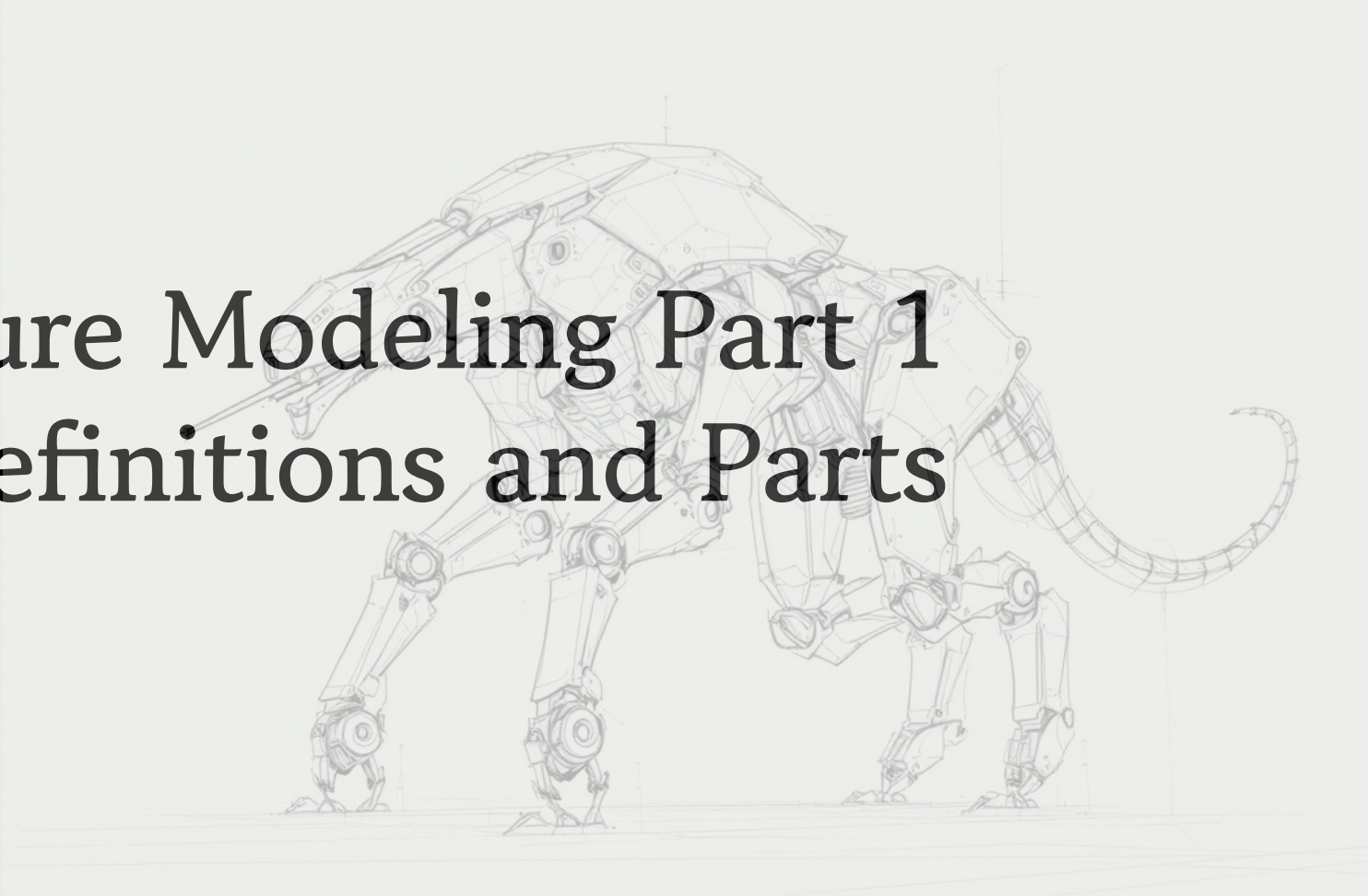


Structure Modeling Part 1

Part Definitions and Parts

Week 05



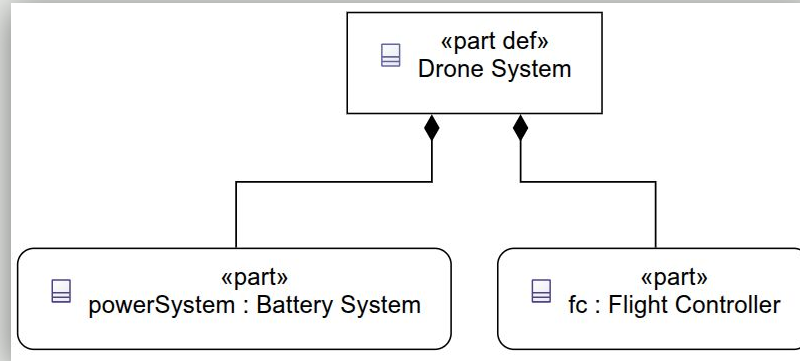
Modeling Structure

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.

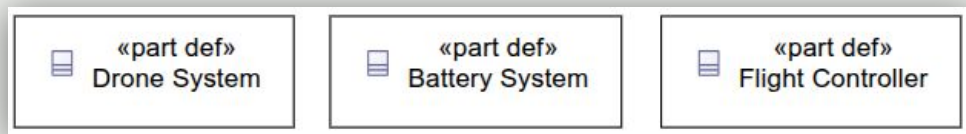


What is a Part Definition

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';
```

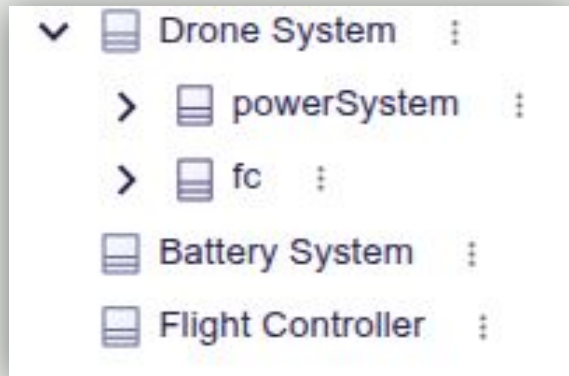
Text Notation of them

What is 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.



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

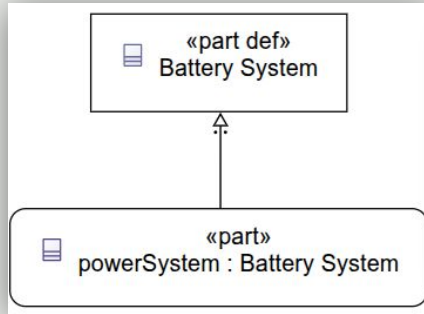
defines how and where the component is used in a system.
Example: battery: BatterySystem



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';
```

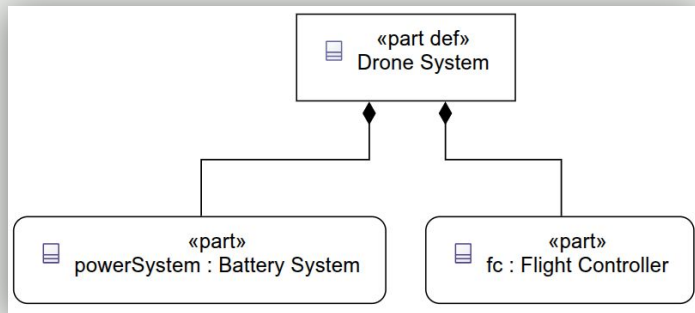
Modeling System Hierarchy

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

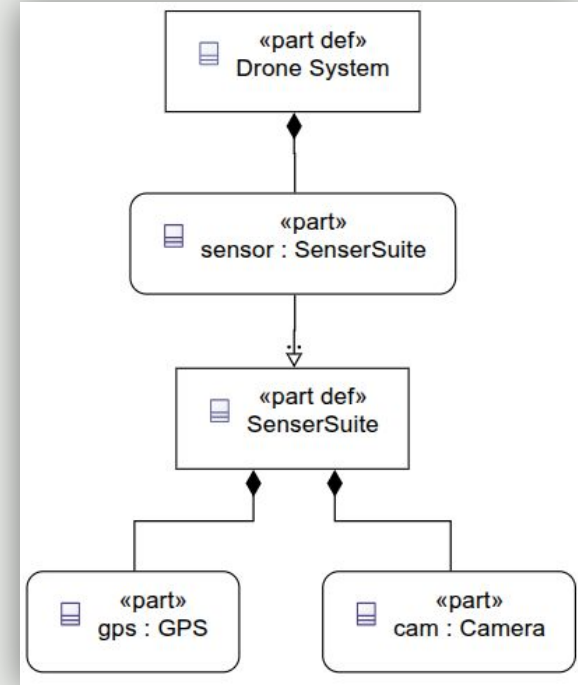
Composition and Decomposition

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**.



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.

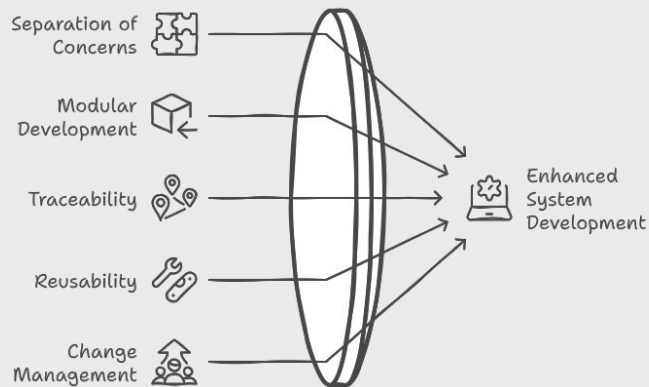
Why Decomposition Matters in MBSE

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**.



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`

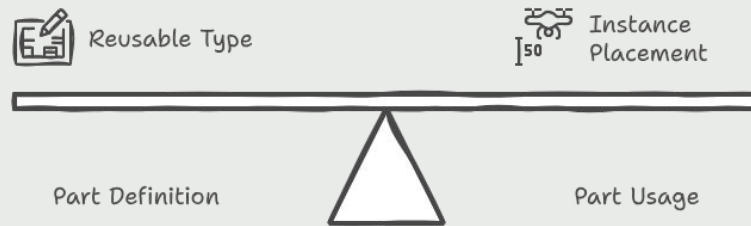
A **Part Usage** is an *instance* of a Part Definition, placed inside another system.

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.



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.

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


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';
```

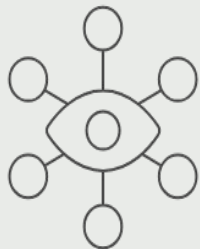


Modeling External Interaction in System Context

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)

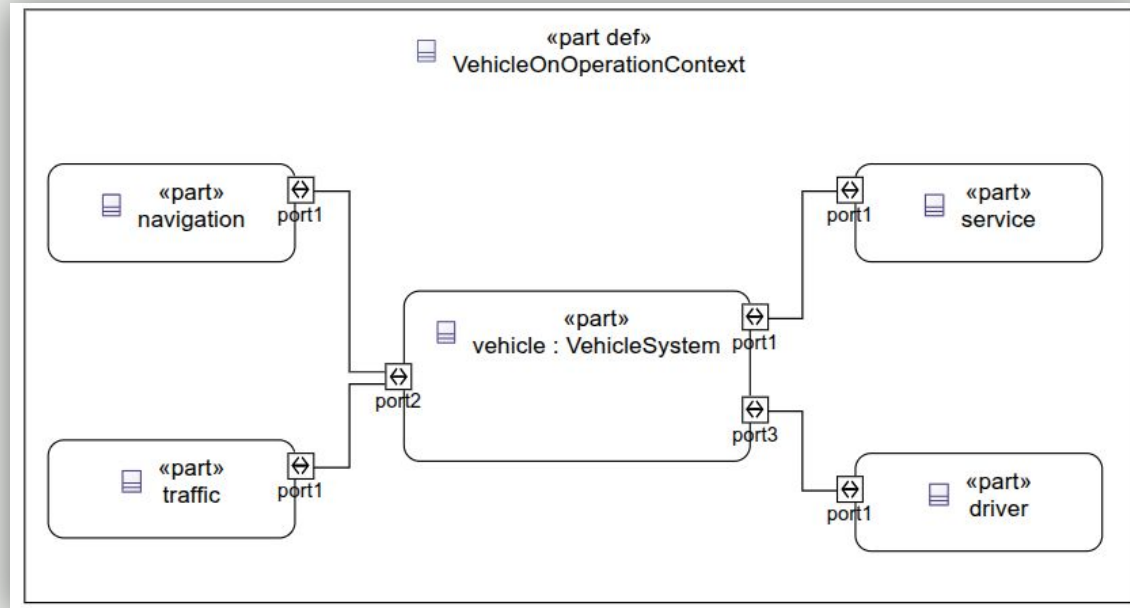


System of Interest (SoI) is the specific system being developed, analyzed, or studied to meet stakeholder needs. It includes all relevant components, interactions, and boundaries necessary to understand or design the system. The SoI is the central focus within the broader context of systems and environments.

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**



What about Ports and Connectors?

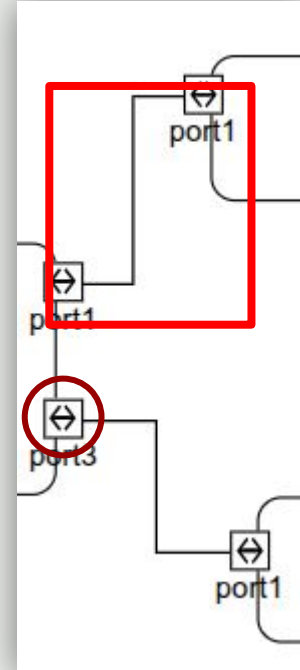
In full SysML v2, it's valid to create **connectors directly between Part Usages** — without using Ports. However, in **SysON**, Ports are **required** to create visual connections. So for now:

- 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



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**;

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**



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)

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:

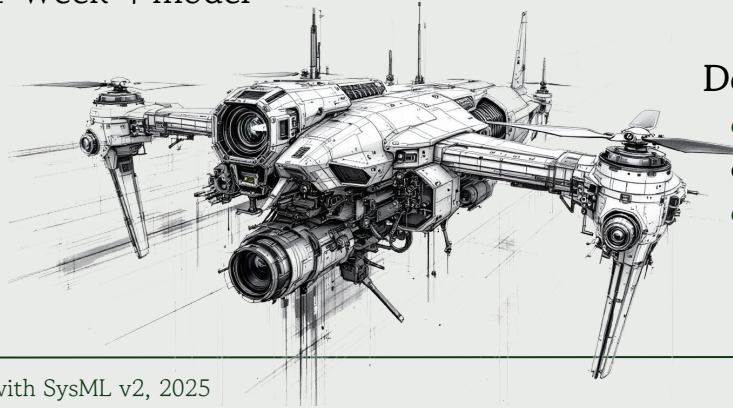
- Your **System-of-Interest (SoI)**: the Delivery Drone
- **External Elements**: customer, control station, charging station, airspace authorities, etc.
- **Interfaces**: signals, messages, physical connections (make them as document for now, to be detailed later)

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



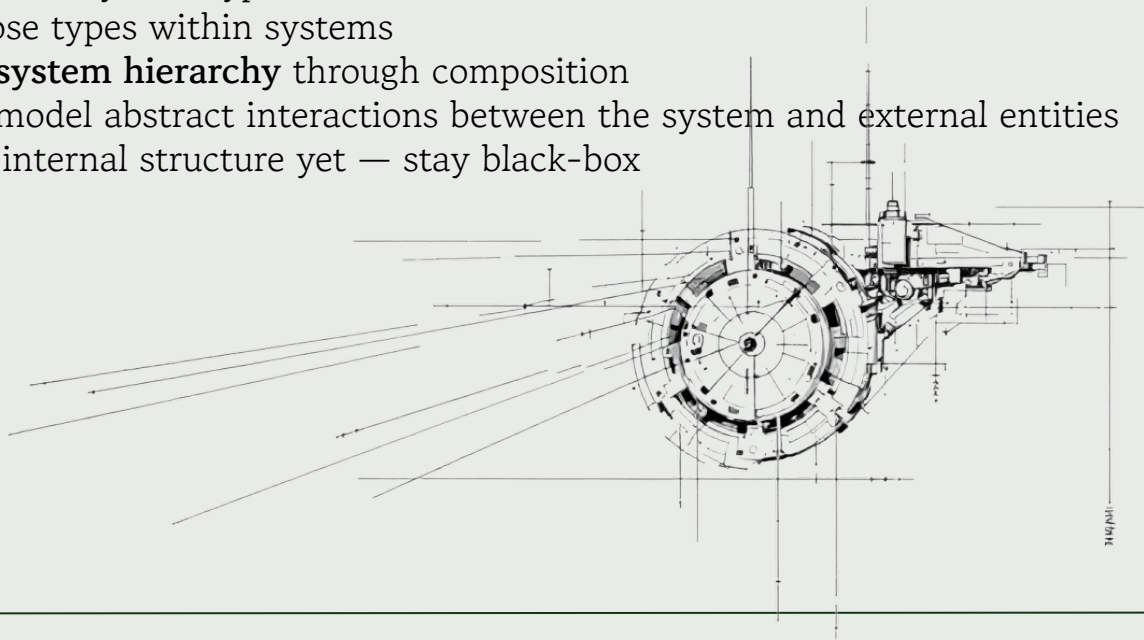
Deliverable: Update your SysON project to include:

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

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!