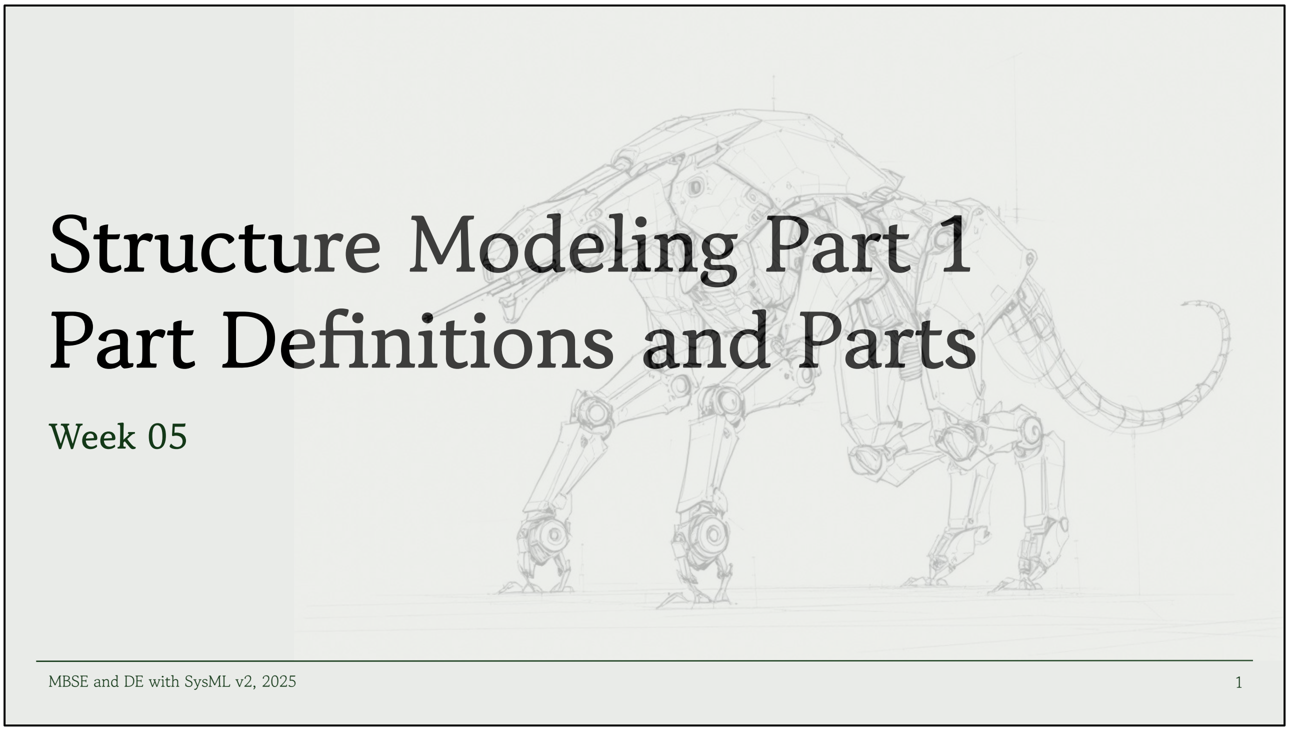


Structure Modeling Part 1
Part Definitions and Parts

Week 05

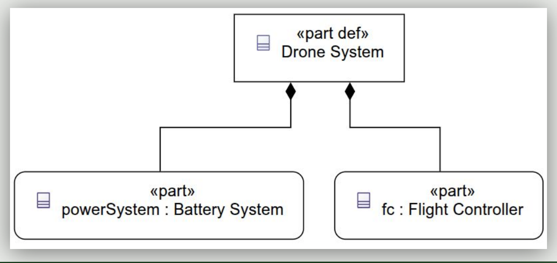
This week we begin a critical part of modeling — representing the **structure** of your system. We'll learn how to define building blocks using **Part Definitions**, how to represent real instances inside the system using **Part Usages**, and how to connect them.
This forms the backbone of your system architecture model — essential for organizing complexity and supporting traceability. We'll end the session with both practice and progress on your course project model — our autonomous drone system.

Modeling Structure

To model a system's internal structure, SysML v2 uses two key building blocks:

- **Part Definition** defines a reusable component type (e.g., Battery System).
- **Part Usage** represents an actual use of a component inside another system (e.g., A Drone System has a Battery System).

Each usage has its **own identity and properties** in the system.
These Part Usages can be **connected** using Connectors to represent interaction or integration.



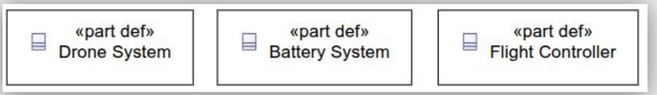
Modeling Structure with Part Definition, Part Usage, and Internal Connections
In system modeling, structure tells us *what* a system is made of and *how* its components are organized. To support this in SysML v2, we use two major elements: **Part Definitions** and **Part Usages**. Think of a Part Definition like a blueprint — it defines a reusable type, such as a Battery System or Flight Controller. The Part Usage is an instance of that blueprint, placed inside another system to represent a real configuration.
You can use the same Part Definition in multiple places but customize each usage with its own values, connections, or constraints. For example, one drone may use a Battery System with 5000mAh, another with 3000mAh — both refer to the same type, but different usages.
These parts don't just sit next to each other — they're connected. The **connections** can represent things like signal flow, power supply, mechanical joints, or communication links.
We'll use connectors to represent those relationships in a simple way today. Ports and interfaces will come later in Week 8.

What is a Part Definition

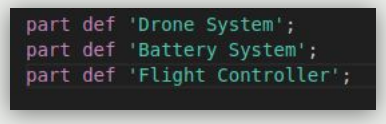
A **Part Definition** describes a reusable *type* of a system component. It is a **blueprint** that defines the internal structure, properties, and sub-parts of that component. It is not a real object — it's the definition or template. Examples:

- part def BatterySystem defines what a battery system *is*.
- part def FlightController defines the behavior and structure of a flight controller.
- part def DroneSystem can include several other parts in its structure.

A Part Definition can be reused across many systems, helping standardize design and reduce redundancy.



Example of Graphical Notation for Part Definition



```
part def 'Drone System';
part def 'Battery System';
part def 'Flight Controller';
```

Text Notation of them

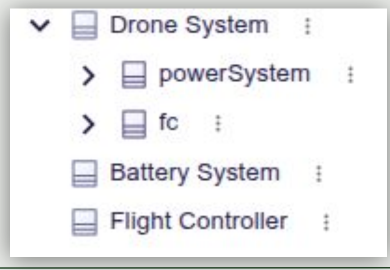
What is a Part Definition?
Let's think of a Part Definition as the **"class" or "type"** of a component. It tells us what a component is, what it contains, and what it's capable of. But by itself, it doesn't mean there's a specific instance of that component yet — we haven't said where it lives or what it connects to.
For example, BatterySystem might define its voltage range, size, and internal substructure. But until we say, "this drone has one," it's just a definition — like defining what a car is, but not pointing to any specific car.
SysML v2 uses part def as the keyword to define these reusable component types. This helps us build libraries of standardized components we can reuse across projects.

What is Part Usage?

A **Part Usage** represents a specific use or instance of a component inside another system. It refers to a **Part Definition**, but it has its own identity, name, and properties within the parent system. This is how we **assemble systems** from reusable parts. Examples:

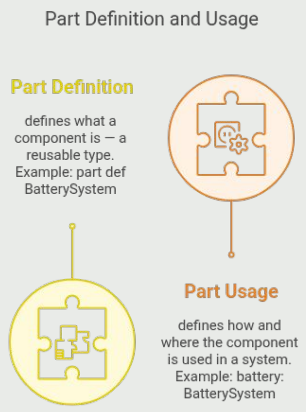
- powerSystem: BatterySystem — A usage named *powerSystem* of the type *BatterySystem*.
- fc: FlightController — A usage named *fc* of the *FlightController* definition.
- sensor1: SensorModule — One of many usages of the same definition.

Part Usages can be **connected** to each other to model real interactions.



What is a Part Usage?
While a Part Definition gives us the blueprint, the **Part Usage** is what places that component into a real system. It's like saying, "this drone uses this specific battery system." Every usage has a **name** and refers back to its **definition**, allowing us to track how many times a type is reused and where.
This is essential for modeling real systems. For example, we may have one *BatterySystem* definition, but three drones might use it — each with different capacities or behaviors depending on their context.
It also lets us model **composition** — putting usages inside higher-level parts, like nesting subsystems into a system.

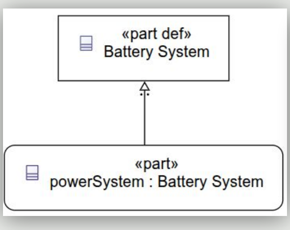
Part Definition vs. Part Usage

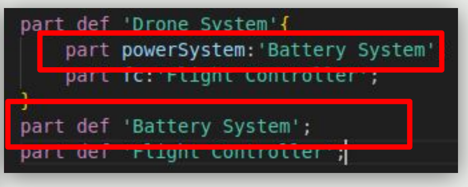

Part Definition and Usage
Part Definition
defines what a component is — a reusable type. Example: part def BatterySystem
Part Usage
defines how and where the component is used in a system. Example: battery: BatterySystem

Think of it like this:

- Part Definition = Blueprint
- Part Usage = Real Part in an Assembly

A single Part Definition can have **many Usages** in different places — all customized and connected differently.





```
part def 'Drone System'{
    part powerSystem:'Battery System'
    part fc:'Flight Controller';
}
part def 'Battery System';
part def 'Flight Controller';
```

Part Definition vs. Part Usage – What's the Difference?

This slide is all about making sure you will **never confuse a Definition and a Usage** — it's one of the most common early mistakes. Think of the Part Definition as a blueprint or template — just like in object-oriented programming, where a *class* defines a type and an *object* is an instance of that class.

So when you say battery: BatterySystem, you're creating a **real-world instantiation** of the concept — a specific battery that now lives inside a drone, and might later connect to something like a flight controller. Multiple drones could all have a battery: BatterySystem usage — and they can vary in their properties, like capacity or power draw.

This distinction is fundamental to everything else in SysML v2 modeling — structure, behavior, and interfaces all depend on it.

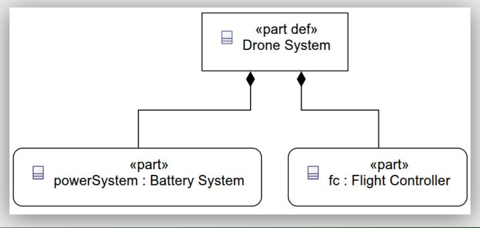
Modeling System Hierarchy

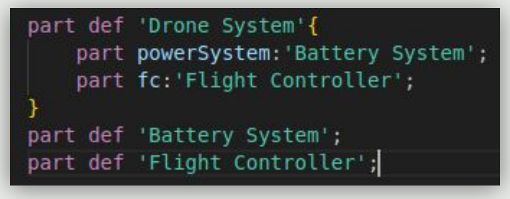
To represent a system's internal structure, we place **Part Usages inside a higher-level Part Definition**. This models the idea of a **system composed of subsystems or components** — like nesting boxes (using {}). For example:

part def DroneSystem contains:

- powerSystem: BatterySystem
- fc: FlightController

This is called **composition** — building complex systems from simpler parts. Each level in the hierarchy can itself contain further parts — allowing **multi-level decomposition**.

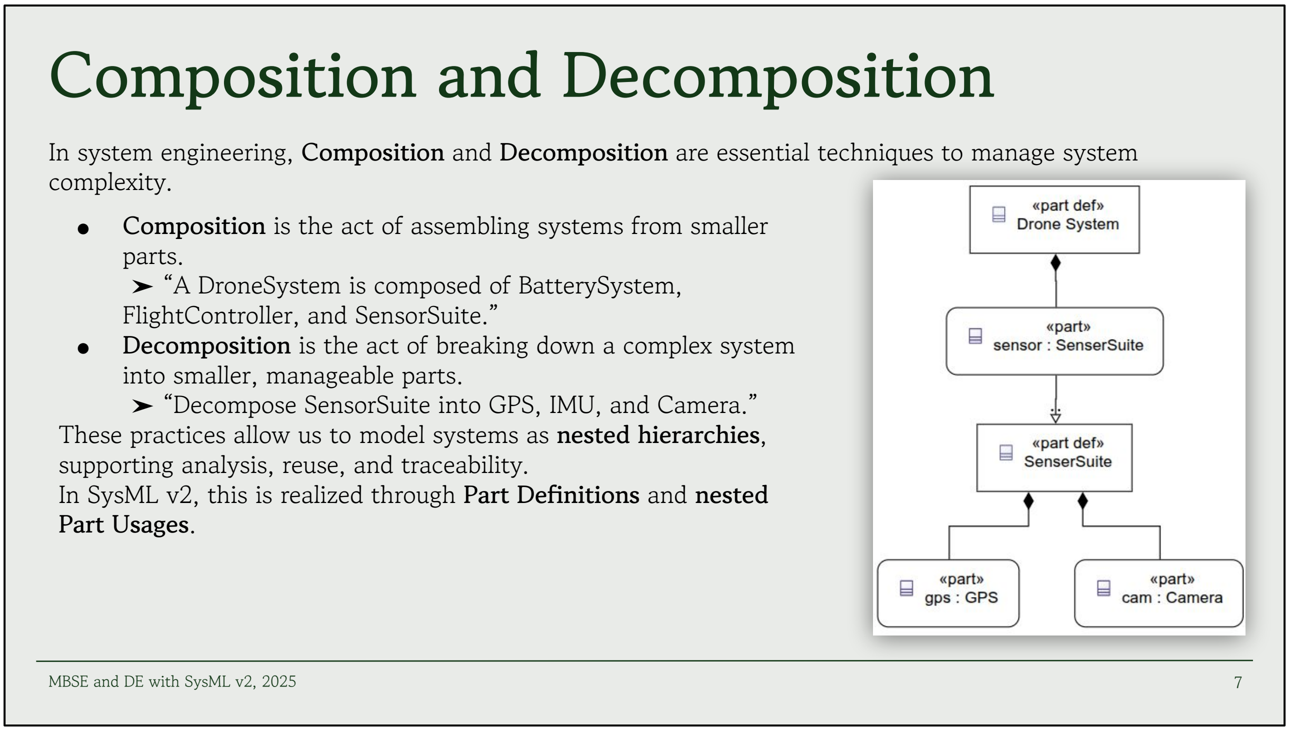




```
part def 'Drone System'{
    part powerSystem:'Battery System';
    part fc:'Flight Controller';
}
part def 'Battery System';
part def 'Flight Controller';
```

Text Notation showing the structure of Drone System

Modeling System Hierarchy using Part Usages Inside Part Definitions
Now that we've defined parts and usages, the next step is to **compose them into a structured hierarchy**. In SysML v2, this is done by placing Part Usages **inside** a Part Definition (using { })
So if DroneSystem is your top-level definition, and you want to include a battery, a flight controller, and a sensor — you define those usages inside it. Each one refers to another Part Definition, like BatterySystem.
This approach supports **modularity** and **scalability** — you can zoom into any level of the system to see its inner parts, or treat it as a black box when looking from above. And later on, these usages will be the points where we add ports, interfaces, and behavior models.

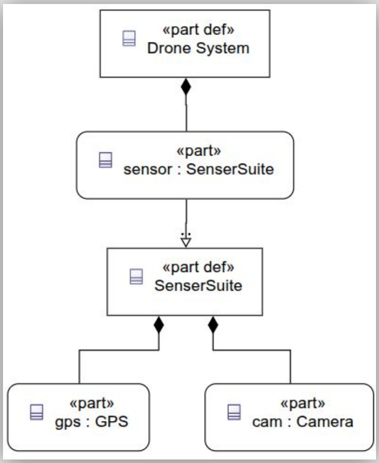


Composition and Decomposition

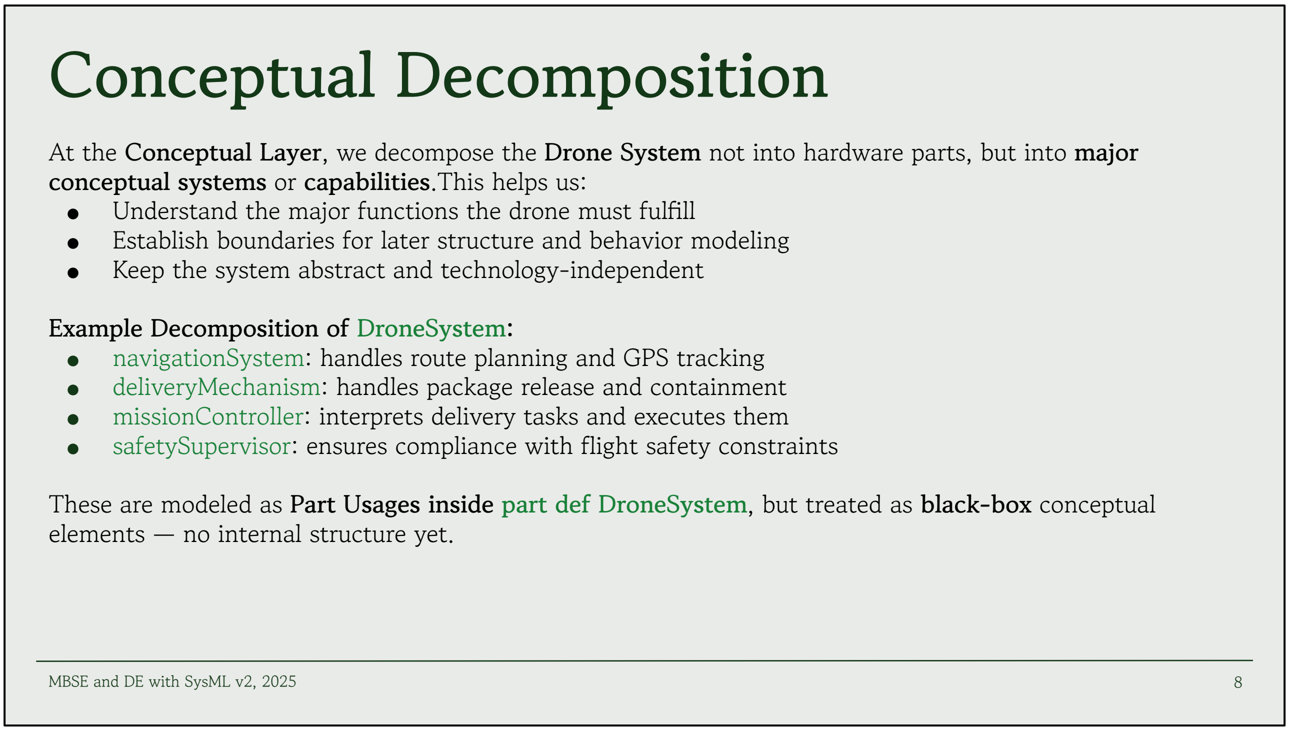
In system engineering, **Composition** and **Decomposition** are essential techniques to manage system complexity.

- **Composition** is the act of assembling systems from smaller parts.
  ➤ "A DroneSystem is composed of BatterySystem, FlightController, and SensorSuite."
- **Decomposition** is the act of breaking down a complex system into smaller, manageable parts.
  ➤ "Decompose SensorSuite into GPS, IMU, and Camera."

These practices allow us to model systems as **nested hierarchies**, supporting analysis, reuse, and traceability.
In SysML v2, this is realized through **Part Definitions** and **nested Part Usages**.



Composition and Decomposition – The Engineering Strategy
When systems get too large, engineers don't try to model them all at once. We **compose** systems from building blocks and **decompose** them into smaller parts for clarity, focus, and better integration.
Composition is like assembling LEGO bricks into a structure. Decomposition is like disassembling a machine to see how it works. Both are about **controlling complexity**, enabling **parallel development**, and making the system **modular and testable**.
In SysML v2, we achieve this through the structural arrangement of Part Definitions and nested Part Usages — which we'll now explore with real examples.



Conceptual Decomposition

At the **Conceptual Layer**, we decompose the **Drone System** not into hardware parts, but into **major conceptual systems** or **capabilities**.This helps us:

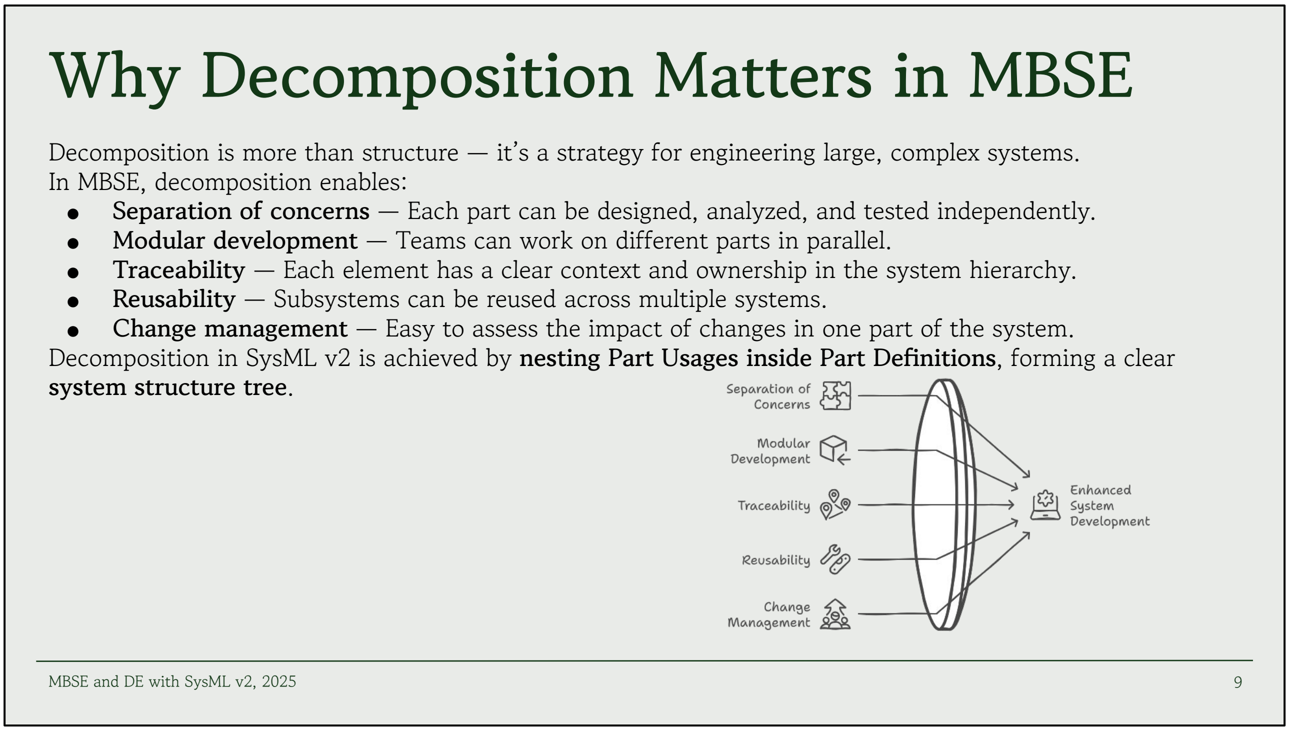
- Understand the major functions the drone must fulfill
- Establish boundaries for later structure and behavior modeling
- Keep the system abstract and technology-independent

Example Decomposition of DroneSystem:

- navigationSystem: handles route planning and GPS tracking
- deliveryMechanism: handles package release and containment
- missionController: interprets delivery tasks and executes them
- safetySupervisor: ensures compliance with flight safety constraints

These are modeled as **Part Usages inside part def DroneSystem**, but treated as **black-box** conceptual elements — no internal structure yet.

Conceptual Decomposition – Drone System as a Set of High-Level Capabilities
When we decompose at the conceptual level, we're not thinking in terms of processors, batteries, or sensors yet. Instead, we identify **core conceptual components** — subsystems that reflect stakeholder needs, use case responsibilities, and operational roles.
For example, if one of your stakeholder needs was "autonomous route following," then a navigationSystem is a natural conceptual part. You don't need to define how it works — just that it exists and will later be refined.
This black-box view is perfect for creating a **System Context**, since we want to know *what the system does*, *who it interacts with*, and *what it's composed of* — without yet committing to specific technologies.

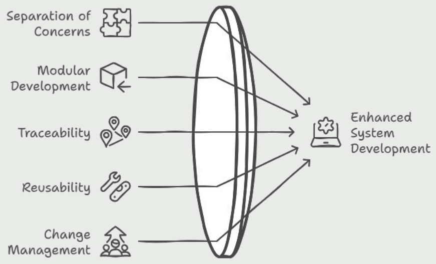


Why Decomposition Matters in MBSE

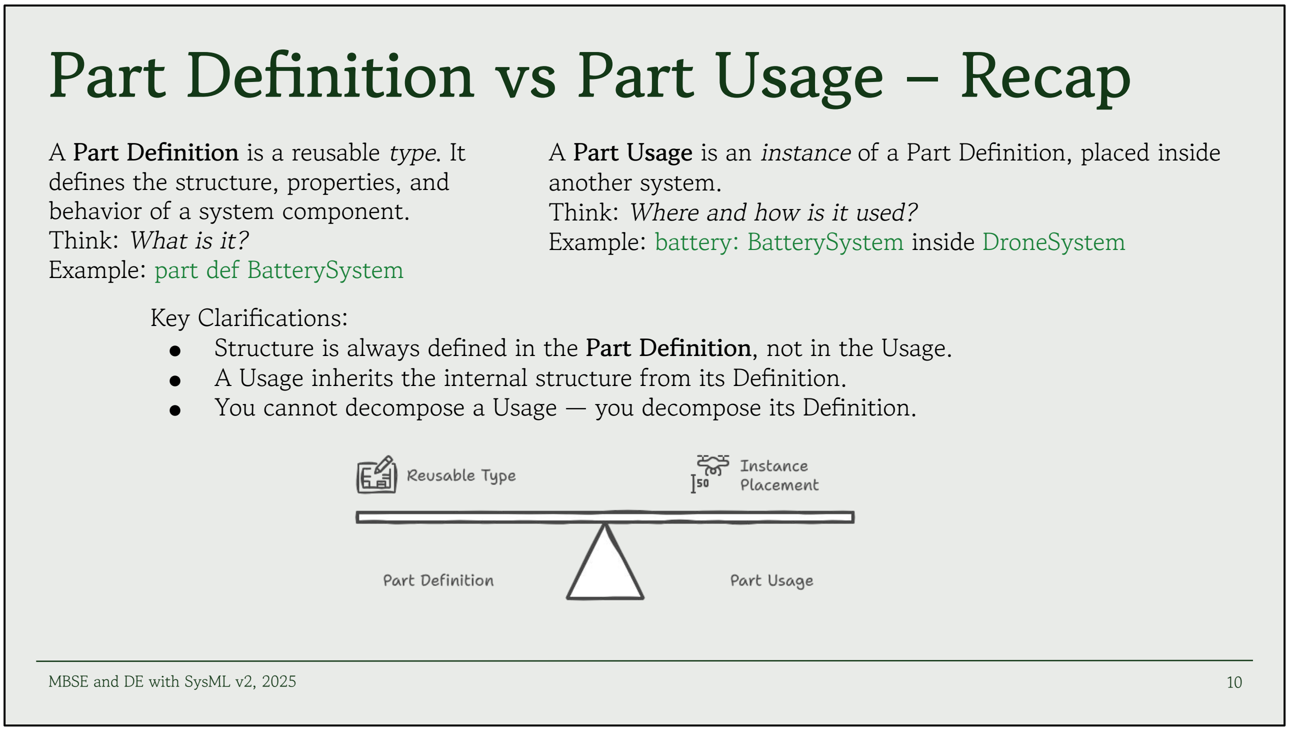
Decomposition is more than structure — it’s a strategy for engineering large, complex systems.
In MBSE, decomposition enables:

- **Separation of concerns** — Each part can be designed, analyzed, and tested independently.
- **Modular development** — Teams can work on different parts in parallel.
- **Traceability** — Each element has a clear context and ownership in the system hierarchy.
- **Reusability** — Subsystems can be reused across multiple systems.
- **Change management** — Easy to assess the impact of changes in one part of the system.

Decomposition in SysML v2 is achieved by **nesting Part Usages inside Part Definitions**, forming a clear **system structure tree**.



Why Decomposition Matters in Model-Based Systems Engineering (MBSE)
In real engineering, complexity is the enemy of progress. MBSE gives us the tools to manage complexity, and decomposition is one of the most powerful strategies. It helps you go from chaos to clarity — by organizing the system into logical parts that can be modeled, tested, and evolved independently.
This structural clarity also supports other layers of modeling — requirements, behavior, interfaces, and verification — all benefit when you decompose your system well. And when you need to change something? You only touch that part of the tree — and trace where the impact goes.
SysML v2 enables this cleanly by using nested Part Usages inside Part Definitions. It's like building a digital twin from the inside out — one piece at a time.



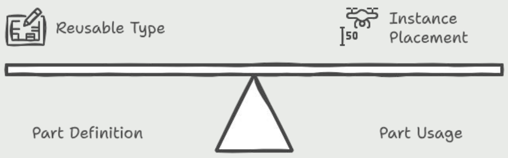
Part Definition vs Part Usage – Recap

A **Part Definition** is a reusable *type*. It defines the structure, properties, and behavior of a system component.
Think: *What is it?*
Example: part def BatterySystem

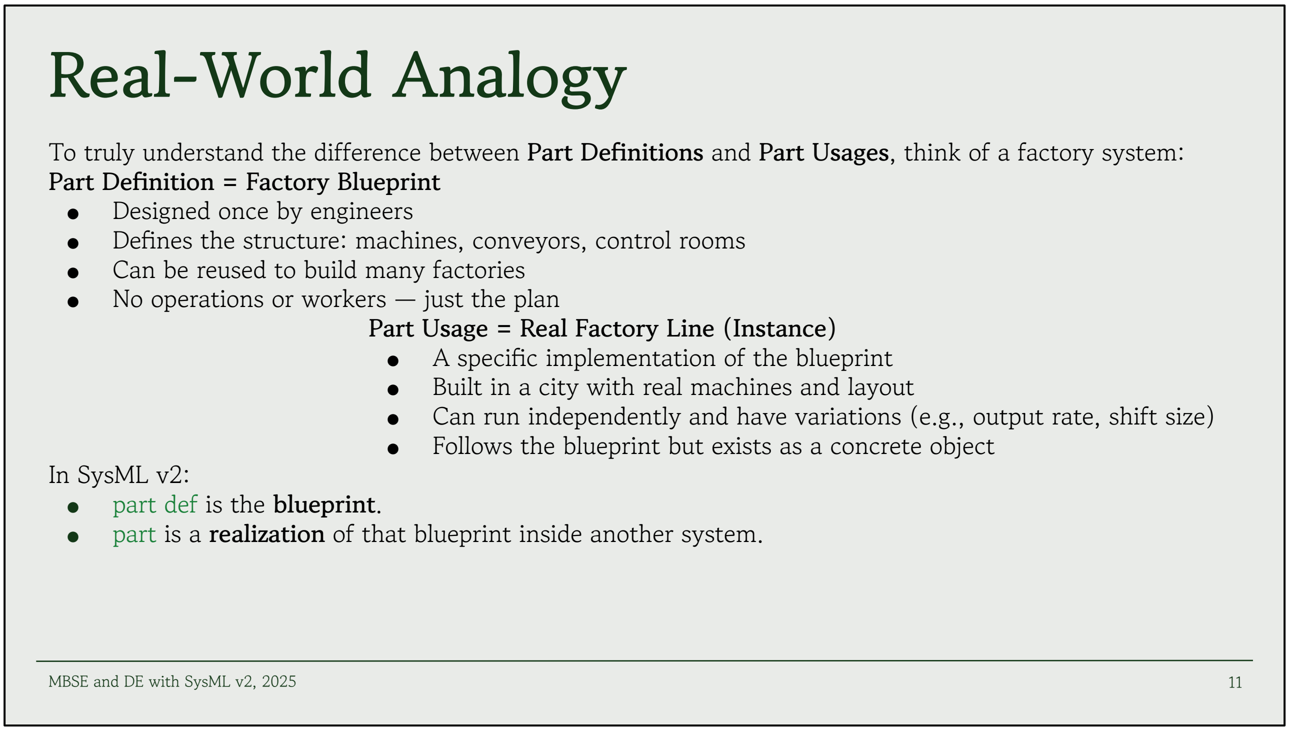
A **Part Usage** is an *instance* of a Part Definition, placed inside another system.
Think: *Where and how is it used?*
Example: battery: BatterySystem inside DroneSystem

Key Clarifications:

- Structure is always defined in the **Part Definition**, not in the Usage.
- A Usage inherits the internal structure from its Definition.
- You cannot decompose a Usage — you decompose its Definition.



Part Definition vs Part Usage – Recap & Clarification
Let’s make sure this is 100% clear before we move forward. A **Part Definition** is where you describe the *what* — it’s like your library of component types. A **Part Usage** is where you say *this system includes this component*. That usage is placed inside another system and inherits all the structure defined in the part def.
If someone asks:
“Where do you define the subparts of a battery?”
The answer is: inside part def BatterySystem, **never inside the usage** like battery: BatterySystem.
This mental separation is what keeps your model consistent, scalable, and modular. It also helps you align everything — structure, behavior, requirements — around a common, reusable base.



Real-World Analogy

To truly understand the difference between **Part Definitions** and **Part Usages**, think of a factory system:

Part Definition = Factory Blueprint

- Designed once by engineers
- Defines the structure: machines, conveyors, control rooms
- Can be reused to build many factories
- No operations or workers — just the plan

Part Usage = Real Factory Line (Instance)

- A specific implementation of the blueprint
- Built in a city with real machines and layout
- Can run independently and have variations (e.g., output rate, shift size)
- Follows the blueprint but exists as a concrete object

In SysML v2:

- part def is the **blueprint**.
- part is a **realization** of that blueprint inside another system.

Real-World Analogy – Factory Blueprint vs Factory Line

Let's visualize this with a physical metaphor — a **factory**. Think of a **Part Definition** like the architectural plan of a factory. It shows where each station goes, how things connect, and what kind of machines are needed — but it's just a drawing.

Now think of the **Part Usage** as an actual working factory, built from that blueprint, sitting in Bangkok or Chiang Mai. It has specific machines, employees, and operational conditions. You can build multiple factories from the same blueprint — and that's exactly what happens when we reuse a Part Definition in multiple systems.

This analogy helps anchor what we've been learning — and prevents one of the biggest early mistakes: **trying to define structure in a usage** instead of its definition.

Common Beginner Mistakes in Structure Modeling

When modeling system structure in SysML v2, beginners often make these errors:

Defining Internal Structure in a Part Usage
Example: Trying to add parts inside battery: BatterySystem
Structure must be defined inside part def BatterySystem

Duplicating Part Definitions for Similar Components
Example: Creating BatterySystem1, BatterySystem2, etc.
Use the same BatterySystem definition with different usages

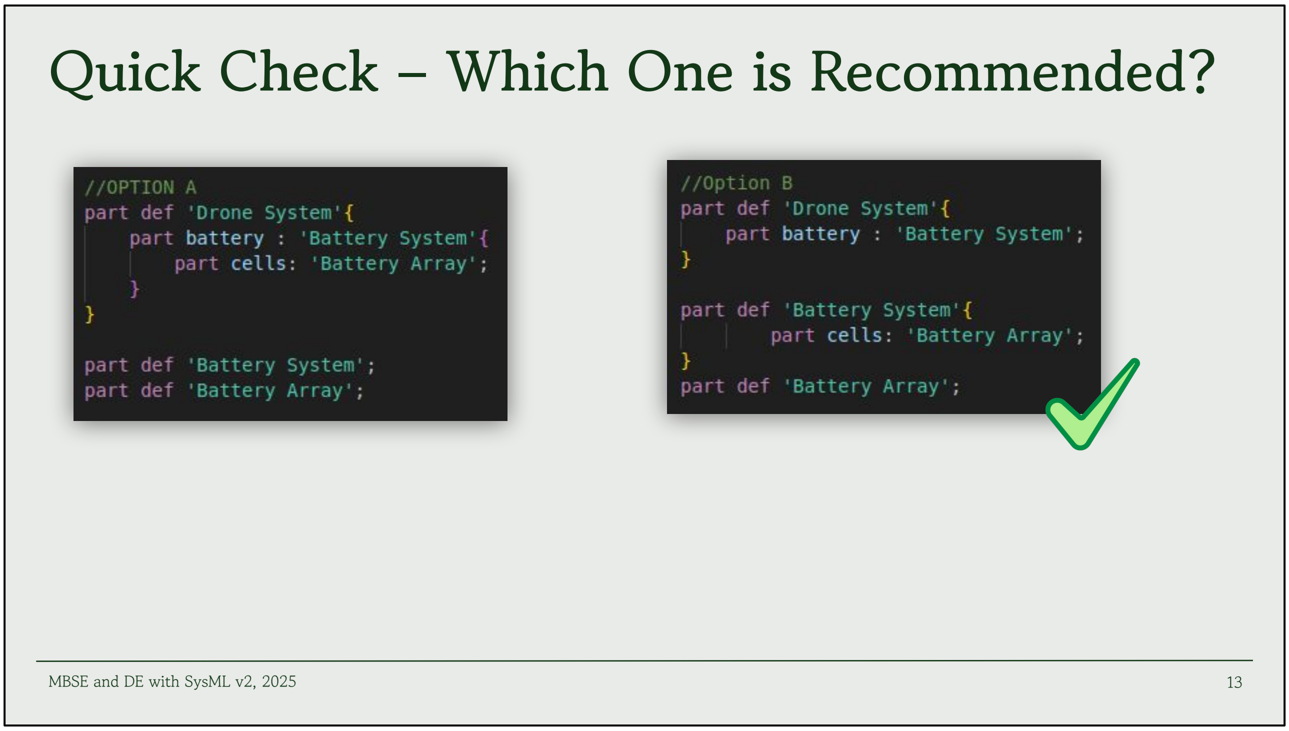
Modeling the Whole System Flat (No Decomposition)
Putting all parts at the top level
Decompose logically: DroneSystem → SensorSuite → GPS, IMU, Camera

Tip: Keep your model clean and scalable by using:

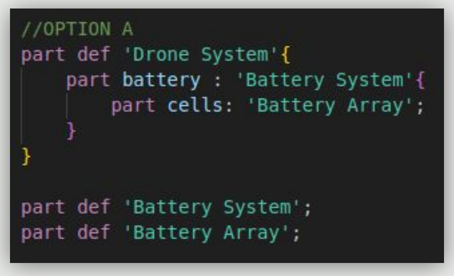
- Clear part defs for reusable components
- Proper hierarchy with nested part usages
- Structure only inside definitions — never inside usages

Common Beginner Mistakes to Avoid

These are traps almost every beginner falls into — and they can really mess up your model's integrity. The most dangerous one is trying to define structure inside a part usage — it won't work, and you'll be confused when nothing connects properly.
The other issue is thinking you need a new definition for every variation — when actually, that's what Part Usages are for. You can reuse one BatterySystem definition and give each usage its own parameters.
Encouraging decomposition from the start builds a model that scales well and mirrors real systems more closely.

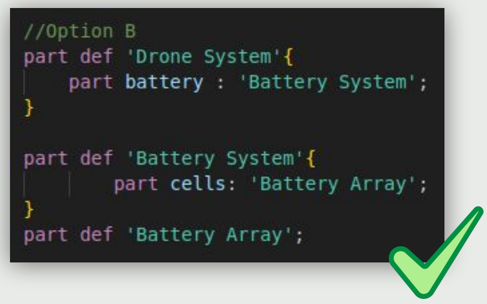


Quick Check – Which One is Recommended?



```
//OPTION A
part def 'Drone System'{
    part battery : 'Battery System'{
        part cells: 'Battery Array';
    }
}

part def 'Battery System';
part def 'Battery Array';
```

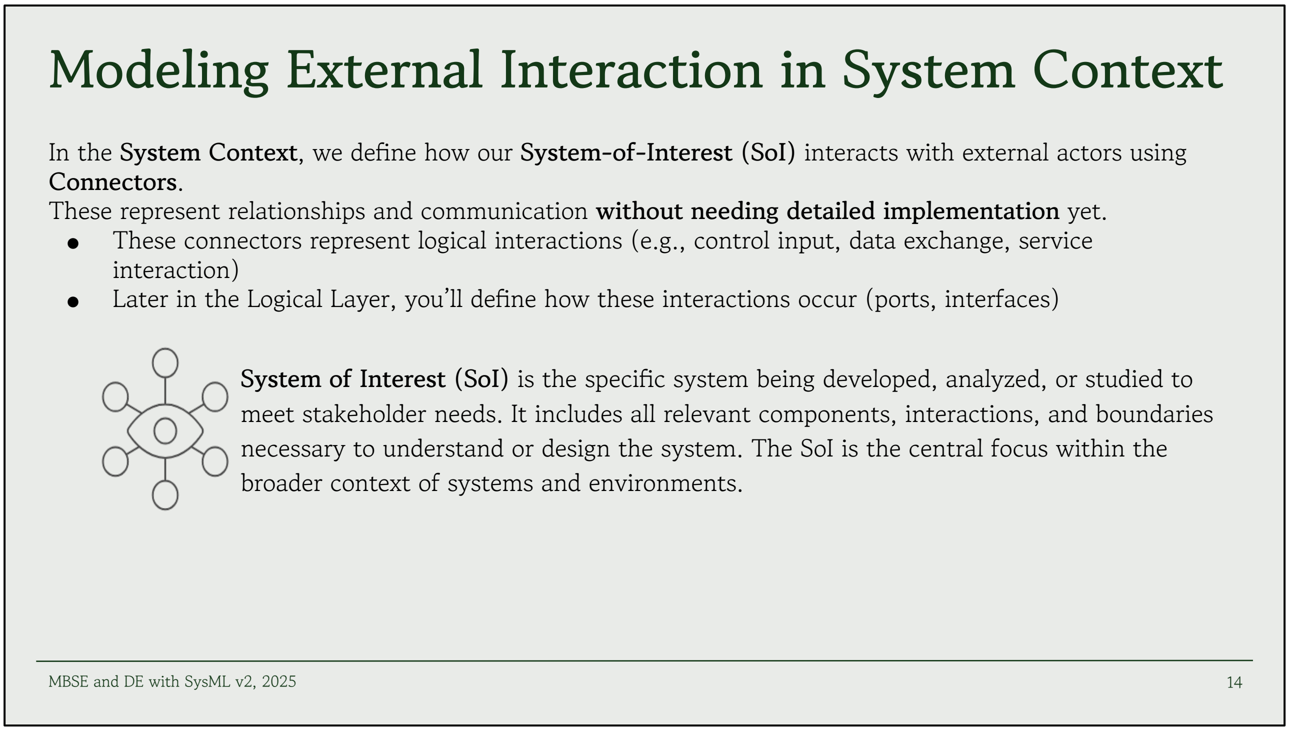


```
//Option B
part def 'Drone System'{
    part battery : 'Battery System';
}

part def 'Battery System'{
        part cells: 'Battery Array';
}
part def 'Battery Array';
```

✓

Quick Check – Which One is Correct?



Modeling External Interaction in System Context

In the **System Context**, we define how our **System-of-Interest (SoI)** interacts with external actors using **Connectors**.
These represent relationships and communication **without needing detailed implementation** yet.

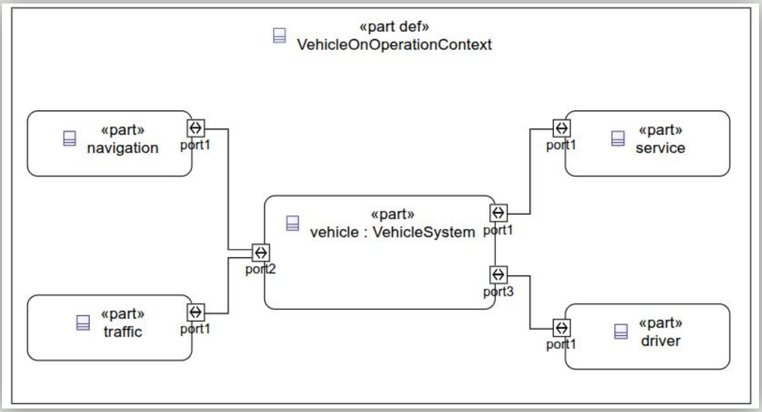
- These connectors represent logical interactions (e.g., control input, data exchange, service interaction)
- Later in the Logical Layer, you'll define how these interactions occur (ports, interfaces)

System of Interest (SoI) is the specific system being developed, analyzed, or studied to meet stakeholder needs. It includes all relevant components, interactions, and boundaries necessary to understand or design the system. The SoI is the central focus within the broader context of systems and environments.

Modeling External Interactions in the System Context
At this stage, we're not concerned about *how* signals are sent — just that the **relationship exists**. This prepares us for interface modeling later and keeps our model structured at the **conceptual** level.

Visual Example – System Context

Here’s a visual representation of the **System Context** for a generic **Vehicle System**, showing how it interacts with its environment.
System-of-Interest: VehicleSystem



Visual Example – System Context of a Vehicle
This visual system context is what you will similarly build for your **own Drone Systems**, but without using this exact example. It keeps things general while letting them practice **modeling boundaries and connectors**.
Use this model in class to explain **how to identify actors**, **draw the boundary**, and **create meaningful conceptual links**. No ports yet — just clean context-level structure based on use cases and stakeholder expectations.

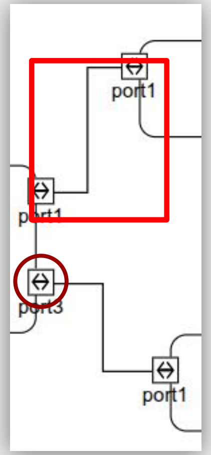
What about Ports and Connectors?

In full SysML v2, it's valid to create **connectors directly between Part Usages** — without using Ports. However, in **SysON**, Ports are **required** to create visual connections. So for now:

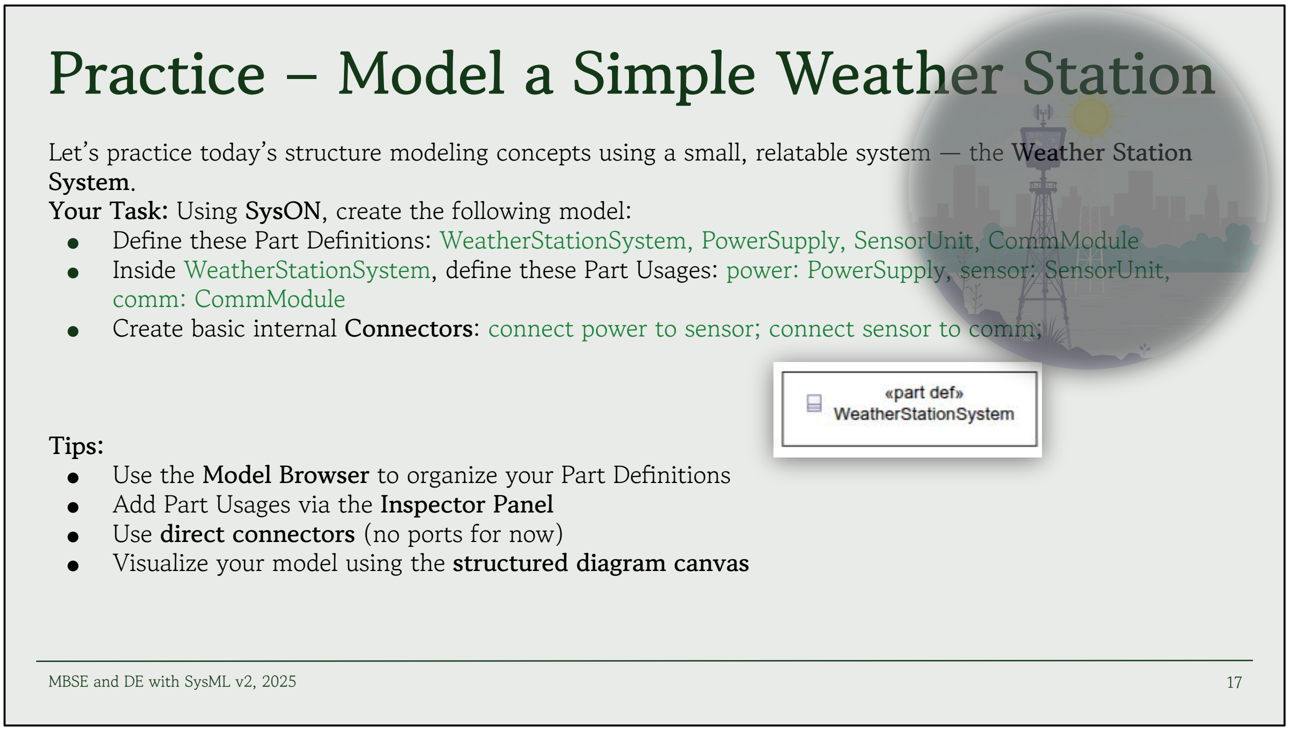
- We will create **simple Ports** on each Part Usage
- But we **won't explain their meaning yet**
- We will only use them to support **basic system context modeling**

Port types, interfaces, connector, and flow semantics will be covered in **Week 8**
This lets us keep the focus on:

- Structure and hierarchy
- Part Definitions and Usages
- System Context boundaries



What about Port and Connection?
Although SysML v2 allows you to model connections without Ports, tools like SysON need them to show connections visually. So for now, we'll **create the Ports as placeholders**, but you don't need to worry about naming them precisely or understanding what they mean yet. We'll revisit Ports properly in Week 8 when we discuss Interfaces.



Practice – Model a Simple Weather Station

Let's practice today's structure modeling concepts using a small, relatable system — the **Weather Station System**.
Your Task: Using **SysON**, create the following model:

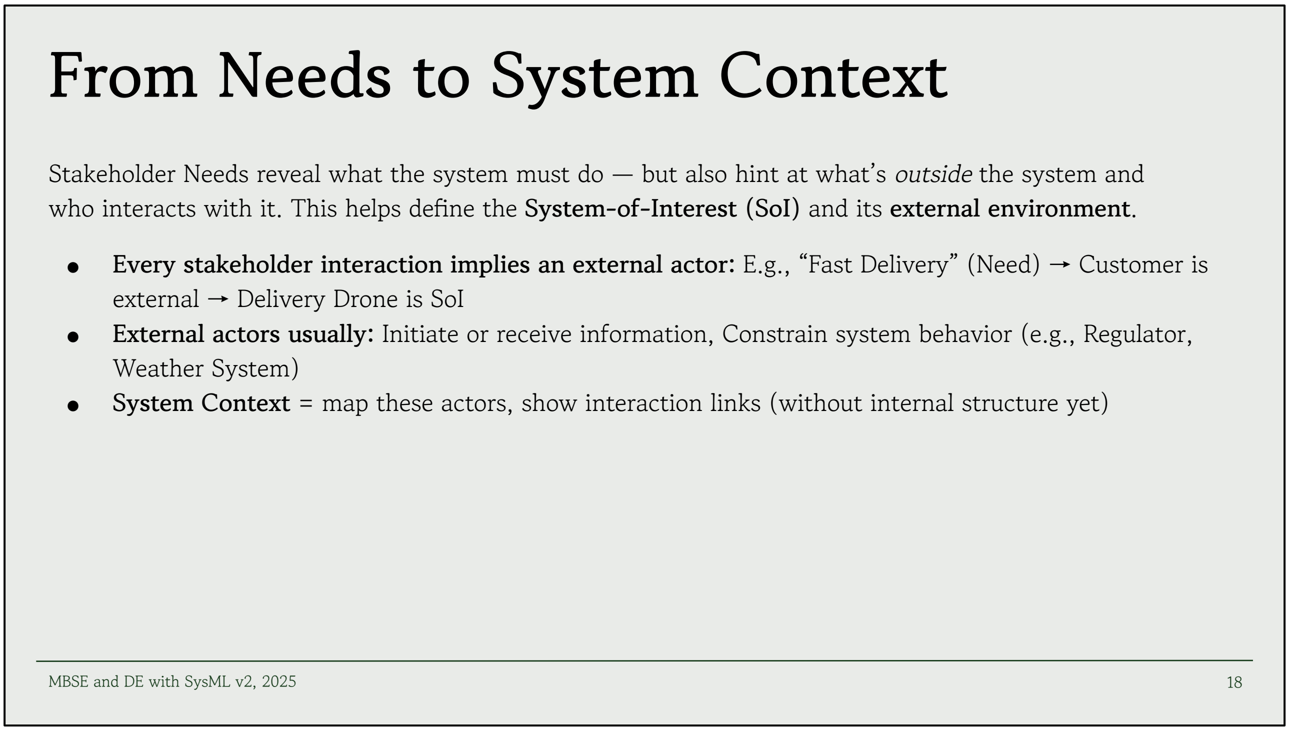
- Define these Part Definitions: WeatherStationSystem, PowerSupply, SensorUnit, CommModule
- Inside WeatherStationSystem, define these Part Usages: power: PowerSupply, sensor: SensorUnit, comm: CommModule
- Create basic internal **Connectors**: connect power to sensor; connect sensor to comm;

«part def»
WeatherStationSystem

Tips:

- Use the **Model Browser** to organize your Part Definitions
- Add Part Usages via the **Inspector Panel**
- Use **direct connectors** (no ports for now)
- Visualize your model using the **structured diagram canvas**

Practice: Model a Simple Weather Station System in SysON
This practice model gives you a safe, simple space to apply what you've learned — without touching your main project yet. The goal is to model clean hierarchy, proper Part Definitions and Usages, and internal connectors using SysON.
It's lightweight, but powerful — you'll get comfortable with the modeling tool and how SysML v2 structure works in practice.
After this, you'll be more than ready to begin structuring your **Autonomous Drone System** with confidence.

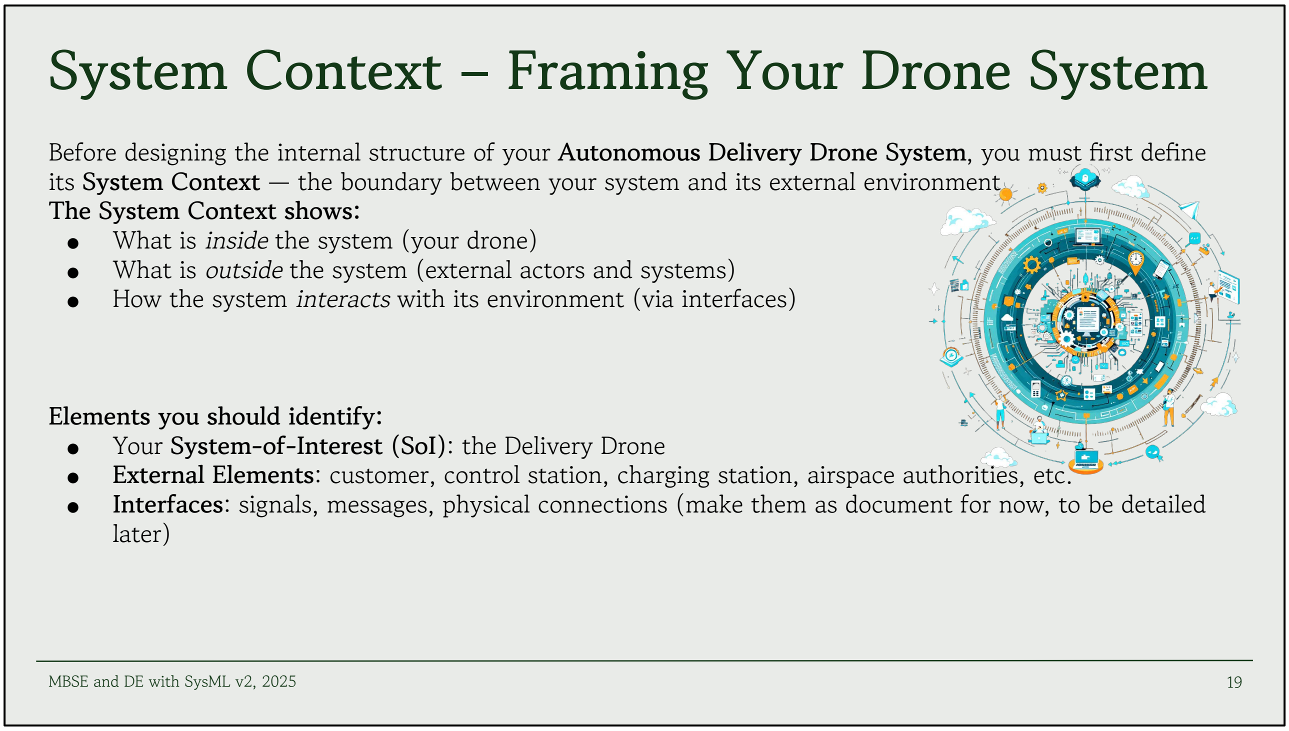


From Needs to System Context

Stakeholder Needs reveal what the system must do — but also hint at what's *outside* the system and who interacts with it. This helps define the **System-of-Interest (SoI)** and its **external environment**.

- **Every stakeholder interaction implies an external actor:** E.g., "Fast Delivery" (Need) → Customer is external → Delivery Drone is SoI
- **External actors usually:** Initiate or receive information, Constrain system behavior (e.g., Regulator, Weather System)
- **System Context** = map these actors, show interaction links (without internal structure yet)

From Needs to System Context
This is about the reasoning process behind defining the system boundary — something many beginners skip.
We don't just guess what's inside vs outside. Instead, we trace it back from what the system is *supposed to do*.
If a stakeholder wants fast delivery, someone has to initiate that — the **Customer** or **Operator** — and that makes them an **external actor**.
That's how we define the System Context: from **Needs**, we extract the actors and link them to the **System-of-Interest**.

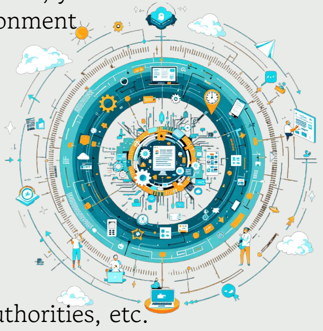


System Context – Framing Your Drone System

Before designing the internal structure of your **Autonomous Delivery Drone System**, you must first define its **System Context** — the boundary between your system and its external environment.

The System Context shows:

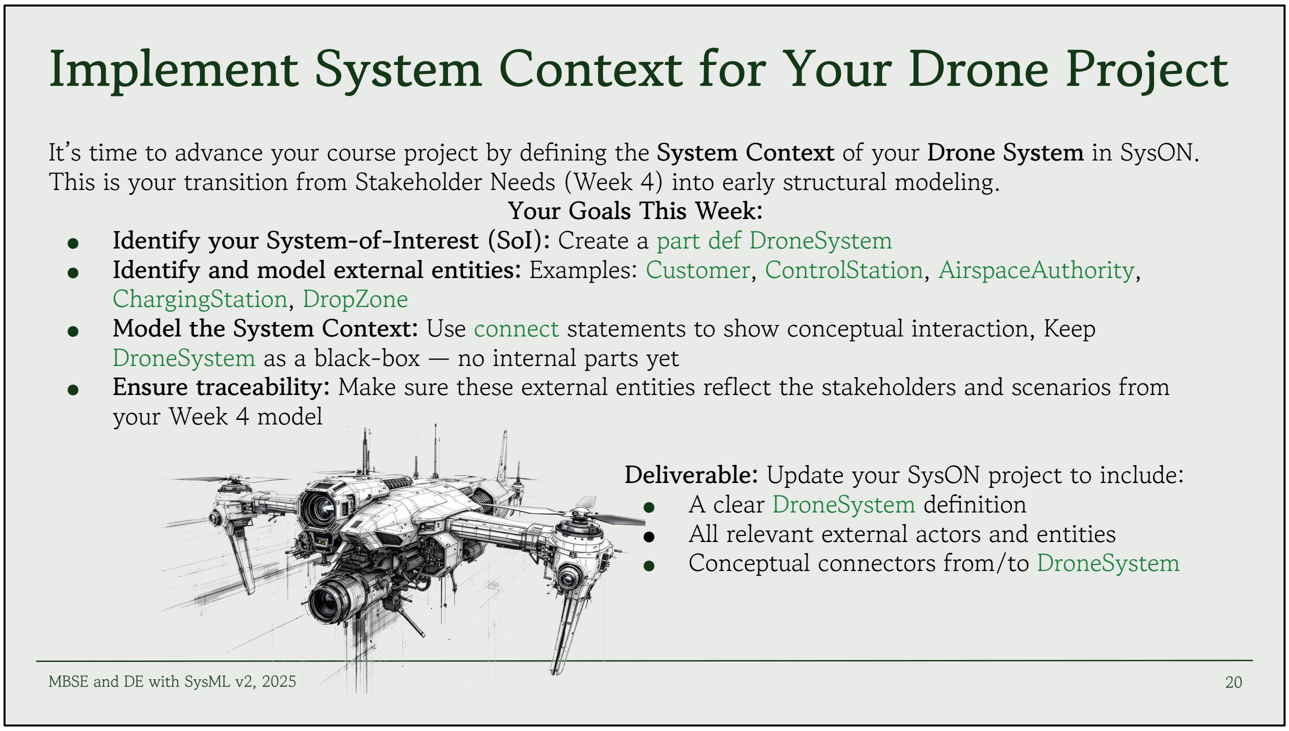
- What is *inside* the system (your drone)
- What is *outside* the system (external actors and systems)
- How the system *interacts* with its environment (via interfaces)



Elements you should identify:

- Your **System-of-Interest (SoI)**: the Delivery Drone
- **External Elements**: customer, control station, charging station, airspace authorities, etc.
- **Interfaces**: signals, messages, physical connections (make them as document for now, to be detailed later)

System Context – Framing Your Drone System
The System Context sets the stage for everything that comes after. It's like drawing a box around your system and asking:
"What does my system need to interact with in order to fulfill its purpose?"
Before we dive into modeling the internal parts of your drone, take a moment to define what's **outside** and how your system connects to it. This ensures that your architecture serves the needs and scenarios you modeled last week.
Later, when we define Ports and Interfaces, we'll return to this System Context and make those interactions explicit.
So as we begin your project modeling, **always start from the system boundary**.

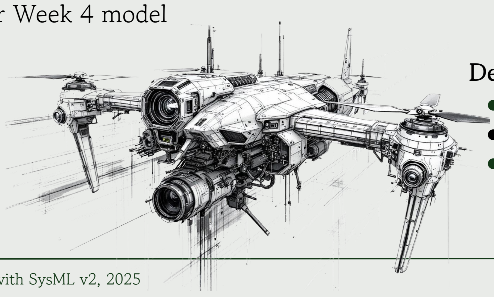


Implement System Context for Your Drone Project

It’s time to advance your course project by defining the **System Context** of your **Drone System** in SysON. This is your transition from Stakeholder Needs (Week 4) into early structural modeling.

Your Goals This Week:

- **Identify your System-of-Interest (SoI):** Create a part def DroneSystem
- **Identify and model external entities:** Examples: Customer, ControlStation, AirspaceAuthority, ChargingStation, DropZone
- **Model the System Context:** Use connect statements to show conceptual interaction, Keep DroneSystem as a black-box — no internal parts yet
- **Ensure traceability:** Make sure these external entities reflect the stakeholders and scenarios from your Week 4 model



Deliverable: Update your SysON project to include:

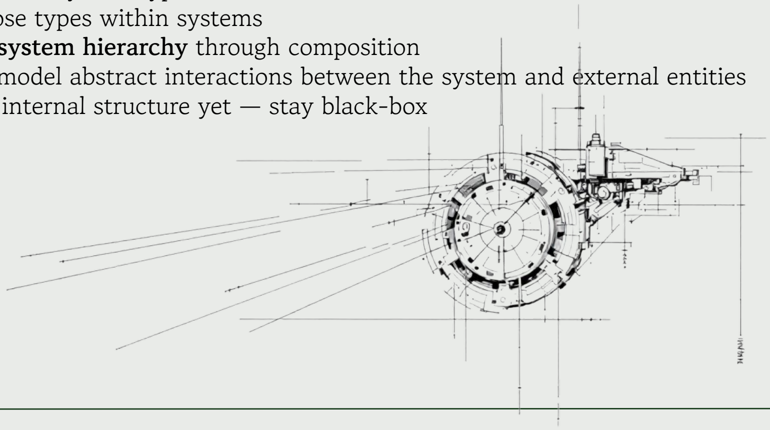
- A clear DroneSystem definition
- All relevant external actors and entities
- Conceptual connectors from/to DroneSystem

Implement the System Context for Your Autonomous Drone System
Now that you’ve practiced with a Weather Station, it’s time to return to your own system. But remember — this week, you're **not designing the architecture yet.**
Your mission is to define your **System Context**: draw the boundary, identify the external world, and declare the interactions. This sets the foundation for logical architecture later.
Keep it abstract, clean, and traceable to stakeholder needs. No ports. No internals. Just pure MBSE conceptual discipline.

Summary of Week 5

This week, we explored how to represent **system structure** at the **Conceptual Layer** using **Part Definitions**, **Part Usages**, and **Connectors**.We stayed focused on building the **System Context**, not internal architecture.**Key Concepts Covered:**

- part def defines reusable system types
- part instantiates those types within systems
- Part Usages form a **system hierarchy** through composition
- Use **Connectors** to model abstract interactions between the system and external entities
- Avoid decomposing internal structure yet — stay black-box



QUESTION!