example, the Customer might care deeply about efficiency. If we define a concern called “Efficiency,” we can link any number of requirements — like “Fast Delivery” or “Battery Optimization” — back to it using the dependency relationship.
This gives us two major benefits. First, we can **see patterns** across stakeholder needs — multiple needs driven by one concern. Second, we can make decisions with **traceable priorities** — when designing a feature, we know if it’s supporting Efficiency, Safety, or another goal.
Like requirements, each concern can include a stakeholder property. This ensures we know **who** is driving the concern. That clarity is essential in trade studies or system refinement later on.

Modeling Stakeholder Concerns

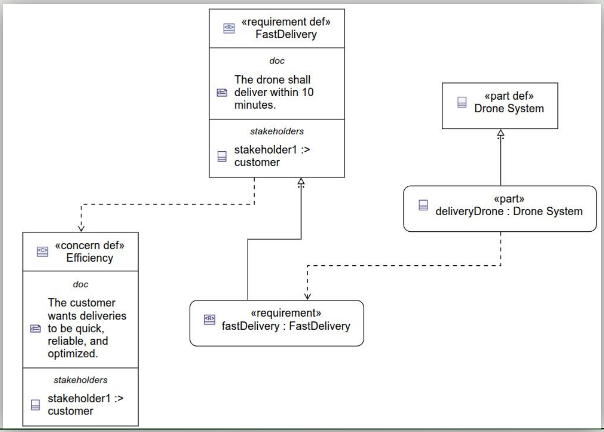
Graphical notation to express that the requirement *FastDelivery* depends on the concern *Efficiency*.





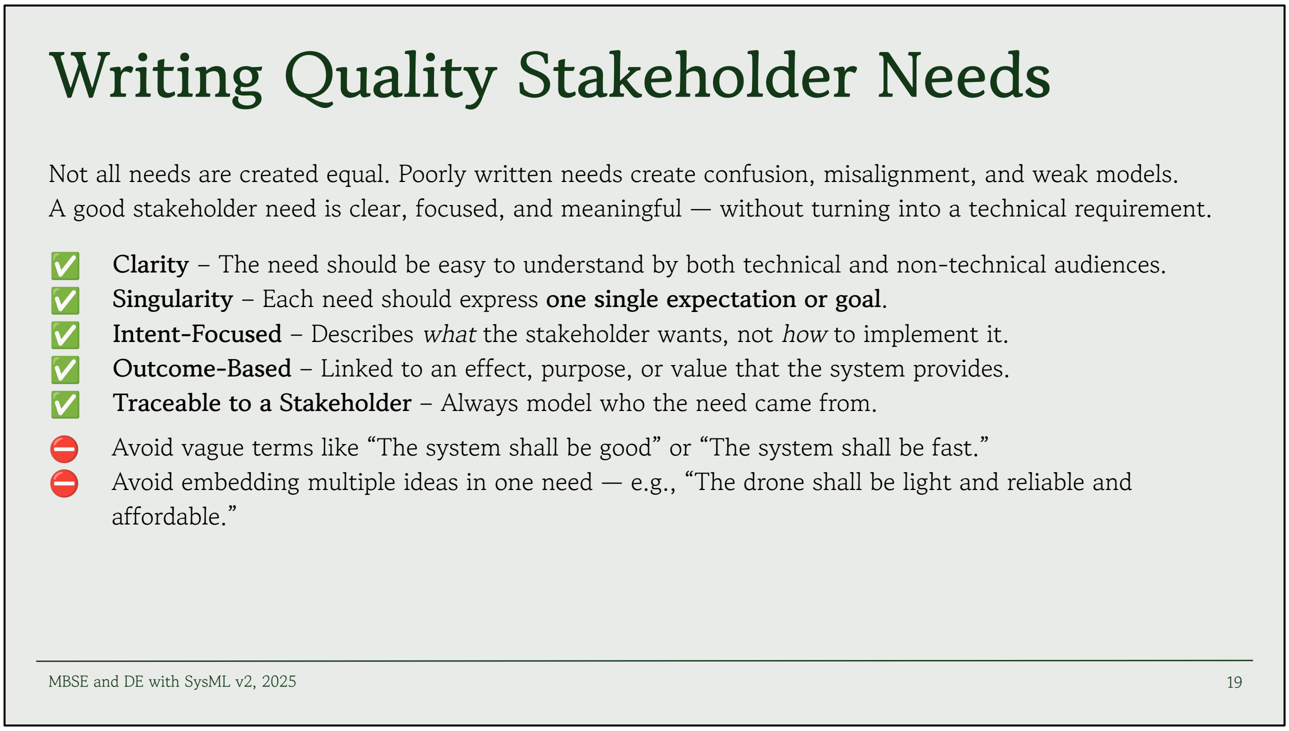
Drone System Example

Let’s look at how stakeholder concerns connect to specific needs in our drone delivery system. This mapping shows how abstract values like *efficiency* shape real, actionable expectations.



«requirement def» FastDelivery
doc
The drone shall deliver within 10 minutes.
stakeholders
stakeholder1 :> customer
«part def» Drone System
«part» deliveryDrone : Drone System
«concern def» Efficiency
doc
The customer wants deliveries to be quick, reliable, and optimized.
stakeholders
stakeholder1 :> customer
«requirement» fastDelivery : FastDelivery

Drone System Example – Mapping Concerns to Requirements
Let’s bring the concepts to life with a simple but realistic example.In our drone delivery system, the **Customer** wants the service to be fast and consistent. That’s not just one feature — it’s a value, a theme. So we model that as a **concern** called “Efficiency.”Now, that concern leads to specific stakeholder needs — like the drone being able to **deliver a package within 10 minutes**. We model that as a requirement: requirement def FastDelivery.
In SysML v2, we represent the **link** between this requirement and the Efficiency concern using a **generic dependency relationship**.
By also using the stakeholder property in both the concern and requirement, we know they both came from the Customer. This gives us **clean traceability** and a strong modeling foundation for future refinement into logical architecture.



Writing Quality Stakeholder Needs

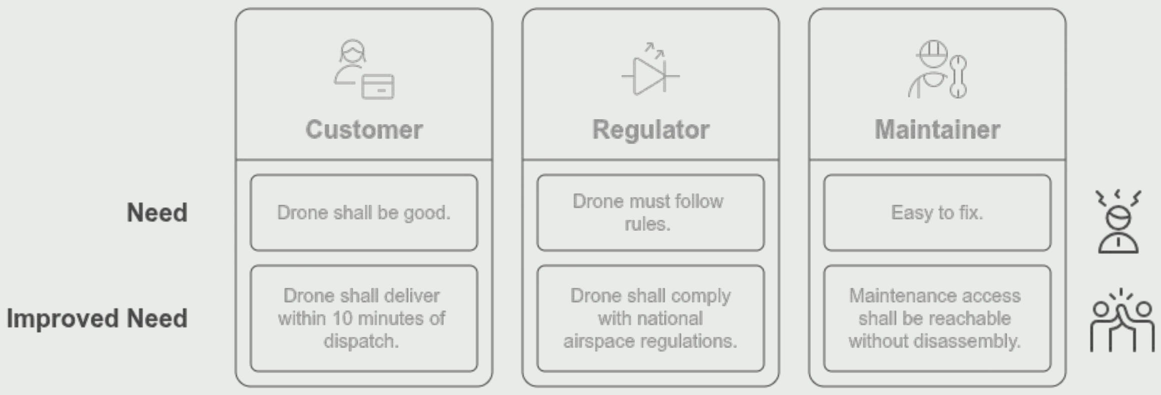
Not all needs are created equal. Poorly written needs create confusion, misalignment, and weak models.
A good stakeholder need is clear, focused, and meaningful — without turning into a technical requirement.

✅ **Clarity** – The need should be easy to understand by both technical and non-technical audiences.
✅ **Singularity** – Each need should express **one single expectation or goal**.
✅ **Intent-Focused** – Describes *what* the stakeholder wants, not *how* to implement it.
✅ **Outcome-Based** – Linked to an effect, purpose, or value that the system provides.
✅ **Traceable to a Stakeholder** – Always model who the need came from.

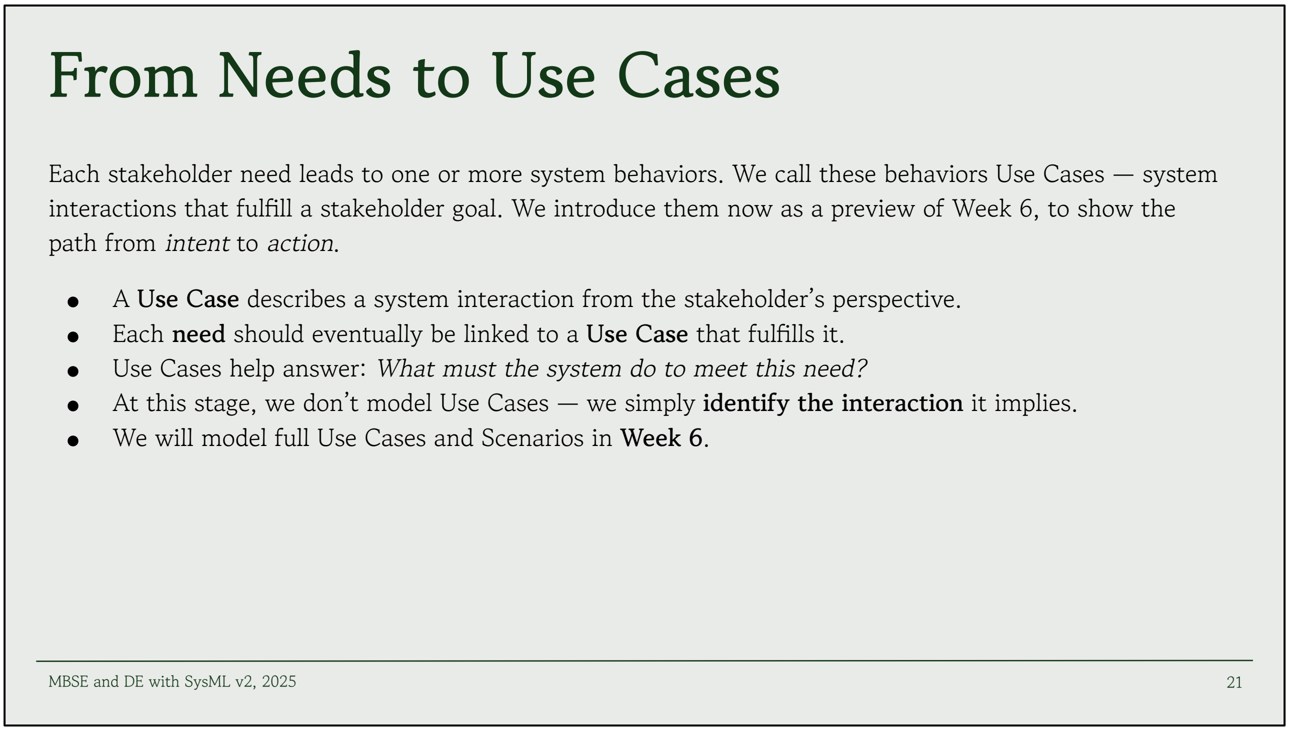
⛔ Avoid vague terms like “The system shall be good” or “The system shall be fast.”
⛔ Avoid embedding multiple ideas in one need — e.g., “The drone shall be light and reliable and affordable.”

Writing Quality Stakeholder Needs
Many system failures can be traced not to bad code or poor hardware — but to **ambiguous or low-quality needs**. That’s why we take time to write good ones.
A good stakeholder need should express one clear expectation. Not a wish list, not a vague idea, and not a technical requirement.
We don’t want: “The drone shall be good.” That’s meaningless. Instead: “The drone shall deliver a package within 10 minutes” — that’s a **focused, outcome-driven** need.
Also avoid combining multiple thoughts in one sentence. A need that says “The drone shall be lightweight, reliable, and affordable” may sound fine, but it’s actually **three separate needs**. Each one should be modeled individually so it can be validated and traced.
At this stage, we don’t need numeric precision — that’s for the Logical Layer. But we do want **well-formed, stakeholder-aligned needs** that can evolve into real system requirements later.

Writing Quality Stakeholder Needs



	Customer	Regulator	Maintainer
Need	Drone shall be good.	Drone must follow rules.	Easy to fix.
Improved Need	Drone shall deliver within 10 minutes of dispatch.	Drone shall comply with national airspace regulations.	Maintenance access shall be reachable without disassembly.



From Needs to Use Cases

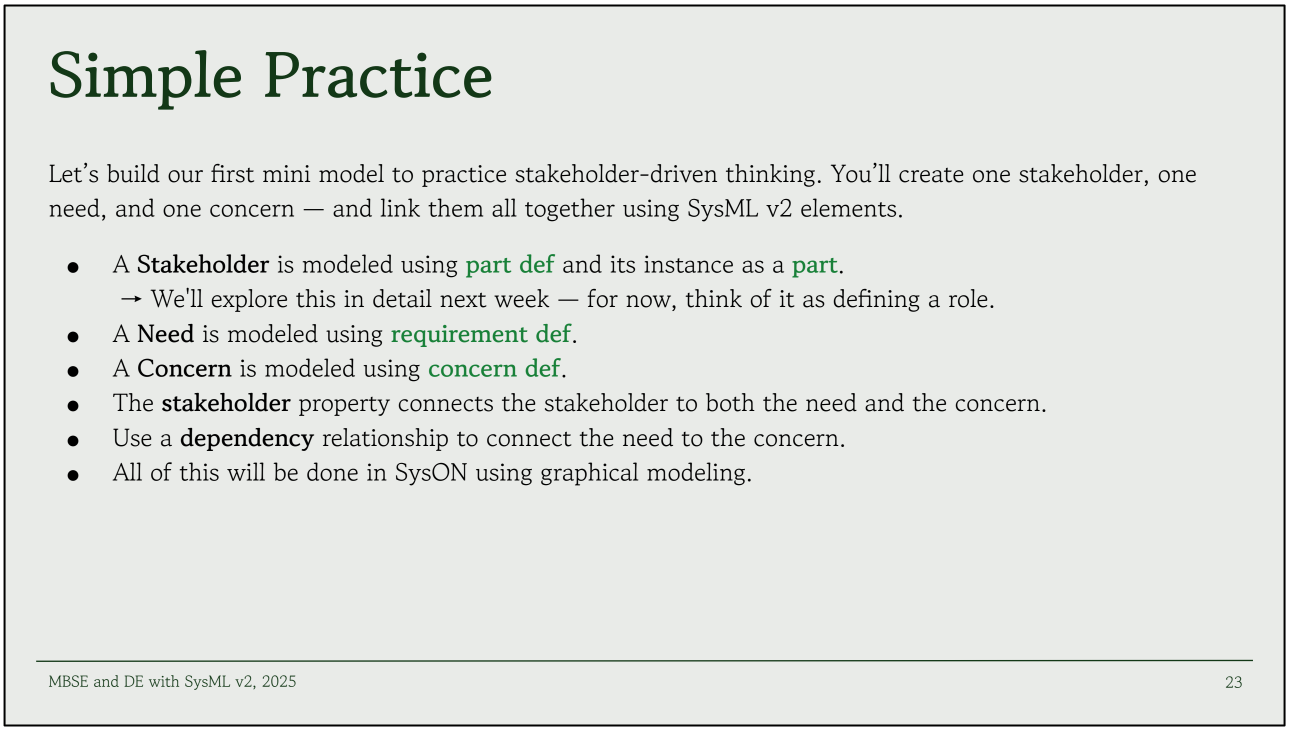
Each stakeholder need leads to one or more system behaviors. We call these behaviors Use Cases — system interactions that fulfill a stakeholder goal. We introduce them now as a preview of Week 6, to show the path from *intent* to *action*.

- A **Use Case** describes a system interaction from the stakeholder's perspective.
- Each **need** should eventually be linked to a **Use Case** that fulfills it.
- Use Cases help answer: *What must the system do to meet this need?*
- At this stage, we don't model Use Cases — we simply **identify the interaction** it implies.
- We will model full Use Cases and Scenarios in **Week 6**.

From Needs to Use Cases – A First Glimpse
Even though we'll go deep into Use Cases in Week 6, it's important we introduce the idea now.
Why? Because every stakeholder need ultimately drives some system behavior — and those behaviors are captured through Use Cases.
If the customer needs "Fast Delivery," what should the system do? It must **perform a delivery** within a certain time. That's a Use Case.
If a regulator cares about airspace safety, the system must **perform navigation while respecting restricted zones**. That's another Use Case.
So think of Use Cases as **the bridge** between *needs* and *system design*. We're not building diagrams or scenarios yet — just showing that every need should eventually lead to an action the system will perform.
This gives us a forward-looking view and prepares their thinking for what's coming in Week 6.

From Needs to Use Cases

Needs	Leads to Use Case
“Deliver within 10 minutes”	Perform Delivery
“Avoid restricted airspace”	Perform Navigation with Airspace Rules
“Allow easy maintenance”	Perform Diagnostics



Simple Practice

Let's build our first mini model to practice stakeholder-driven thinking. You'll create one stakeholder, one need, and one concern — and link them all together using SysML v2 elements.

- A **Stakeholder** is modeled using **part def** and its instance as a **part**.
  ➝ We'll explore this in detail next week — for now, think of it as defining a role.
- A **Need** is modeled using **requirement def**.
- A **Concern** is modeled using **concern def**.
- The **stakeholder** property connects the stakeholder to both the need and the concern.
- Use a **dependency** relationship to connect the need to the concern.
- All of this will be done in SysON using graphical modeling.

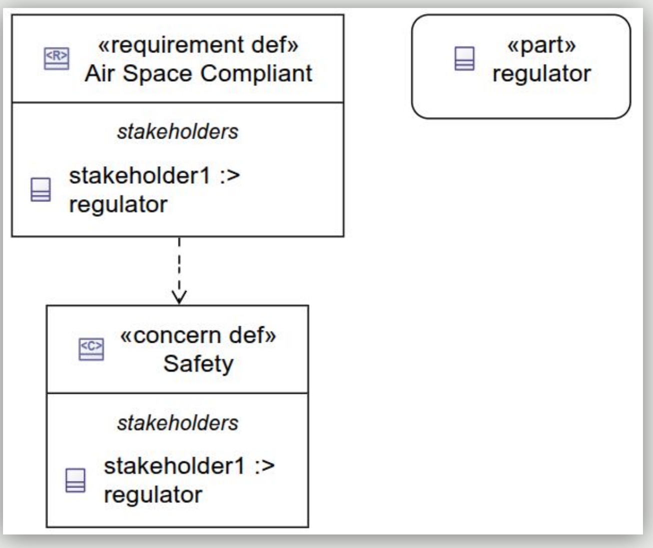
Simple Practice – Stakeholder, Need, and Concern
Before we start your main project, let's get comfortable with SysML v2 by building a **small, guided model**.
You're going to create three elements:

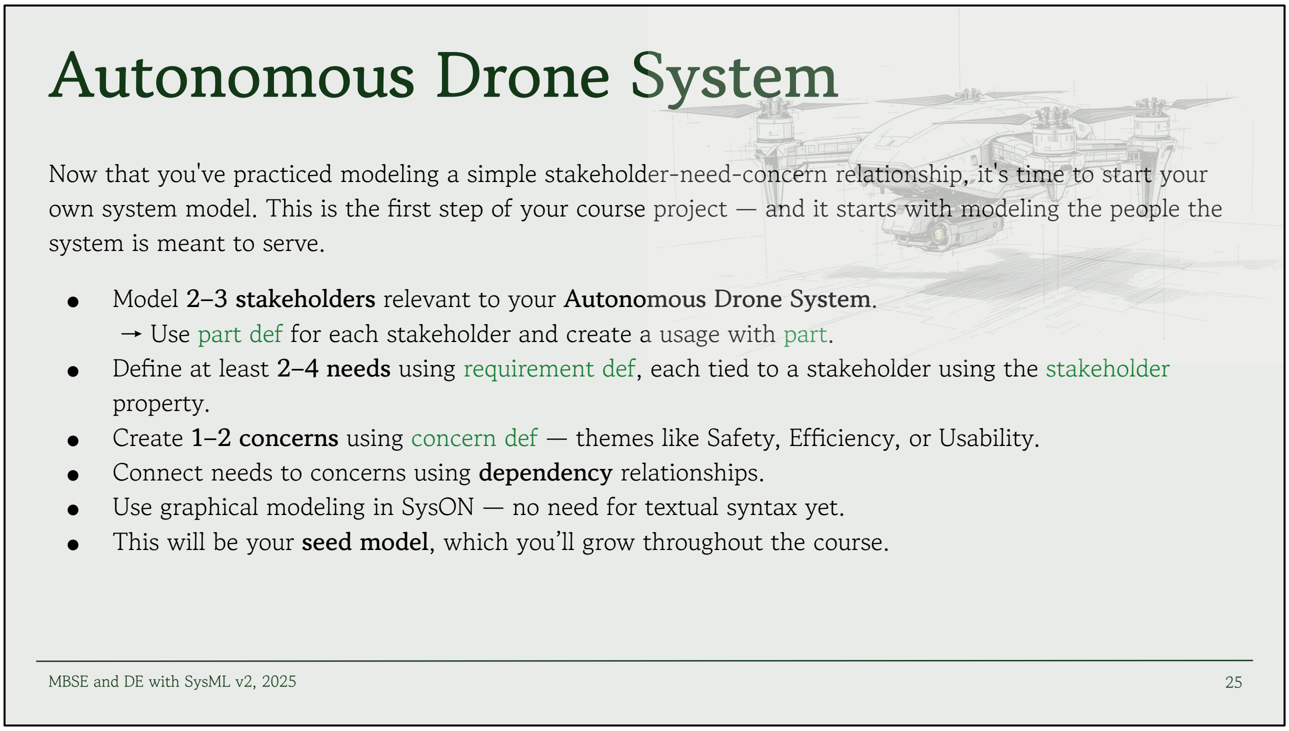
1. A **Stakeholder** — in this case, let's use Regulator.
   - We model the stakeholder using part def Regulator, and then create a usage of it with part.
   - You can think of this like defining a role and placing it in the system for modeling purposes. We'll cover structure modeling more deeply next week.
2. A **Need** — for example, "Drone shall comply with airspace laws."
   - Model this using requirement def AirspaceCompliant.
   - Set its stakeholder property to point to the Regulator.
3. A **Concern** — here, the concern is Safety.
   - Use concern def Safety, and also set stakeholder = Regulator.

Finally, you'll **connect the requirement to the concern** using a dependency link.

This exercise helps you practice using **graphical modeling**, **property assignment**, and **traceability** — all in a clean, focused setup.

Simple Practice



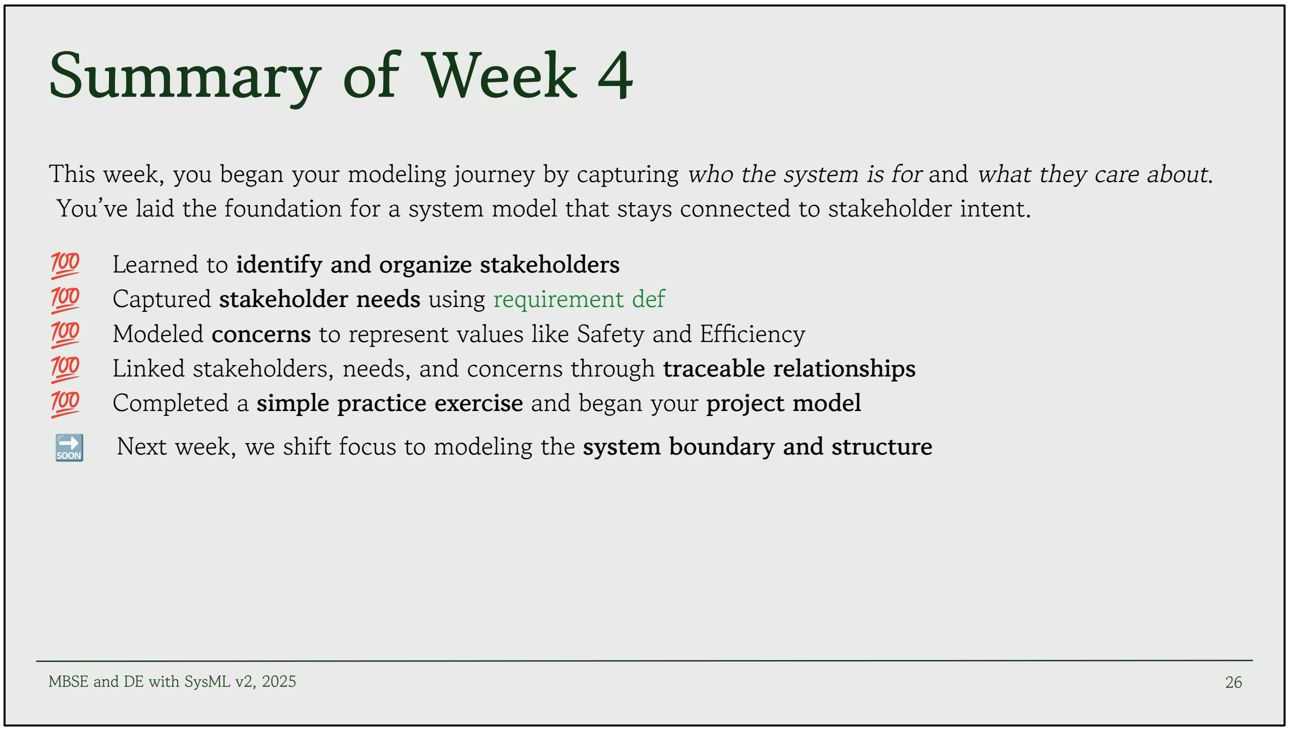


Autonomous Drone System

Now that you've practiced modeling a simple stakeholder-need-concern relationship, it's time to start your own system model. This is the first step of your course project — and it starts with modeling the people the system is meant to serve.

- Model **2–3 stakeholders** relevant to your **Autonomous Drone System**.
  → Use part def for each stakeholder and create a usage with part.
- Define at least **2–4 needs** using requirement def, each tied to a stakeholder using the stakeholder property.
- Create **1–2 concerns** using concern def — themes like Safety, Efficiency, or Usability.
- Connect needs to concerns using **dependency** relationships.
- Use graphical modeling in SysON — no need for textual syntax yet.
- This will be your **seed model**, which you'll grow throughout the course.

Start Building Your Project Model – Autonomous Drone System
Now it's time to move from learning to doing — and start building your **own system model**.
Your mission today is to define the **first layer** of your Autonomous Drone System: the people and what they want.
Think about who interacts with the system. For example,. Is it the Customer who orders the package? The Operator who launches the drone? The Regulator who controls airspace? Each of these becomes a part def Stakeholder in your model.
Then, define their **needs**. What do they want the drone to do or avoid? "Deliver fast," "Avoid noise," "Follow airspace rules" — these are all requirement def elements.
Don't forget to group needs by **concerns** — such as Safety or Efficiency — using concern def.
Link them with **dependency relationships** and always assign the stakeholder property to both your needs and concerns.
You'll begin modeling this in class — and if you don't finish, you'll complete it as homework.
This becomes your **baseline model**, and we'll build on it every week.



Summary of Week 4

This week, you began your modeling journey by capturing *who the system is for* and *what they care about*. You've laid the foundation for a system model that stays connected to stakeholder intent.

- 💯 Learned to **identify and organize stakeholders**
- 💯 Captured **stakeholder needs** using requirement def
- 💯 Modeled **concerns** to represent values like Safety and Efficiency
- 💯 Linked stakeholders, needs, and concerns through **traceable relationships**
- 💯 Completed a **simple practice exercise** and began your **project model**
- 🔜 Next week, we shift focus to modeling the **system boundary and structure**

You've now taken the first real step into MBSE modeling using SysML v2.
What you've done this week is deceptively powerful. By identifying stakeholders, defining their needs, and structuring concerns, you've anchored your system model in **purpose**.
This foundation will shape everything that follows — your architecture, behavior, constraints, and evaluation.
Next week, we move from *what the system must do* to *what the system is*. We'll define the **System Boundary**, identify what's inside vs outside, and begin modeling its **internal parts**.
Your stakeholder needs will guide this — so keep refining your model as homework, and bring your updated work next time.

QUESTION!