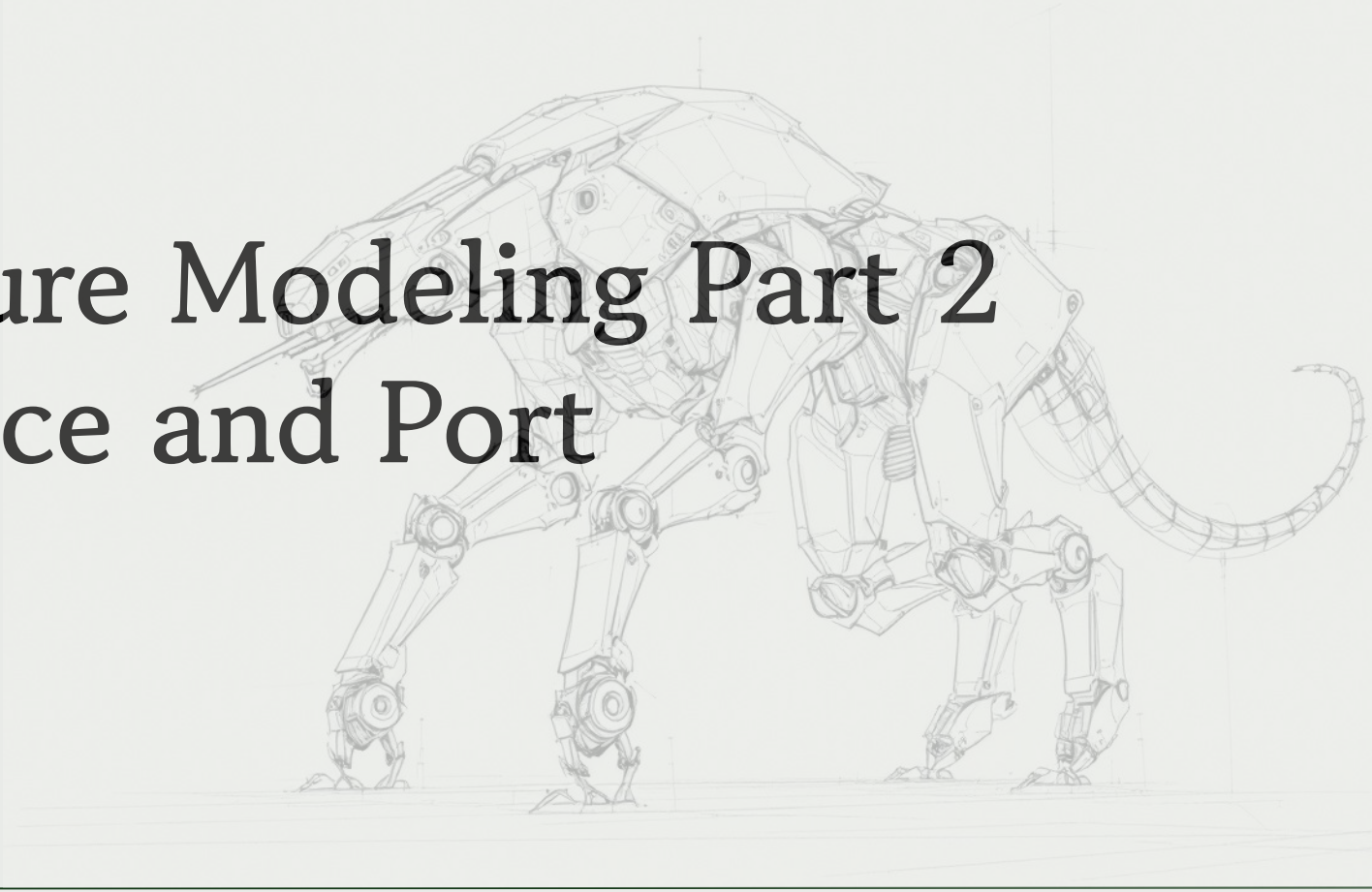
Structure Modeling Part 2 Interface and Port

Week 08



Structure Modeling Part 2

This week, you will learn how to model system interactions using Ports, Interfaces, and Flows — building a modular, traceable system architecture.
You will also revisit system decomposition and specialization to support correct architecture modeling.

Goals for Today:

- Define **Ports** (port def, port) as interaction points
- Define **Flows** using flow features and **Items** (item def, item)
- Understand **correct system decomposition** for placing Ports
- Model **Specialization** of Parts, Ports, Items, and Interfaces
- Define **Interfaces** (interface def) as connection compatibility
- Model **Interface Contracts** with Port + Item + Interface
- Practice modeling in **SysON**
- Apply Interface and Flow modeling to your **Drone System** project

Why Model Interactions and Flows?

Ports, Items, and Interfaces make system interactions explicit and traceable.
They define *where interaction occurs*, *what flows*, and *which connections are valid* — supporting robust, modular architectures.

Key Principle: If you do not model Ports and Flows explicitly — system interactions remain hidden, hard to test, and hard to maintain.

Why Model Interactions and Flows:

- **Clarify communication between Parts**:
  - Where do interactions happen? (port)
  - What flows? (item)
  - How is compatibility ensured? (interface def)
- Enable **modular and reusable architectures**:
  - Parts can be replaced or upgraded if they support the same Interface
- Support **traceability**:
  - Which Parts depend on which interactions?
  - What data, power, or signals flow across boundaries?
- Strengthen **validation**:
  - Interface Contracts define what must be tested
  - Clear separation of internal and external behavior

What Is a Port in SysML v2?

A Port represents an interaction point on a Part. It defines *where* connections are made and *what can flow* through those connections, using **flow features** and **Items**.

- In SysML v2, a **port def** defines an interaction point type.
  Each **port def** owns **flow features** that describe what flows through the Port — specifying both the **direction** and the **item def** that flows.
- A **port** is a usage of a port def, attached to a **part usage** in your system structure.
  Ports can represent any kind of interaction: **data**, **power**, **fluid**, **signal**, or **mechanical force**.
- By modeling Ports, we make interactions **explicit** — not hidden in internal behavior.
  We can also model **directionality**:
  - **in** — the Port receives flow.
  - **out** — the Port sends flow.
  - **inout** — flow is bi-directional.

The **Port defines where interaction happens** and **what flows**, by owning flow features — not by being "typed" by an Item.

What Is an Item and Flow?

An Item defines what flows through a connection. It specifies *the type of data, power, material, or signal* that is exchanged across a Port or Interface.

- In SysML v2, an **item def** defines the **type** of item that can flow through Ports.
  An **item** is an instance of this type — it represents a specific flow during system operation.
- Flows are modeled as **flow features** inside a port def.
  Each flow feature declares its **direction** (in, out, inout) and the **item def** it uses.
- You can model **any kind of flow**:
  - **Data** — telemetry, commands, sensor readings.
  - **Power** — voltage, current.
  - **Material** — fluids, gases, consumables.
  - **Signals** — logical triggers or timing events.
  - **Mechanical** — force, torque, displacement.

The **item def** defines *what flows*. The **flow feature** in a port def defines *how it flows* (direction).

Defining Ports with Flow Features

A port def defines a reusable Port type. It specifies *where interaction happens* and *what flows* through flow features.

In SysML v2, a **port def** is a definition of an interaction point.

It can be reused across multiple Parts — promoting **modularity** and **design consistency**.

A port def owns one or more **flow features**:

- Each flow feature declares a **direction** (in, out, inout).
- Each flow feature references an **item def** — specifying what flows.

```
item def TelemetryItem;

port def TelemetryPort {
  out telemetryData: TelemetryItem;
  in statusReport: StatusItem;
}
```

By defining Ports this way:

- You separate the **definition** of interaction points from their **usage** on Parts.
- You can define **standardized Ports** across multiple components.
- You enable **traceable, testable interaction modeling**.

A **port def** owns flow features — defining both *what flows* and *in which direction*.

Using Ports on Parts

A port is the usage of a port def. It defines *where interaction occurs* on a specific Part in your system structure. In SysML v2, once you have defined a **port def**, you can create a **port** on a **part usage**.

The port references a port def. It declares **that this Part exposes this interaction point**.
A Port does not introduce new flow features — it implements the flow features from its port def.

When building system architecture:

- Ports declare **connection points** between Parts.
- Interfaces (introduced later) define **which Ports can connect**.

By using **Ports correctly**:

- You model **where interactions happen**.
- You make Part boundaries **clear and explicit**.
- You support **plug-and-play** component design.

```
part def DroneSystem {
  port telemetry: TelemetryPort;
  port command: CommandPort;
}
```

A **Port** is an **interaction point on a Part** — based on a reusable port def.

Defining Flow Features in Ports — Direction + Item

Flow features define what flows through a Port — and in which direction. They are declared inside the port def, using an **Item definition**.

A **flow feature** is owned by a **port def**. It defines:

- The **direction** of flow:
  - **in** — the Port receives this Item.
  - **out** — the Port sends this Item.
  - **inout** — the Port both sends and receives.
- The **item def** — the type of thing that flows.

By defining flow features:

- You make interaction expectations **explicit**.
- You support **Interface Contracts**.
- You enable **validation of connections**.

The flow feature answers: **What flows? In which direction?**

It belongs to the **port def** — not to the Port usage.

```
port def TelemetryPort {
  out telemetryData: TelemetryItem;
  in command: CommandItem;
}
```

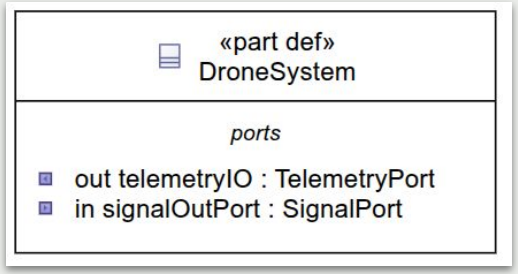
Example — Modeling a Telemetry Port + Items

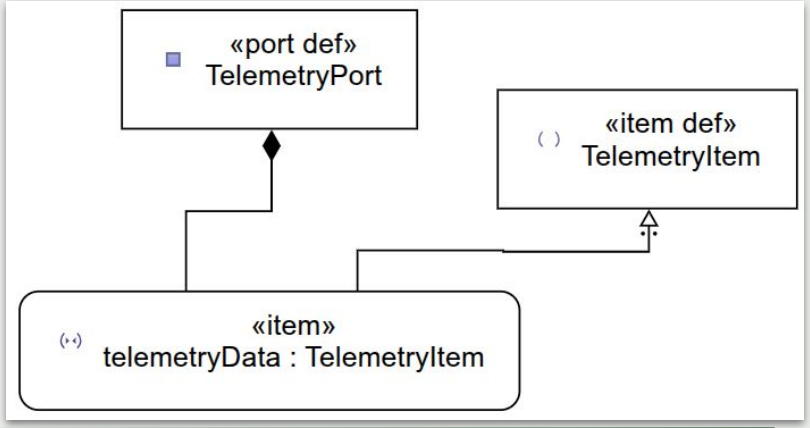
Let's look at a full example: How to model a **Telemetry Port** that sends telemetry data and receives status updates — using port def and item def. In this example:

- We define two **Items** — one for telemetry data, one for status reports.
- We define a **port def** with flow features — one out, one in.
- We place a **Port** on a Part to expose this interaction point.

Key Concepts in Action:

- item def defines *what flows*.
- **Flow feature** (inside port def) defines *direction* + *Item*.
- port places the interaction point on a Part.





Revisiting System Decomposition

Where Should Port Live?

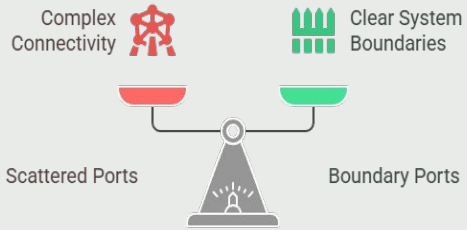
Ports represent external interaction points of a Part. Therefore, it is important to place Ports on the *correct level* of system decomposition.

In SysML v2, each port belongs to a **part usage**. It should represent an **external interaction** of that Part.

Best practice:

- Ports should live on **Parts that participate in system-level interactions**.
- Internal sub-parts that do not directly interface with external systems typically do not expose Ports at their level — they interact through the containing Part.

Correct system decomposition defines clear boundaries. Ports belong on Parts that define those boundaries.



Specialization of Parts — Modeling System Variants

Specialization allows you to model system variants cleanly. A specialized part def inherits structure (including Ports) from its parent — enabling reuse and extension.
In SysML v2, a part def can **specialize** another part def. This means the specialized Part:

- Inherits **structure** — Parts, Ports, attributes.
- Can **add** new elements.
- Can **override** (refine) certain features.

```
part def DroneSystem;

part def AdvancedDroneSystem :> DroneSystem {
  // Adds or refines components
  port secureCommand: SecureCommandPort;
};
```

When you model system variants:

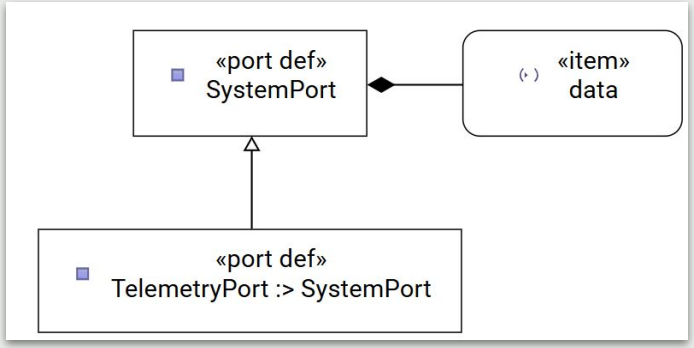
- You can define a **base architecture**.
- Then specialize it for: Product lines, Configurations, Customer options, Technology upgrades

Specialization promotes reuse and modularity — you don't repeat architecture; you extend it.

Specialization of Ports, Items, and Interfaces

Modeling Interface Families

Specialization enables modeling families of Ports, Items, and Interfaces. It supports **reusable architecture** and **incremental refinement** — without duplication.



In SysML v2:

- **port def** can specialize another port def.
- **item def** can specialize another item def.
- **interface def** can specialize another interface def.

This allows you to:

- Define a **base Port** or Interface — shared across systems.
- Create **variants** that add new flow features or refine flow Items.
- Model **Interface families** — supporting modular, replaceable components.

Specialization = reuse + variation. It gives you structured ways to model different **versions** or **levels** of Ports, Items, and Interfaces.

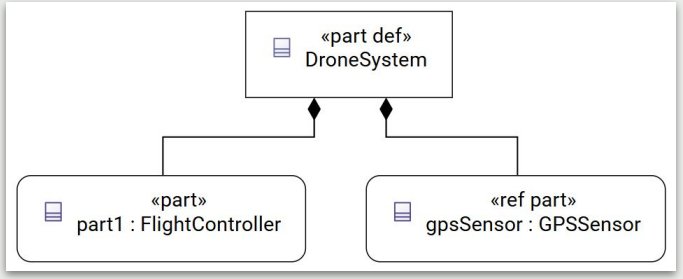
Composition vs Reference

Composition and reference model different kinds of relationships between Parts.

Use part for ownership and structure. Use ref for external or shared references.

In SysML v2:

- A part represents a **composed element**.
  It is a **contained sub-part** — owned by its parent Part.
- A ref part represents a **reference to an external element**.
  It is not owned — it points to an element that exists elsewhere.



Key Concept:

Use part when the child is **structurally part of the parent**.

Use ref part when the parent **uses or communicates with** an external element.

Attributes — Defining Part Properties

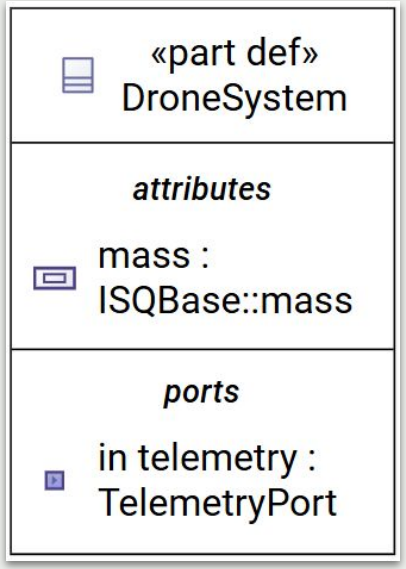
Attributes represent intrinsic properties of a Part. They are defined in the part def, and describe characteristics such as size, capacity, configuration, or state.

In SysML v2:

- An **attribute** is a **feature of a** part def.
- It defines a **property** of the Part:
  - Constant or configurable.
  - May affect behavior or interfaces.

Attributes give your model:

- **Descriptive power** — to model the characteristics of system elements.
- **Parametric modeling** — attributes can be used in constraints and analysis.
- **Interface Contracts** — Ports may depend on attribute values.



Define **attributes** in part def to model what the Part *is*. Define **Ports** to model how the Part *interacts*.

What Is an Interface in SysML v2?

An interface def defines a connection type — it specifies which Ports (or other Interfaces) can connect. It does *not* define what flows — flow definitions remain in Ports.

In SysML v2:

- An interface def defines a **connection compatibility**.
- It lists **end features** that can be connected:
  - **Ports**
  - **Interfaces**
  - **Connections**

Important:

- The interface def itself does **not define flow features**.
- **Flow features** are owned by the port def.

The interface def simply declares:

- **Which kinds of Ports can be connected**.
- Under which **Interface** the connection is valid.

interface def = **connection definition** → defines *who can connect*.

port def = **interaction point** → defines *what flows and how*.

Definition vs Usage Pattern — Interface vs Port

Interfaces and Ports follow the Definition vs Usage pattern.

The interface def defines *connection compatibility*.

The port def defines *interaction points and flow features*.

The port places the interaction point on a Part.

interface def:

- Defines a **connection type**.
- Declares **which Ports can be connected**.
- Syntax: end → refers to a port def.

port def:

- Defines an **interaction point type**.
- Declares **flow features** — direction and item def.

port:

- Places a **Port** on a part usage.
- References a port def.

Use **Interfaces** to model **connection compatibility**. Use **Ports** to model **interaction points and flows**.

Using Interface to Enable Modularity

(Plug-and-Play Architectures)

Interfaces support modular system design. By standardizing connections, Interfaces allow Parts to be replaced or upgraded — without breaking the architecture.

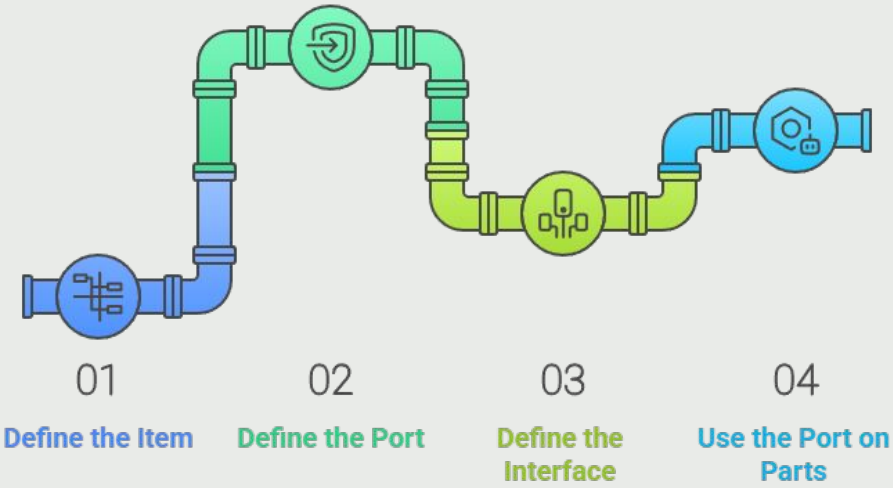
When you model **Interfaces**:

- You define **connection compatibility** once — in the interface def.
- Any **Port** based on the compatible port def can be connected.
- This enables:
  - **Replaceable components**
  - **Upgradable modules**
  - **Configurable product variants**
  - **Multi-vendor interoperability**

Modularity = design where **Interfaces remain stable** even if Parts evolve.

Example — Interface Compatibility in a Power System

Let's apply this pattern to a simple Power System.
We will model an Interface for **Power** — connecting a Battery to an ESC (Electronic Speed Controller).



Modeling with Ports + Items + Interfaces supports **modular**, **reusable**, and **traceable** system architectures.

What Is an Interface Contract?

An Interface Contract defines the expectations on an Interface.
It specifies *what must be exchanged*, *how it must be exchanged*, and *any rules or constraints* that apply. In SysML v2, the concept of an **Interface Contract** is modeled using:

- The **flow features** declared in the port def.
- The **connection compatibility** defined by the interface def.
- Optionally, additional **constraints** and **behavior rules**.

An Interface Contract answers:

- **What flows?** → defined by item def.
- **In which direction?** → defined by flow features.
- **Between which components?** → defined by interface def (Port compatibility).
- **Under what constraints?** → may be captured in:
  - **attribute values**
  - **invariants**
  - **state-based conditions**

Interface Contract = the full set of expectations governing how two Parts interact through an Interface.

Modeling Interface Contracts

Interface Contracts are modeled by combining Interface definitions, Port flow features, and Item definitions. You can also include **attributes** and **constraints** to express specific expectations.
Core structure of an Interface Contract:

a. **What flows** → item def
b. **Where it flows** → port def with flow features
c. **Who connects** → interface def specifying compatible Ports
d. **Contract expectations**:
    i. **Attributes** on Parts
    ii. **Constraints** on flows or attributes
    iii. **Behavior rules** (optional — beyond today's scope)

The combination of **Interface + Port + Item + Constraints** forms a **testable, traceable Interface Contract**.

Practice Prep

Modeling Ports, Items, and Interfaces in SysON

You are now ready to model full system interactions in SysON.
Follow this pattern: **Item → Port → Interface → Part → Connection**.

Modeling Steps:

1. **Define Items**.PowerItem
2. **Define Port**:PowerPort
3. **Define Interface**:PowerLink
4. **Add Ports to Parts**: Battery and ESC
5. **Connect Parts using Interface**: Using PowerLink

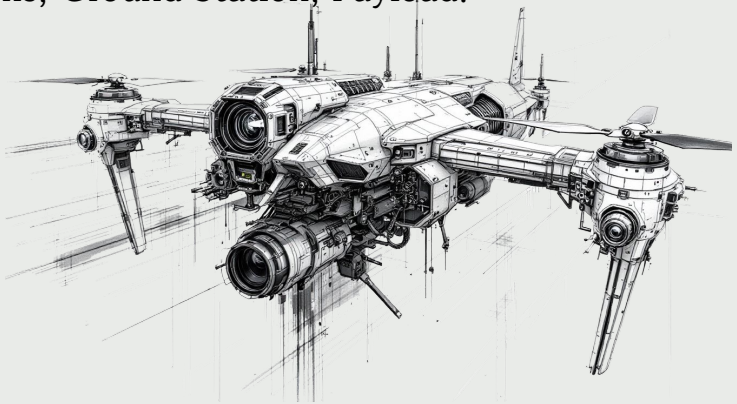
Student Project — Revisit System

Decomposition (Add Correct Ports)

Now revisit your Drone System decomposition. Add **Ports** at the correct architectural levels — to model how your system interacts internally and externally.

Project Guidance:

- **Review your Part hierarchy** (from Week 5).
- For each **Part** that connects to: Other Parts, External systems, Ground Station, Payload.
- Decide:
  - Does this Part need a **Port**?
  - What **Item(s)** flow through that Port?
  - Which **Interface** governs the connection?
- Place port only on:
  - **System boundary Parts**.
  - **Key sub-system Parts**.
  - Parts that interact via defined Interfaces.



Ports belong on Parts that define interaction boundaries — not on deep internal components.

Define Key Interfaces and Items

Now define the key Interfaces and Items for your Drone System. Focus on major interaction flows — starting with the most important ones.

Suggested Interfaces to Model:

TelemetryLink — connects Drone to Ground Station.

- TelemetryItem → data out.
- StatusItem → data in.

PowerSupply — connects Battery to ESC or other loads.

- PowerItem → out/in.

CommandChannel — connects Ground Station to Drone.

- CommandItem → in.

SensorDataLink — connects Sensors to Flight Controller.

- SensorDataItem → out.

Start with **major interaction flows** — you can refine and expand Interfaces later.

Summary of Week 08

This week you learned how to model system interactions explicitly.
You now have Ports, Flows, and Interfaces — enabling traceable, testable, and modular system architecture.

Key Takeaways:

- **Ports** (port def, port) define *interaction points* and *flow features*.
- **Items** (item def, item) define *what flows* through Ports.
- **Interfaces** (interface def) define *connection compatibility — who can connect*.
- **System Decomposition**: Ports belong on Parts that define interaction boundaries.
- **Composition vs Reference**: Use part for ownership, ref for external references.
- **Attributes**: Define Part properties — often referenced in Interface Contracts.
- **Interface Contracts**: Combine Interface + Port + Item + Constraints.

"Your model now shows not just *what the system is made of* — but *how it communicates and interacts.*"

QUESTION!