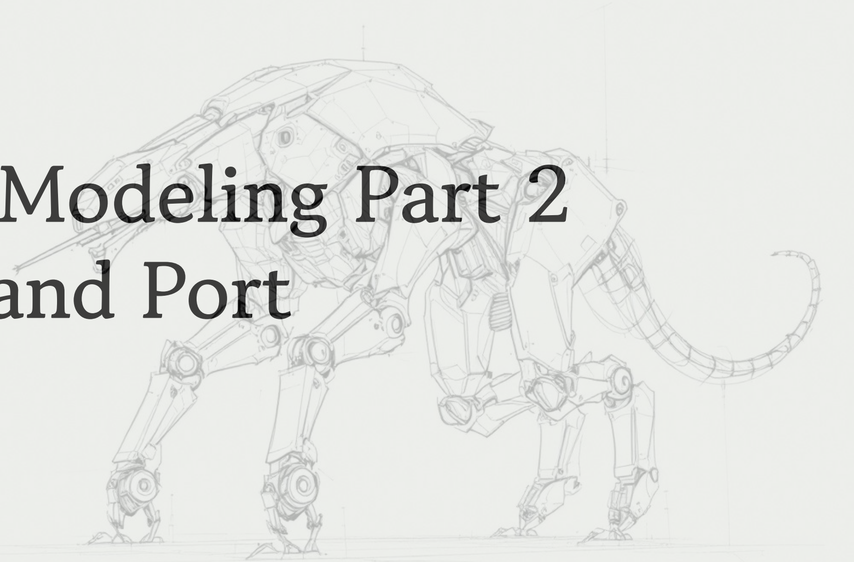
Structure Modeling Part 2
Interface and Port



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

Welcome to **Week 8** — we are continuing to strengthen the Logical Layer of your system model.
Today, you will learn how to model:

- **How system parts interact**
- **What flows between them**
- And **how to ensure modularity and compatibility**

You will also revisit **system-level decomposition** — to ensure Ports are placed at the correct architectural level.
And you will see how to use **Specialization** to model **interface families**, **port variants**, and **system configurations**.
By the end of this week, your Drone System model will not only show what it is made of — but also **how it communicates and exchanges information internally and externally**.

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

Why Model Interactions and Flows?
Why do we model **Ports**, **Items**, and **Interfaces**?
Because without them — system diagrams become ambiguous:

- Who talks to whom?
- What data is exchanged?
- What assumptions are being made?

When we model:

- **Ports** define *where* interaction happens.
- **Items** define *what flows* through those Ports.
- **Interfaces** define *which Ports are compatible* for connection.

This gives us:

- **Clear communication** between Parts.
- **Reusable architecture patterns** — one Interface, many implementations.
- **Traceability** — we can analyze how Parts interact and verify those interactions.

Without explicit Ports, Flows, and Interfaces — system integration is difficult, fragile, and error-prone.

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.

Why Model Interactions and Flows?

In SysML v2, the **Port** is the primary modeling element that says:

> *"This is where my Part can interact with the rest of the system."*

You define the type of interaction point using a port def.

Inside that definition, you specify **flow features**:

- You describe **what will flow** — using item def.
- You describe **the direction** — is this Port receiving, sending, or both?

When you create a port on a part usage, you are declaring:

> "This Part has this kind of interaction point — and it is expected to exchange this type of Item."

This brings **clarity and structure** to system interactions — and it enables **modular architecture**, as we will soon see.

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).

Why Model Interactions and Flows?

Now that you know what a Port is, let's look at **what actually flows through the Port**.

That's the role of **item def** and **item**.

An item def defines the **type of flow** — it could be data, power, material, or any other physical or informational content.

Then, inside the **Port definition**, you declare **flow features**:

- You specify the **direction** — is this Port sending, receiving, or both?
- You specify the **Item** type that flows through it.

For example:

A TelemetryPort might have an **out telemetryData: TelemetryItem** flow feature.

Now the system model knows:

- Where interaction happens (**Port**)
- What flows (**Item**)
- In which direction.

This gives us a complete picture of system interaction — and prepares us to model **Interface compatibility** in the next segments.

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*.

Modeling Ports, Items, and Flows
A **port def** is where you define a **reusable interaction point**.
This is important:

- You are not just declaring Ports ad hoc on each Part.
- You are defining a **Port type** that can be used consistently across your architecture.

The port def owns **flow features**:

- Each flow feature has a **direction** — in, out, or inout.
- Each flow feature references an **item def** — defining what flows.

For example, a TelemetryPort might:

- **Send telemetry data** (out telemetryData).
- **Receive status reports** (in statusReport).

This model is:

- **Modular** — the same Port definition can be used on many Parts.
- **Traceable** — the flows are explicitly modeled.
- **Testable** — Interface Contracts can verify the flow expectations.

You will use this pattern in your Drone System model — to ensure that interactions are clear and reusable.

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.

Modeling Ports, Items, and Flows
Now that you've defined your port def — how do you use it?
You create a **port** on a **Part**.
The port:

- Is based on a port def.
- Declares **where interaction happens** on the Part.
- Implements the **flow features** already defined in the port def.

You do not redefine the flows here — you are simply saying: "This Part has this Port."
Now the model knows that **DroneSystem** exposes a **TelemetryPort**. This makes system interactions **explicit** — and ready to connect to other Ports via Interfaces.

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;
}
```


Modeling Ports, Items, and Flows

Flow features are how you tell the system: “Here is what flows through this Port.”

They are defined **inside** the port def — not on the port usage.

Each flow feature:

- Declares a **direction** — in, out, inout.
- References an **item def** — the type of Item that will flow.

We see that:

- The Port **sends telemetry data** (out).
- The Port **receives commands** (in).

Now the model knows:

- What flows through this Port.
- How it should be connected.
- What must be tested in an Interface Contract.

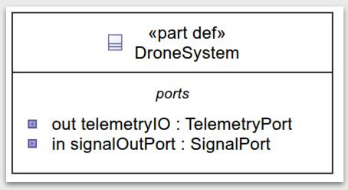
Example — Modeling a Telemetry Port + Items

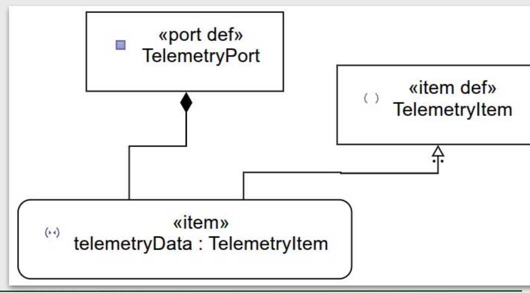
Let's look at a full example: How to model a **Telemetry Port** that sends telemetry data and receives status updates — using port def and item def. In this example:

- 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.





Modeling Ports, Items, and Flows

Here's a **complete example** of how to model a Telemetry Port — exactly as you'll do in your Drone System project.

First, we define **what flows** — using item def:

- A TelemetryItem — represents telemetry data.
- A StatusSignal — represents system status feedback.

Next, we define a **Port type** — TelemetryPort:

- It **sends telemetry data** (out).
- It **receives status reports** (in).

Finally, we add a **Port** to the DroneSystem: port telemetry: TelemetryPort;

Now the model knows:

- **What interaction happens** at this Port.
- **What Items flow**.
- **In which direction**.

This makes system interaction **clear, reusable, and testable**.

Revisiting System Decomposition

Where Should Port Live?

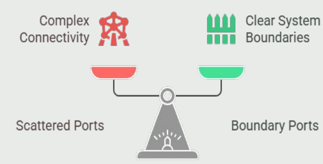
Ports represent external interaction points of a Part. Therefore, it is important to place Ports on the *correct level* of system decomposition.

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.



Revisiting System-Level Decomposition + Specialization

Before we introduce **Interfaces**, it is important to revisit **system decomposition**.

Why?

Because **Ports** represent **external interaction points**:

- They are not "internal implementation details."
- They should model where a Part connects to the rest of the system — or to external systems.

This means:

- You should place Ports on **Parts that define system or sub-system boundaries**.
- Internal components should interact through their container Part — unless they truly expose an external Port.

For example:

- The **DroneSystem** might have Ports for:
  - Telemetry
  - Power
  - Command interface
- But internal components — like the MotorController — may not expose Ports directly to the outside.

Correct decomposition:

- Keeps your architecture **clear**.
- Makes **Interfaces reusable**.
- Avoids **hidden or tangled interaction paths**.

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.

Revisiting System-Level Decomposition + Specialization

Specialization is a key pattern in **system modeling** — and now that you are adding **Ports and Interfaces**, it becomes even more useful.
In SysML v2, a part def can **specialize** another part def:

- The specialized Part **inherits** all structure from its parent.
- It can add new **Parts**, **Ports**, or other features.
- It can refine existing Ports or Interfaces.

This allows you to:

- Model **system variants** without duplicating architecture.
- Keep **common structure consistent**.
- Model **different configurations** of your Drone System.

For example:

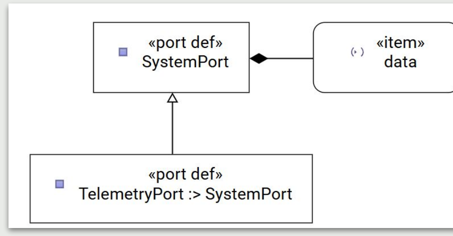
- You can define a DroneSystem with standard Ports.
- Then create AdvancedDroneSystem that adds a **secure command Port** — while reusing the existing architecture.

This promotes **clean, reusable models** — and supports **product line engineering**.

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.

Revisiting System-Level Decomposition + Specialization

Specialization is not just for Parts — it works everywhere in your architecture.

You can specialize:

- **item def** — to model different levels or types of flow content.
- **port def** — to model Port variants or families.
- **interface def** — to model Interface families.

This is very useful when building:

- **Product lines** — where some variants have enhanced Ports.
- **Component families** — where some Interfaces carry more data or stricter protocols.

For example:

- You can define a TelemetryPort that sends basic telemetry.
- Then define a HighRateTelemetryPort — specialized — that sends enhanced telemetry.

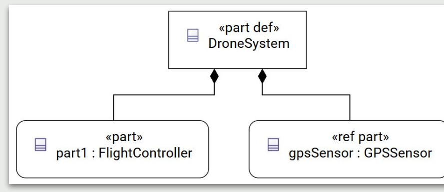
Now you can model **compatibility** and **variation** — while keeping your architecture **clean** and **reusable**.

You will use this pattern in your Drone System to model **optional Interfaces**, **advanced variants**, or **upgradable components**.

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.

Revisiting System-Level Decomposition + Specialization
As you refine your system model, it's important to model **correct relationships** between Parts.
part means **composition**:

- The child is part of the parent.
- The parent controls the lifecycle.
- If the parent is deleted, so is the child.

ref part means **reference**:

- The parent **uses** the referenced Part.
- The reference is **external** — not owned.
- The referenced Part may be shared by other components.

You will see this often when modeling **Interfaces**:

- Many Ports connect to **referenced Parts** — not composed sub-parts.

For example:

- A DroneSystem may **own** its FlightController (part).
- But it may **reference** a shared GPS sensor (ref part) — which is used by multiple sub-systems.

Modeling this correctly makes **Ports and Interfaces** more accurate — and supports **modularity and reuse**.

Attributes — Defining Part Properties

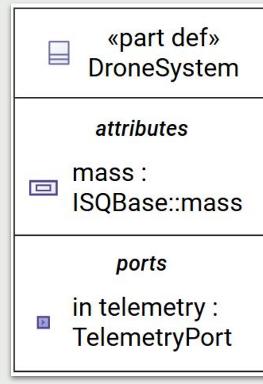
Attributes represent intrinsic properties of a Part. They are defined in the part def, and describe characteristics such as size, capacity, configuration, or state.

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*.

Revisiting System-Level Decomposition + Specialization

Attributes are how you model the **properties** of a Part. They belong to the **part def** — not to the part usage.

Attributes describe:

- **Size**
- **Capacity**
- **Configuration**
- **State variables**
- Any property that defines what the Part *is* or *has*.

Later, you can use these **attributes** in:

- **Behavior models**
- **Constraints**
- **Interface Contracts** (e.g., Ports that only operate within certain voltage ranges)

This keeps your model **rich and analyzable** — not just structural.

In your Drone System project, you will add attributes to:

- **Battery** (capacity, voltage)
- **Motors** (thrust rating)
- **Flight Controller** (firmware version)

Attributes complement **Ports**:

- Attributes say **what the Part is.**
- Ports say **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.*

Interface Definitions and Connection Compatibility

Now that your model has **correct Parts, Ports, Items, and Flows** — we can introduce **Interfaces**.

In SysML v2:

- An **interface def** defines a **connection type**.
- It does **not define flow features**.
- It defines **which Ports (or Interfaces)** can connect under this Interface.

Think of it this way:

- The port def says: “Here’s what flows through this Port.”
- The interface def says: “Here’s which Ports can be connected under this Interface.”

Now the model knows: Only Ports of type TelemetryPort can be connected under TelemetryLink.

This gives you:

- **Modular architecture** — Ports are reusable across Parts.
- **Interface Contracts** — you can define compatibility rules.
- **Traceability** — Interfaces are first-class model elements.

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**.

Interface Definitions and Connection Compatibility

Interfaces and Ports follow the **Definition vs Usage** pattern — just like other parts of SysML v2.

You first define an **interface def**:

- It declares which Ports are compatible for this connection.
- It uses the **end** keyword to reference the port def.

Separately, you define a **port def**:

- It defines what the Port is — an interaction point.
- It defines **flow features**: what flows, and in which direction.

Then, you place a **port** on a part usage:

- The Port uses a port def.
- It declares **where** the interaction happens.

Finally, you connect Ports using an **Interface**:

- The Interface enforces **compatibility**.
- It makes connections **explicit and reusable**.

This is the correct SysML v2 modeling pattern — and you will use it throughout your Drone System project.

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.

Interface Definitions and Connection Compatibility

One of the biggest benefits of modeling **Interfaces** is that they enable **modular system design**. You can define an Interface once — using interface def. Then any Port — if it matches the port def — can connect under that Interface.

This means:

- You can replace components **without changing the rest of the architecture**.
- You can model **variants** or **product upgrades** easily.
- You can define **supplier/vendor Interfaces** — to allow interoperability.

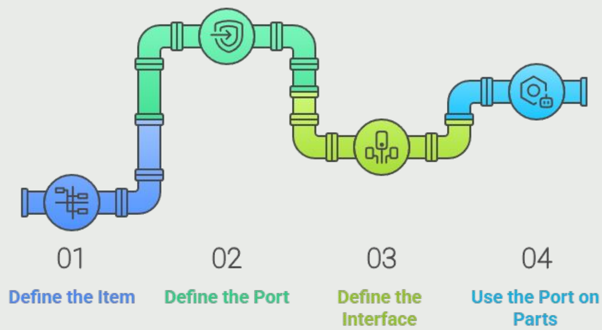
For example:

- A PowerLink Interface connects PowerPorts.
- Whether you use Battery V1 or Battery V2 — as long as they provide a PowerPort, they can connect.

This supports **Plug-and-Play Architectures** — a key goal of modern system design.

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.

Interface Definitions and Connection Compatibility

Here's a **neutral example** — one you can apply in many systems.

We want a **Battery** to supply **Power** to an **ESC** (Electronic Speed Controller).

How do we model this?

Step 1 — Define the **Item**:

- PowerItem — representing voltage, current, or power units.

Step 2 — Define the **Port**:

- A PowerPort — with out powerOut and in powerIn.

Step 3 — Define the **Interface**:

- PowerLink — connects two PowerPorts.

Step 4 — Place **Ports** on Parts:

- Battery → output: PowerPort
- ESC → input: PowerPort

Now the connection is:

- **Explicit** — what flows is clearly defined.
- **Modular** — you can swap the Battery or ESC.
- **Traceable** — the flow can be tested and validated.

This is the same pattern you will apply in your own projects — for any type of interaction.

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
An **Interface Contract** defines: *What must happen when two Parts are connected through this Interface.*
The **structure** of the Contract comes from:

- The port def — which declares **what flows** and **in which direction**.
- The interface def — which declares **connection compatibility**.

The **semantics** of the Contract — *what must be true* — comes from:

- **Attributes** on the connected Parts.
- **Constraints** — such as voltage levels, data rates, timing.
- **State-dependent behavior** — in more advanced models.

For example: In a **PowerLink**, we may require:

    - The **Battery** provides powerOut at **12 V ± 5%**.
    - The **ESC** accepts powerIn in that range — and limits current draw.

Modeling these expectations is what makes Interfaces **testable** — you can write test cases against the Contract.
As your Drone System grows, you will define **Interface Contracts** for:

- Telemetry
- Command
- Power
- Sensor data
- Mechanical or payload interfaces

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**.

Modeling Interface Contracts

How do we model a **full Interface Contract**?

You combine these elements:

1 **item def** — defines *what flows*.
2 **port def** — defines *where it flows* and *in which direction*.
3 **interface def** — defines *who connects*.
4 Optionally — **Attributes** and **Constraints** define *what must be true* about the connection.

For example — in this PowerLink:

- PowerItem defines what flows.
- PowerPort defines the flow directions.
- PowerLink defines who can connect.
- Battery and ESC define **attributes** — voltage, current — which form part of the Contract.

When you later write **test cases**, you test whether these expectations are met.

This is how Interfaces move from **diagram boxes** to **real, testable architecture**.

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

Practice in SysON (Ports + Interfaces + Items)

Now you are ready to build full system interactions in **SysON**.

Here is the correct modeling flow:

1 Start with **Items** — what will flow.

2 Define **Ports** — where interaction happens.

3 Define **Interfaces** — which Ports can connect.

4 Place **Ports** on **Parts**.

5 Connect **Parts** using Interfaces.

You will follow this same flow in every architecture model — whether for:

- Data connections
- Power
- Control signals
- Mechanical interfaces

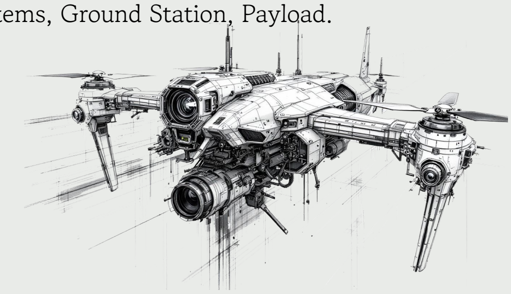
Student Project — Revisit System

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.

Project Integration — Drone System

Now that you know how to model **Ports** correctly — it's time to revisit your **Drone System architecture**.

Ask yourself:

- Where should the **Ports live**?
- Which Parts interact with **external systems** or **other sub-systems**?

Remember:

- **Ports belong on Parts that define interaction boundaries.**
- Not every sub-part should have Ports.
- Internal components often interact **through** their containing Part.

Review your **Week 5 decomposition**:

- Add Ports to the correct Parts.
- Start thinking about **Interfaces** you will define to govern those connections.

This will give you a **clean, modular, traceable system architecture** — ready for Interface modeling.

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.

Project Integration — Drone System

Now it's time to define **real Interfaces** in your Drone System. Start with the **most important interactions** — don't try to model everything at once.

Good starting Interfaces:

- **TelemetryLink** — Drone ↔ Ground Station.
- **PowerSupply** — Battery → ESC or loads.
- **CommandChannel** — Ground Station → Drone.
- **SensorDataLink** — Sensors → Flight Controller.

Follow this modeling flow:

1️⃣ Define **Items**.
2️⃣ Define **Ports**.
3️⃣ Define **Interfaces**.
4️⃣ Place **Ports** on Parts.
5️⃣ Connect Parts using **Interfaces**.

Keep it simple at first — model what you understand. You can **refine** and **expand** later.

This will give you a **clean, modular architecture** — ready for testing and future extensions.

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.*"

This week you've reached an important milestone:
Your model now **communicates**.
You've added:

- **Ports** — where interaction happens.
- **Items** — what flows.
- **Interfaces** — who connects.
- **Attributes** — intrinsic Part properties.
- **Composition and References** — correct system architecture.

You've also learned how to model **Interface Contracts** — making your system interactions **testable** and **traceable**.
Now your model shows not just **what the system is made of** — but also **how it interacts**, both internally and with the external world.
This is the foundation for:

- Modeling **behavior across Interfaces**.
- Modeling **Interface Contracts** in more detail.
- Supporting **integration testing** and **validation**.

You will continue building on this in the next weeks — your architecture is now ready for deeper modeling.

QUESTION!