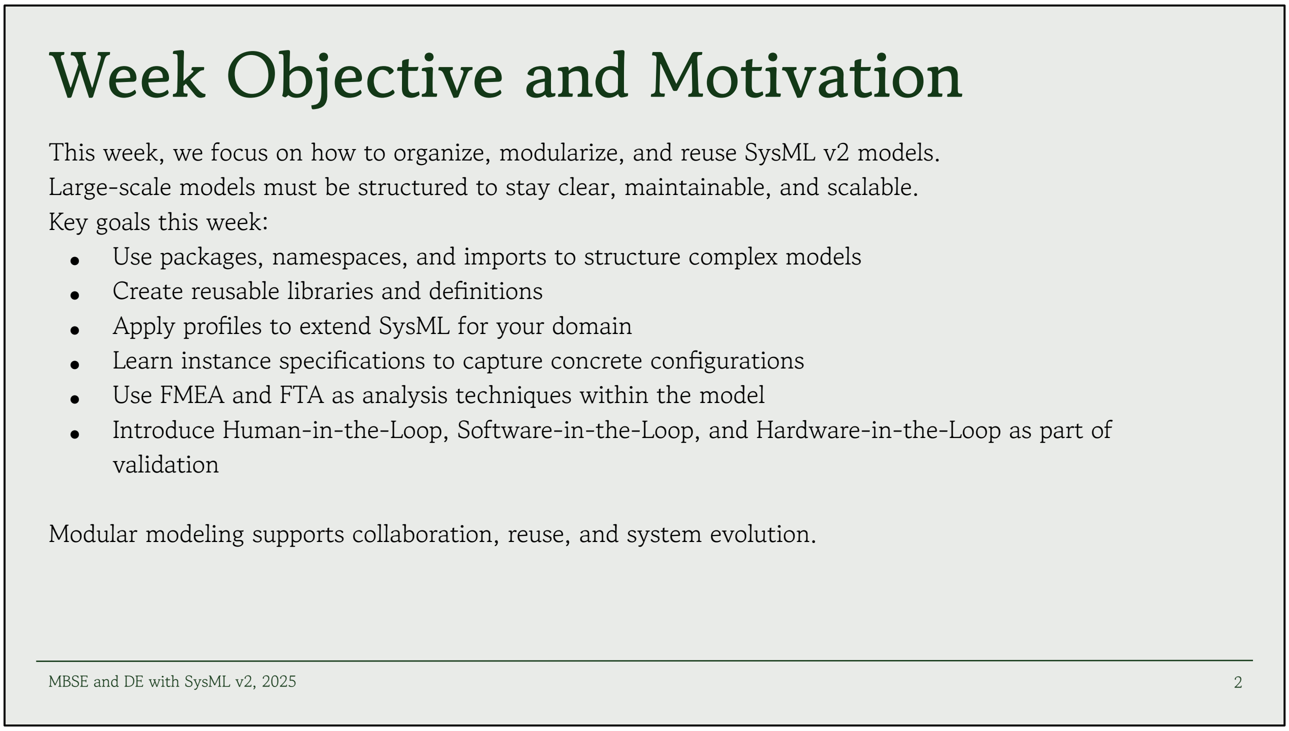
Advanced Topics and Best Practices

Week 14



Week Objective and Motivation

This week, we focus on how to organize, modularize, and reuse SysML v2 models.
Large-scale models must be structured to stay clear, maintainable, and scalable.
Key goals this week:

- Use packages, namespaces, and imports to structure complex models
- Create reusable libraries and definitions
- Apply profiles to extend SysML for your domain
- Learn instance specifications to capture concrete configurations
- Use FMEA and FTA as analysis techniques within the model
- Introduce Human-in-the-Loop, Software-in-the-Loop, and Hardware-in-the-Loop as part of validation

Modular modeling supports collaboration, reuse, and system evolution.

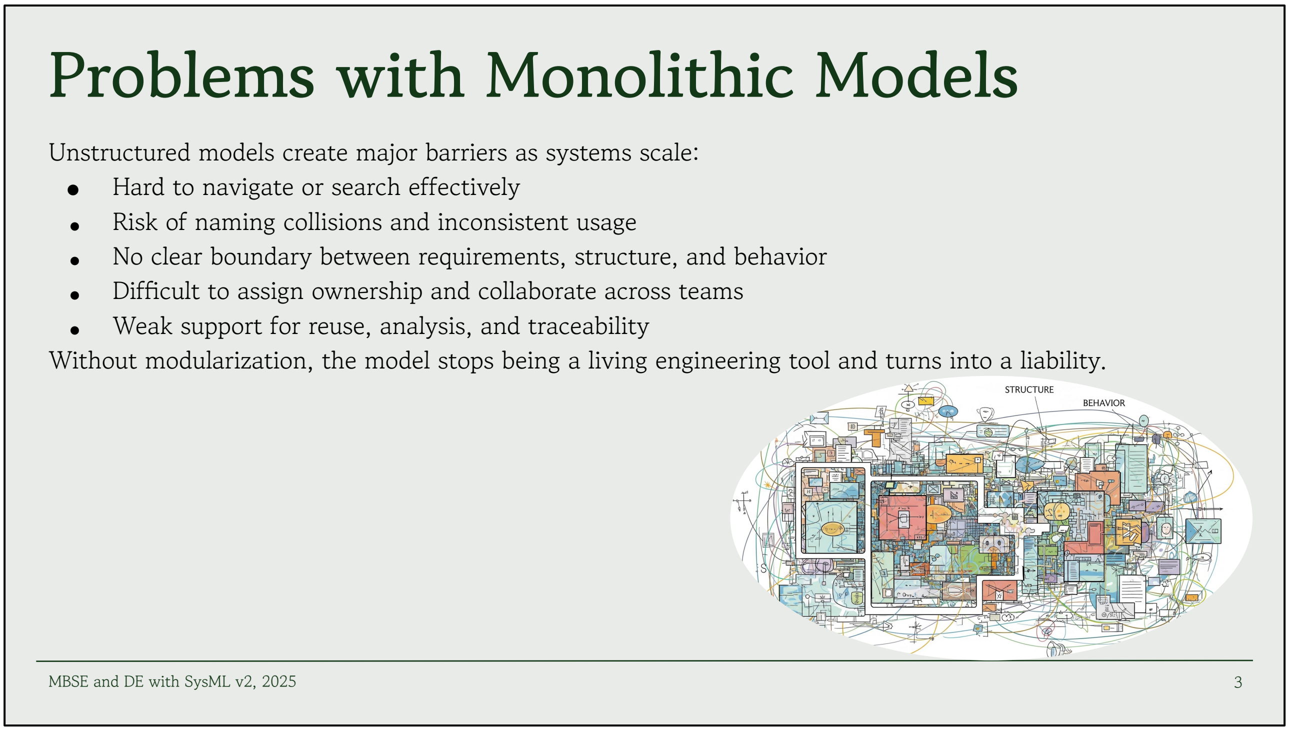
This week we shift from building new content to organizing and structuring what we already have. The challenge in real projects is not only defining the system but keeping the model usable as it grows.
In practice, system models expand fast: multiple stakeholders, dozens of diagrams, and hundreds of elements. If everything stays in one flat structure, it becomes nearly impossible to reuse, verify, or extend.
That is why SysML v2 provides strong organizational mechanisms:

- **Packages and namespaces** let us divide a large model into modular parts.
- **Imports and libraries** allow reuse of definitions across teams and tools.
- **Profiles** enable tailoring the language for specific domains such as aerospace, automotive, or maritime.
- **Instance specifications** help capture specific configurations or scenarios from the general model.

We will also connect model organization to **analysis and validation**:

- Using **FMEA (Failure Mode and Effects Analysis)** and **FTA (Fault Tree Analysis)** to evaluate reliability and risks directly in the model.
- Introducing **Human-in-the-Loop, Software-in-the-Loop, and Hardware-in-the-Loop** as progressive validation strategies that bridge the model with real operators, software, and hardware.

By the end of this week, your drone system will be modular, reusable, and structured for professional MBSE practice, with the foundation to scale into system-of-systems modeling and advanced validation workflows.

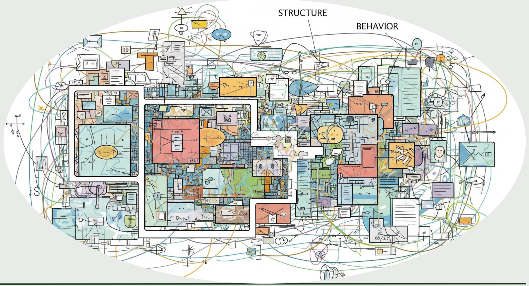


Problems with Monolithic Models

Unstructured models create major barriers as systems scale:

- Hard to navigate or search effectively
- Risk of naming collisions and inconsistent usage
- No clear boundary between requirements, structure, and behavior
- Difficult to assign ownership and collaborate across teams
- Weak support for reuse, analysis, and traceability

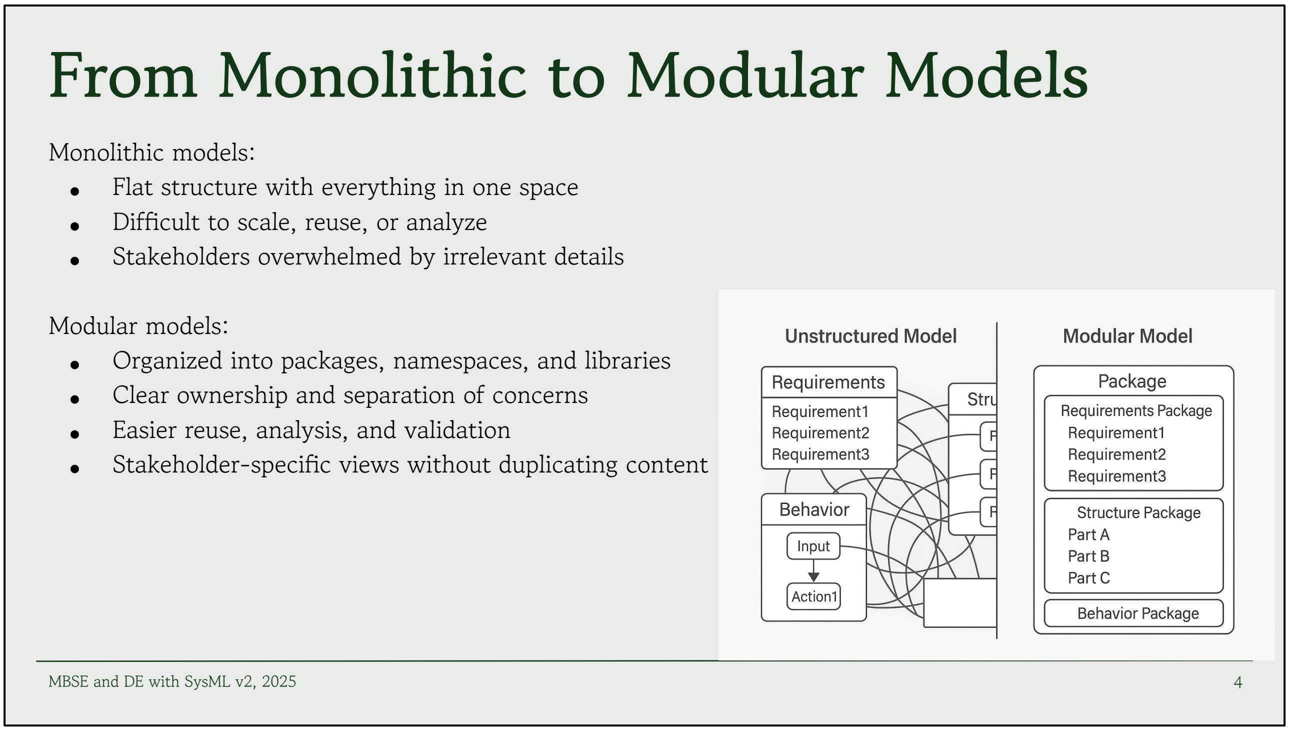
Without modularization, the model stops being a living engineering tool and turns into a liability.



When models are left as one big block, they quickly become unmanageable. At the start, it feels simple: everything in one diagram, one namespace, one repository. But once the system grows, the cracks show.

- Navigation becomes painful. You spend more time searching than modeling.
- Naming clashes happen because everyone defines parts, actions, or attributes in isolation.
- Requirements, structure, and behavior blur together, making it hard to separate concerns.
- Teams cannot collaborate well, since ownership of elements is unclear.
- Reuse and analysis suffer: you cannot easily apply libraries, run FMEA or FTA, or set up configurations for Human-in-the-Loop or Software-in-the-Loop testing.

In MBSE, the model should be an enabler, not a burden. That is why organization is more than just tidiness — it is what makes large-scale, collaborative, and safety-critical modeling possible.



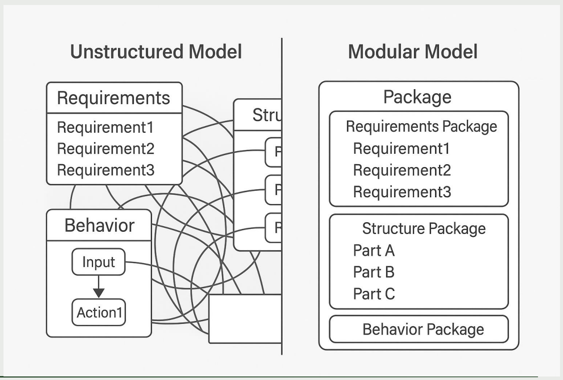
From Monolithic to Modular Models

Monolithic models:

- Flat structure with everything in one space
- Difficult to scale, reuse, or analyze
- Stakeholders overwhelmed by irrelevant details

Modular models:

- Organized into packages, namespaces, and libraries
- Clear ownership and separation of concerns
- Easier reuse, analysis, and validation
- Stakeholder-specific views without duplicating content

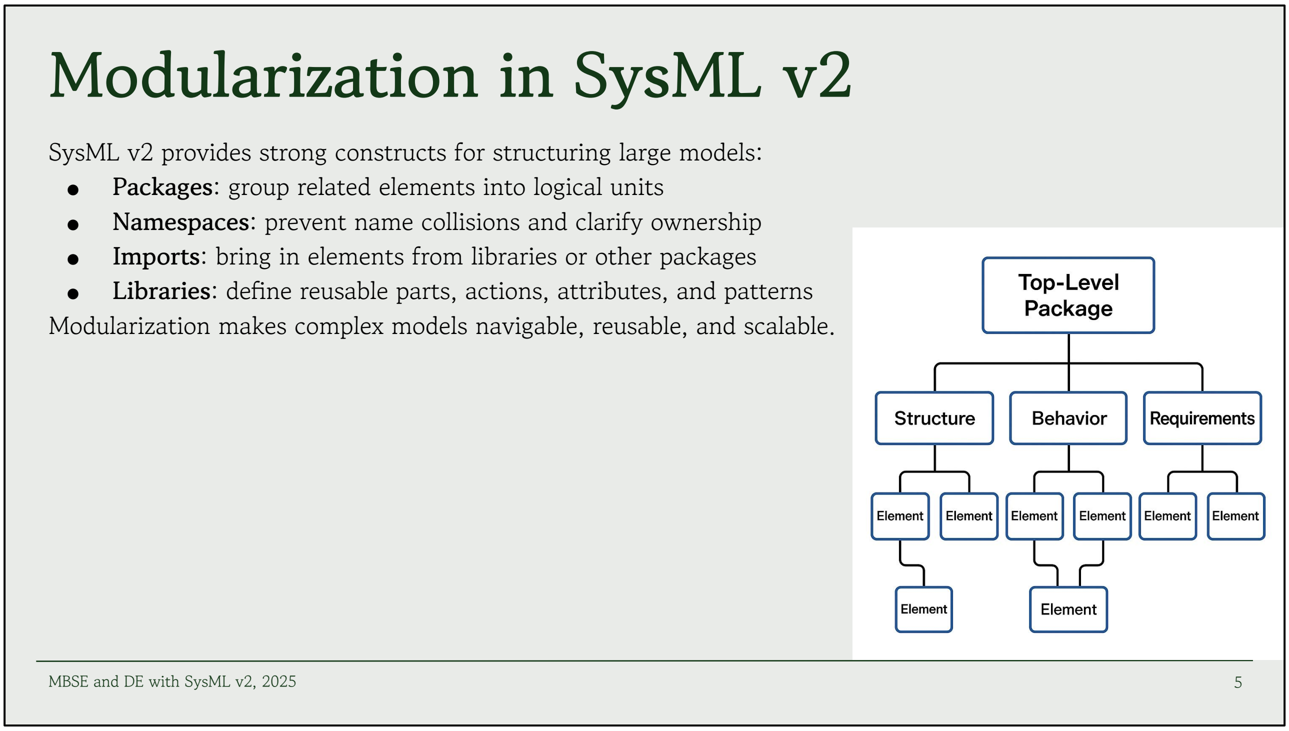


The cure for monolithic models is modularization. Instead of placing all requirements, behaviors, and structures into a single flat model, we break the system into organized units.

- **Monolithic models** are simple at first but collapse under complexity. They confuse stakeholders, block collaboration, and make advanced analysis nearly impossible.
- **Modular models** use SysML v2 concepts like packages, namespaces, and imports to structure the model. Each subsystem or concern has its own space, but everything remains connected in one coherent system.

This approach allows:

- Clear separation of requirements, structure, and behavior.
- Libraries of reusable definitions that scale across projects.
- Support for analysis techniques such as FMEA and FTA, which require clean traceability.
- Easier integration with Human-in-the-Loop, Software-in-the-Loop, and Hardware-in-the-Loop setups.

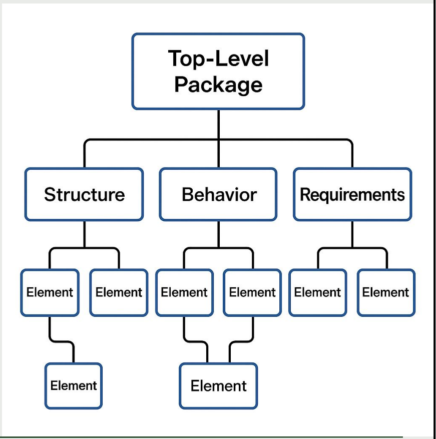


Modularization in SysML v2

SysML v2 provides strong constructs for structuring large models:

- **Packages**: group related elements into logical units
- **Namespaces**: prevent name collisions and clarify ownership
- **Imports**: bring in elements from libraries or other packages
- **Libraries**: define reusable parts, actions, attributes, and patterns

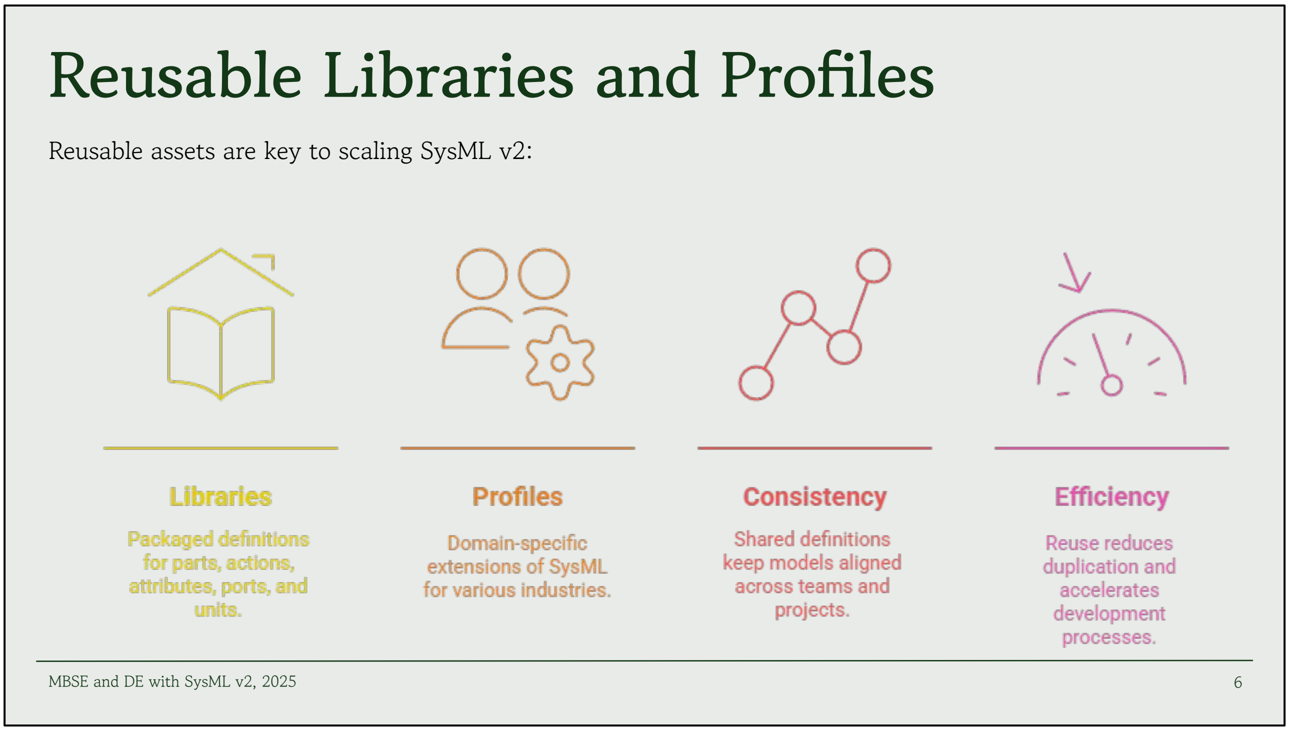
Modularization makes complex models navigable, reusable, and scalable.



Modularization is one of the most powerful upgrades in SysML v2. It goes beyond tidiness and directly enables scalability, collaboration, and reuse.

- **Packages** create natural boundaries in the model. Each subsystem or concern can be placed in its own package.
- **Namespaces** ensure unique, clear naming. They prevent collisions and clarify ownership when multiple teams contribute.
- **Imports** allow you to reuse definitions. You don't need to copy and paste parts or attributes — you can reference them cleanly across the model.
- **Libraries** provide standard, reusable building blocks. They can capture common definitions such as SI units, aerospace components, or company-specific patterns.

This modular approach makes the drone system easier to maintain, extend, and validate. It also sets the stage for domain-specific profiles, FMEA/FTA integration, and simulation setups like Human-in-the-Loop or Hardware-in-the-Loop.



Reusable Libraries and Profiles

Reusable assets are key to scaling SysML v2:

Libraries

Packaged definitions for parts, actions, attributes, ports, and units.

Profiles

Domain-specific extensions of SysML for various industries.

Consistency

Shared definitions keep models aligned across teams and projects.

Efficiency

Reuse reduces duplication and accelerates development processes.

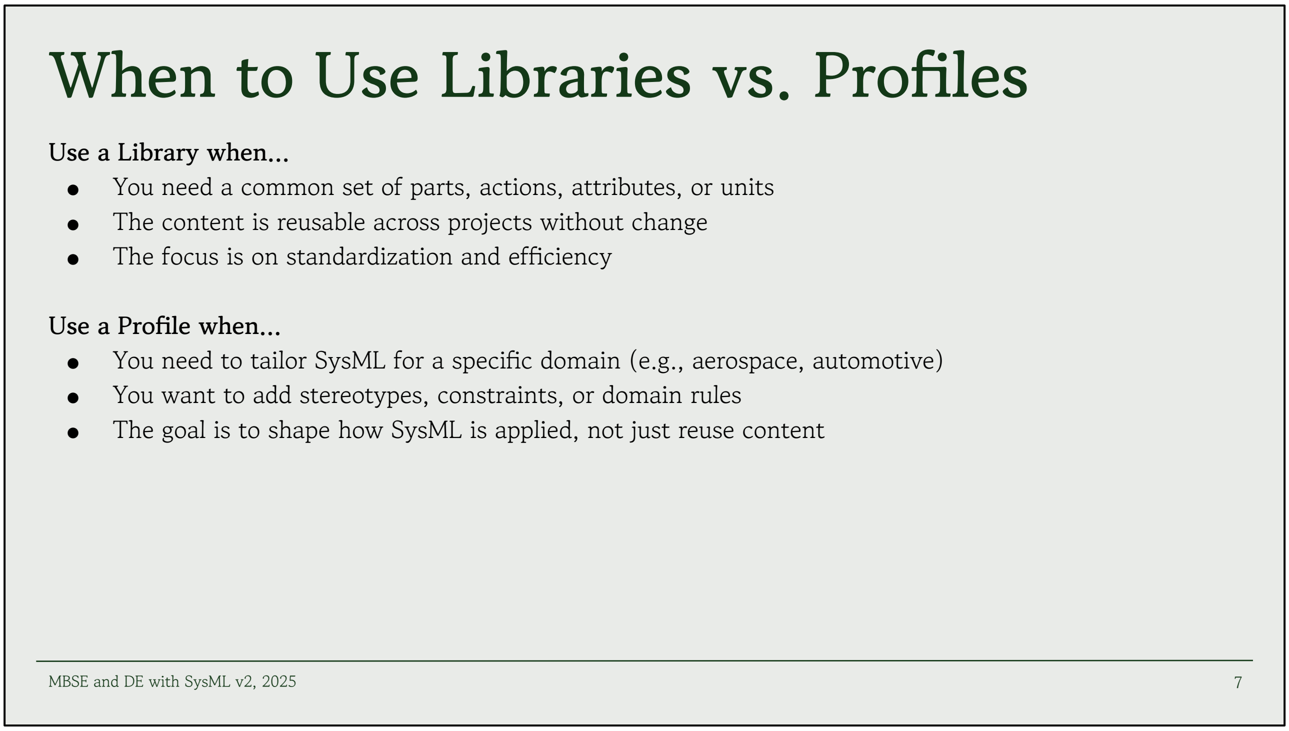
MBSE and DE with SysML v2, 2025 6

Once models are modularized, the next step is reuse. Large organizations cannot afford to reinvent the same definitions for every project.

- **Libraries** provide collections of reusable definitions. For example, you might have a standard library for physical units, common UAV subsystems, or control system patterns. These libraries can be imported into any model, ensuring consistency across projects.
- **Profiles** let you extend SysML v2 for your specific domain. Instead of bending the language to fit, you define stereotypes and constraints that match your field — aerospace, automotive, maritime, or medical systems. A profile makes the model speak your domain's language while still following the SysML standard.

Together, libraries and profiles form the backbone of scalable MBSE practice. They allow different teams to work in parallel while staying consistent. They also make models easier to share with suppliers, customers, or regulators, since everyone sees the same definitions and domain conventions.

For our drone system, we can build a small reusable library of subsystems and apply a lightweight profile for UAV operations. This will demonstrate how professional MBSE projects combine reuse with domain specialization.



When to Use Libraries vs. Profiles

Use a Library when...

- You need a common set of parts, actions, attributes, or units
- The content is reusable across projects without change
- The focus is on standardization and efficiency

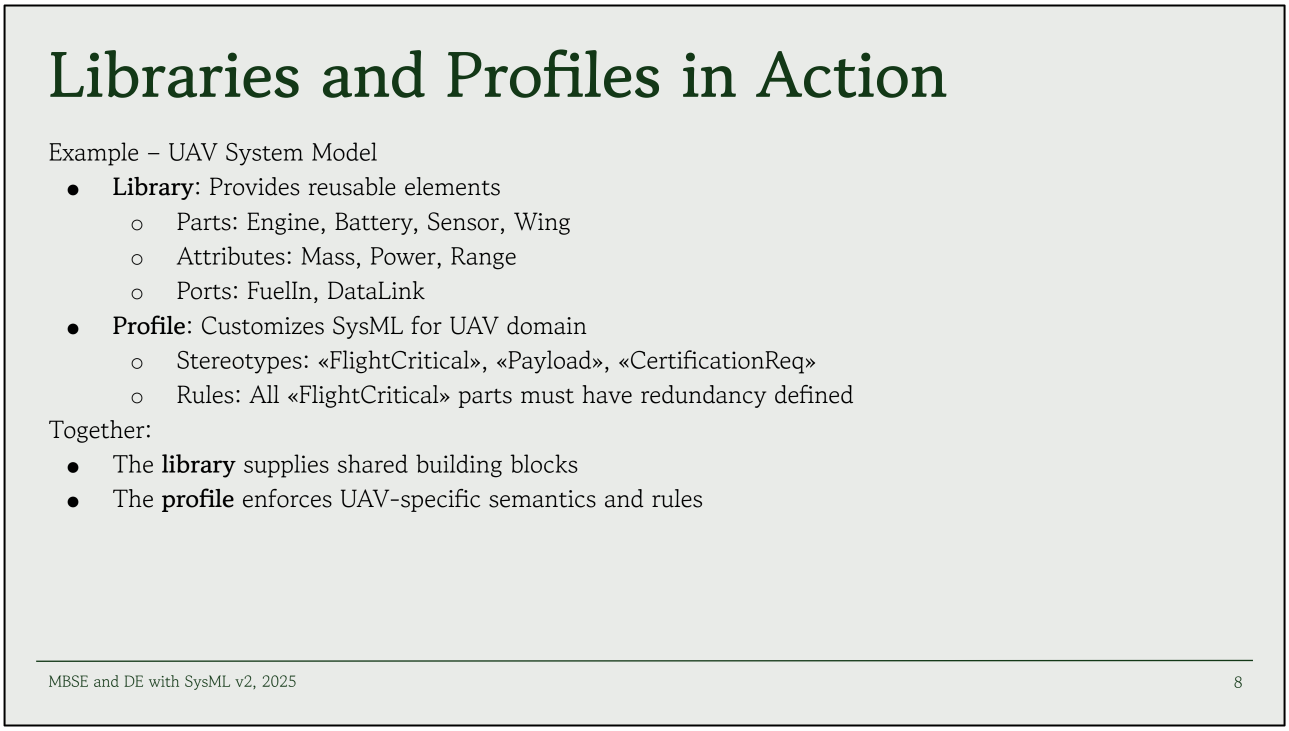
Use a Profile when...

- You need to tailor SysML for a specific domain (e.g., aerospace, automotive)
- You want to add stereotypes, constraints, or domain rules
- The goal is to shape how SysML is applied, not just reuse content

For the question on whether to put something in a library or a profile. The answer depends on the purpose.

- **Libraries** are for reusable content. If you have standard UAV subsystems, SI units, or common analysis patterns, package them in a library. Any project can import that library and use the definitions as-is.
- **Profiles** are for specialization. If you need SysML to reflect the language and rules of your domain, a profile is the right choice. For example, an aerospace profile might add stereotypes for "Flight Control Software" or "Certification Requirement." A maritime profile might add "Hull Structure" or "Classification Rule."

Think of libraries as **what you reuse**, and profiles as **how you adapt SysML to your world**.
Both can be used together: a profile might rely on libraries to enforce consistency, and a library can be applied under a profile to provide domain-specific extensions.
In our drone project, we can use libraries for subsystems like propulsion and sensors, while applying a UAV profile to ensure the model speaks in aviation terms.



Libraries and Profiles in Action

Example – UAV System Model

- **Library**: Provides reusable elements
  - Parts: Engine, Battery, Sensor, Wing
  - Attributes: Mass, Power, Range
  - Ports: FuelIn, DataLink
- **Profile**: Customizes SysML for UAV domain
  - Stereotypes: «FlightCritical», «Payload», «CertificationReq»
  - Rules: All «FlightCritical» parts must have redundancy defined

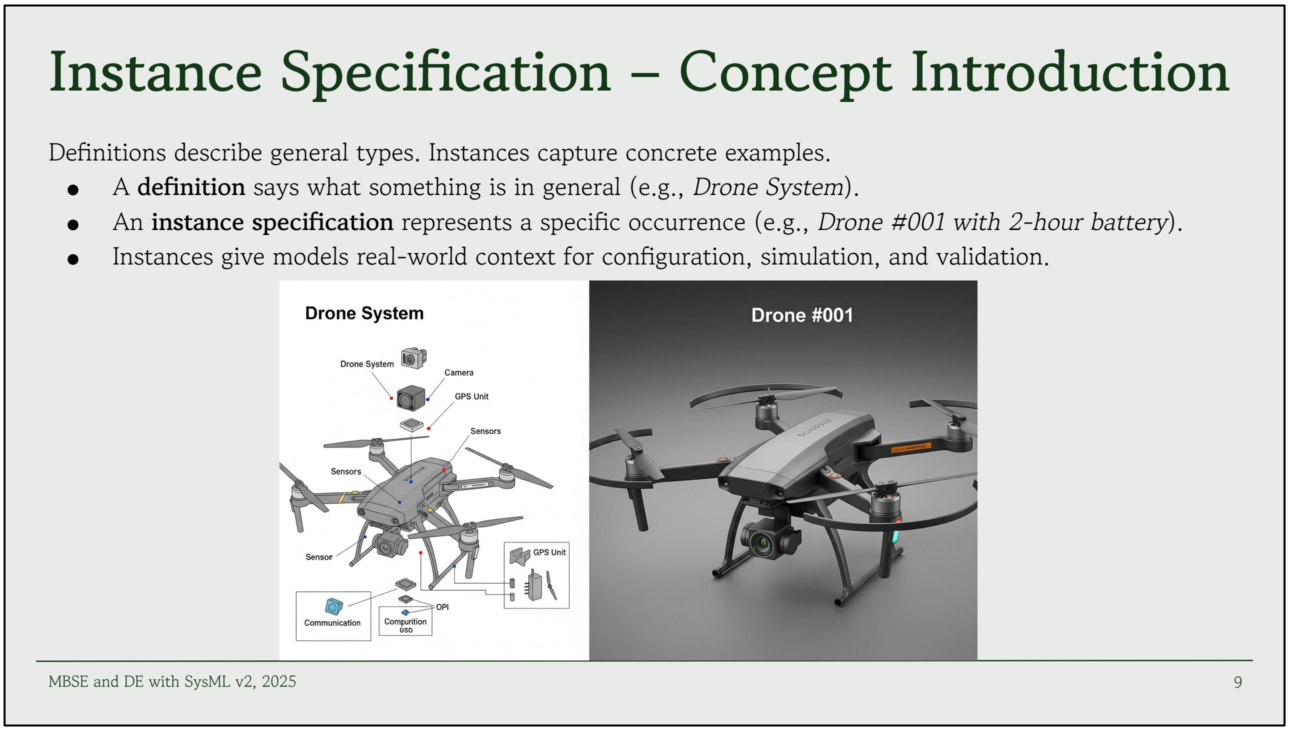
Together:

- The **library** supplies shared building blocks
- The **profile** enforces UAV-specific semantics and rules

This example ties everything together.
Suppose we are modeling a UAV. Instead of defining basic subsystems from scratch, we import them from a **UAV Library**: propulsion system, battery pack, sensors, and so on. These definitions are reusable across multiple drone projects.
At the same time, we apply a **UAV Profile**. The profile extends SysML with UAV-specific stereotypes, such as «FlightCritical» for essential avionics or «Payload» for mission equipment. It also brings domain rules, like requiring redundancy for all flight-critical components or linking certification requirements to regulatory standards.
This combination shows the power of SysML v2 modularization:

- **Libraries** give us consistent, reusable content.
- **Profiles** ensure the model reflects the language, constraints, and rules of the UAV domain.

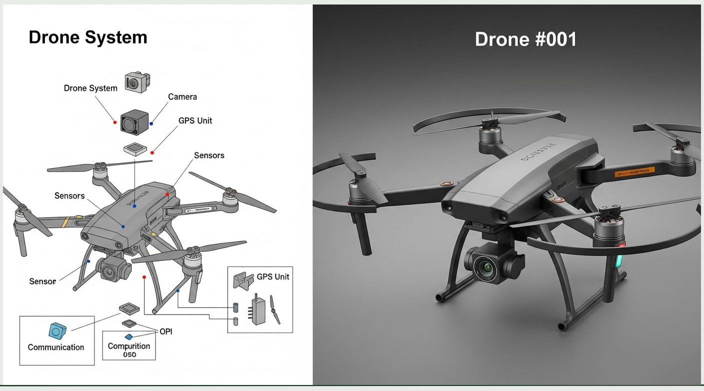
For our course project, you'll start by using a small subsystem library and then apply a lightweight UAV profile to make your model look and behave like a real aerospace engineering artifact.



Instance Specification – Concept Introduction

Definitions describe general types. Instances capture concrete examples.

- A **definition** says what something is in general (e.g., *Drone System*).
- An **instance specification** represents a specific occurrence (e.g., *Drone #001 with 2-hour battery*).
- Instances give models real-world context for configuration, simulation, and validation.



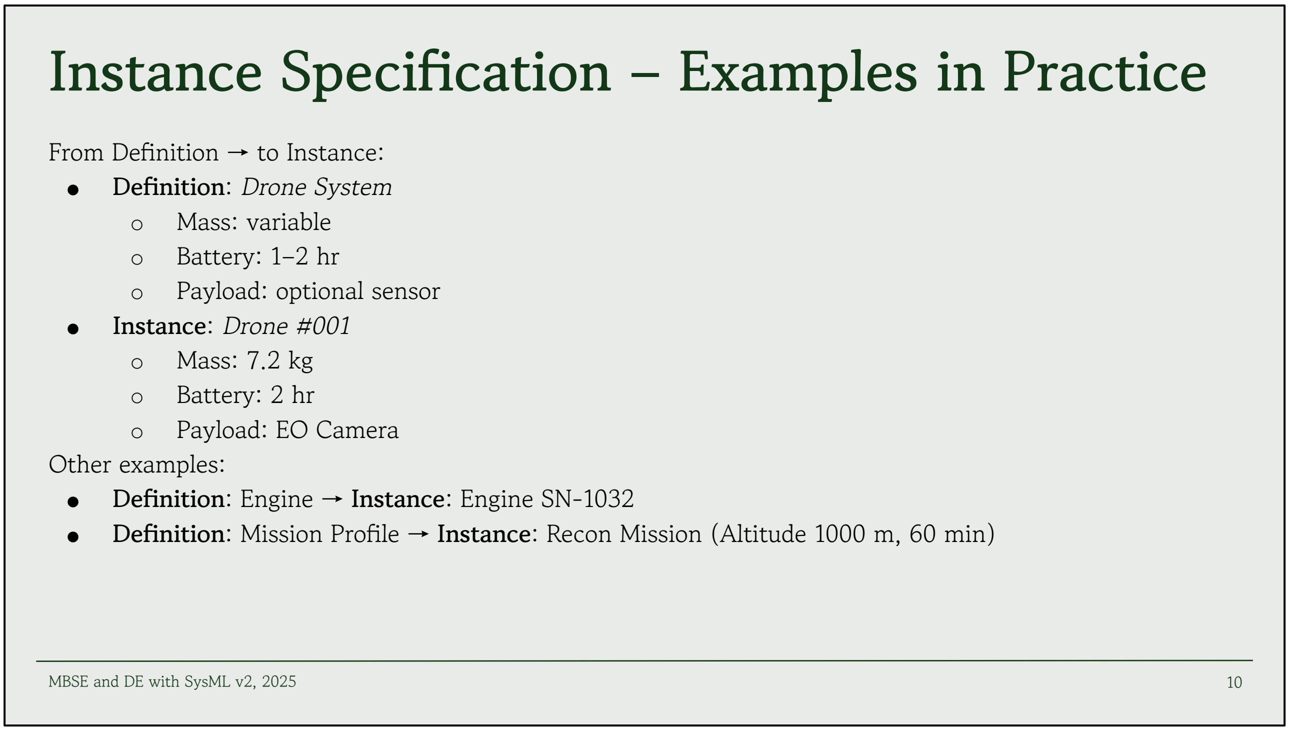
Up to now, most of our modeling has focused on **definitions** — describing general types of parts, actions, or systems. For example, we defined a *Drone System* with properties like mass, power, and payload capacity.

But engineering is not only about general types. We often need to capture **specific cases**:

- A particular UAV with a given serial number and actual battery size.
- A mission scenario where the drone carries one sensor package instead of another.
- A prototype configuration used in testing.

This is where **instance specifications** come in. They turn abstract definitions into **concrete examples** that can be analyzed, simulated, and validated.

Think of definitions as the blueprint, and instances as the real units you build, configure, and fly. Without instance specifications, we cannot link the model to the real system or run meaningful analyses like trade-offs, FMEA, or simulation loops.



Instance Specification – Examples in Practice

From Definition → to Instance:

- **Definition**: *Drone System*
  - Mass: variable
  - Battery: 1–2 hr
  - Payload: optional sensor
- **Instance**: *Drone #001*
  - Mass: 7.2 kg
  - Battery: 2 hr
  - Payload: EO Camera

Other examples:

- **Definition**: Engine → **Instance**: Engine SN-1032
- **Definition**: Mission Profile → **Instance**: Recon Mission (Altitude 1000 m, 60 min)

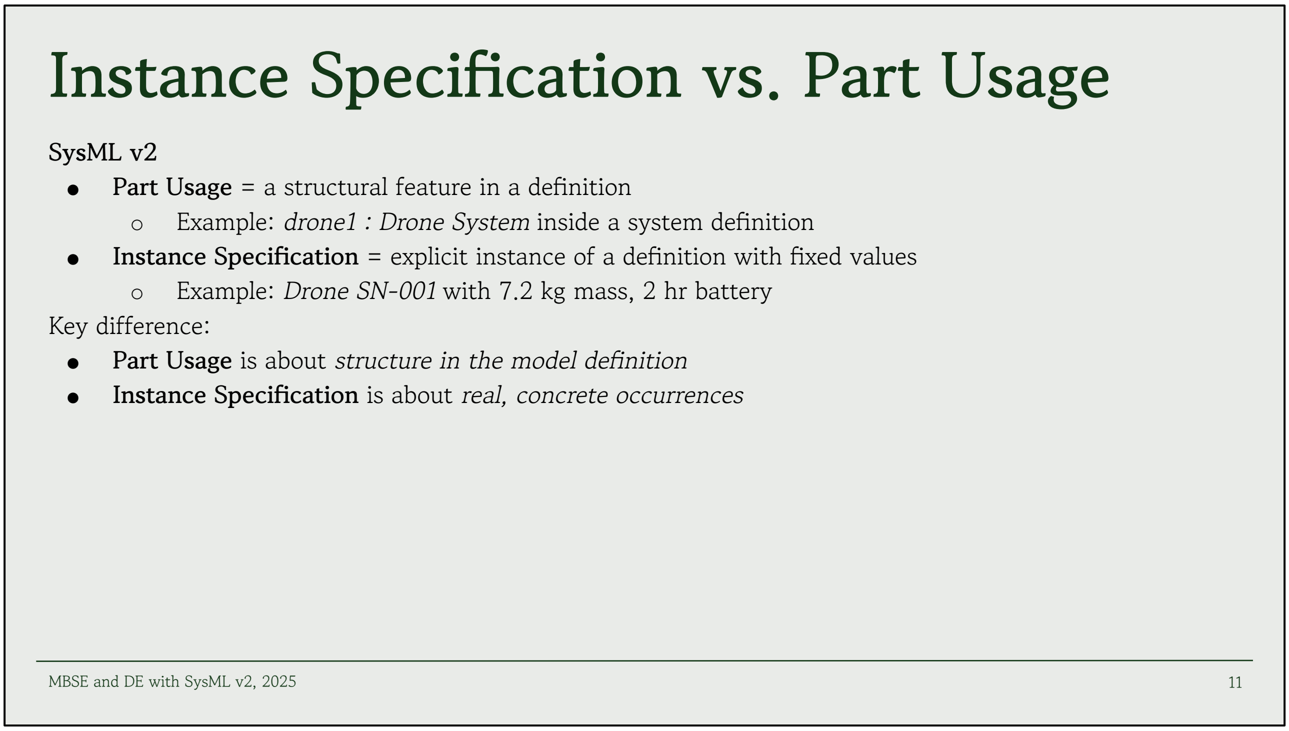
To see how this works in practice, let's take our drone example.

- At the **definition level**, we model a *Drone System*. It has properties like mass, battery capacity, and payload slots — but the values are general or variable.
- At the **instance level**, we create *Drone #001*. Here we fill in the actual numbers: 7.2 kg mass, 2-hour battery, and a payload with an EO camera.

This shift from general to specific happens everywhere in systems engineering:

- An **Engine definition** captures the general design, but an **engine instance** is the actual unit you install in the aircraft.
- A **Mission Profile definition** outlines possible operations, but an **instance** is the concrete mission you're running today, with exact altitude and duration.

Instance specifications make models **actionable**. They are the bridge between system design and real-world execution. They let us simulate, test, and validate not just "a drone" but *this drone* under *this configuration*.


Instance Specification vs. Part Usage

SysML v2

- **Part Usage** = a structural feature in a definition
  - Example: *drone1 : Drone System* inside a system definition
- **Instance Specification** = explicit instance of a definition with fixed values
  - Example: *Drone SN-001* with 7.2 kg mass, 2 hr battery

Key difference:

- **Part Usage** is about *structure in the model definition*
- **Instance Specification** is about *real, concrete occurrences*

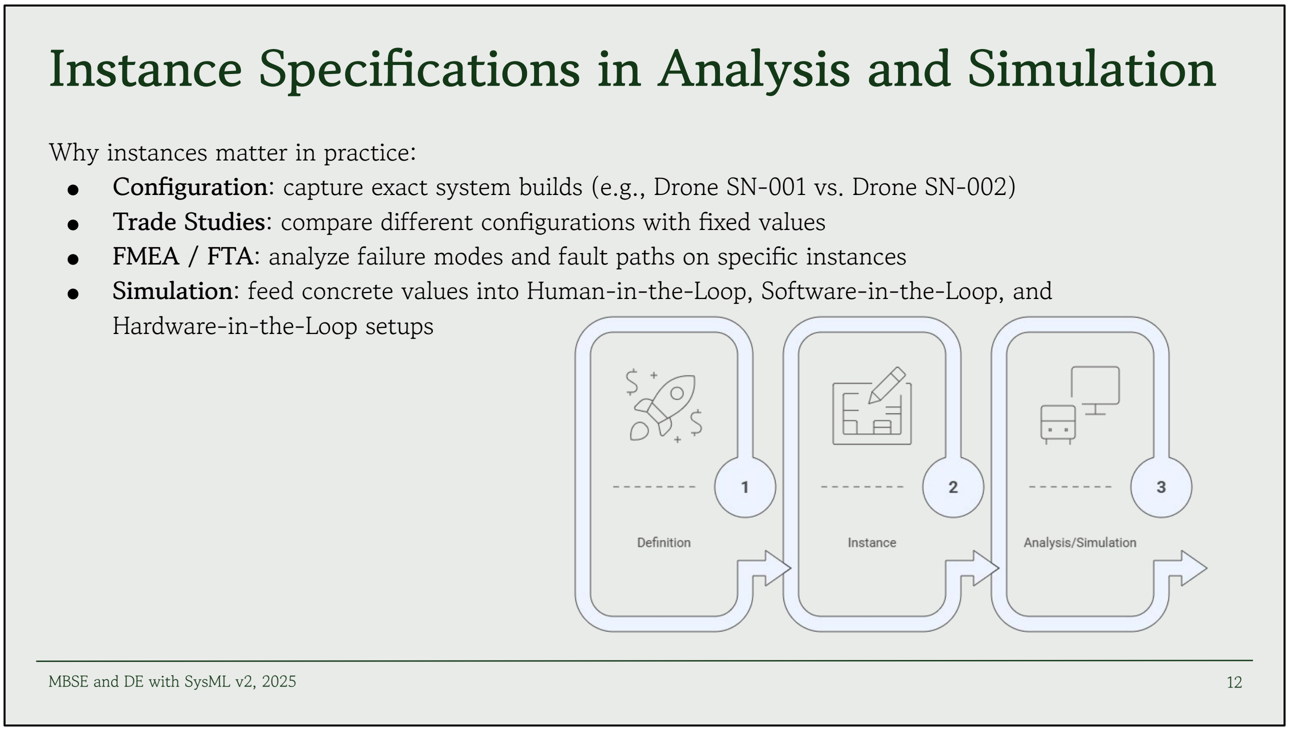
In **SysML v2**, the concept has been made clearer and separated:

- A **Part Usage** is part of the *definition layer*. It tells us that a system includes a part of a given type. For example, in a system definition we might have *drone1 : DroneType*. This still describes structure, not a real instance.
- An **Instance Specification** in SysML v2 is where you describe a specific, concrete occurrence. For example, *Drone SN-001* with actual values for mass and battery capacity.

So:

- **Part Usage = slot in the definition world**
- **Instance Specification = actual configured example**

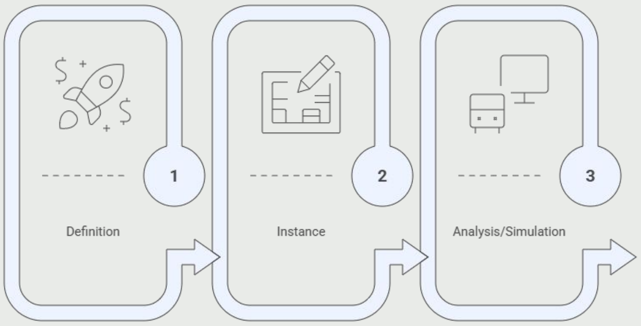
This separation makes models easier to scale, reuse, and analyze. You don't mix general design with specific cases — you can keep both, linked but distinct.



Instance Specifications in Analysis and Simulation

Why instances matter in practice:

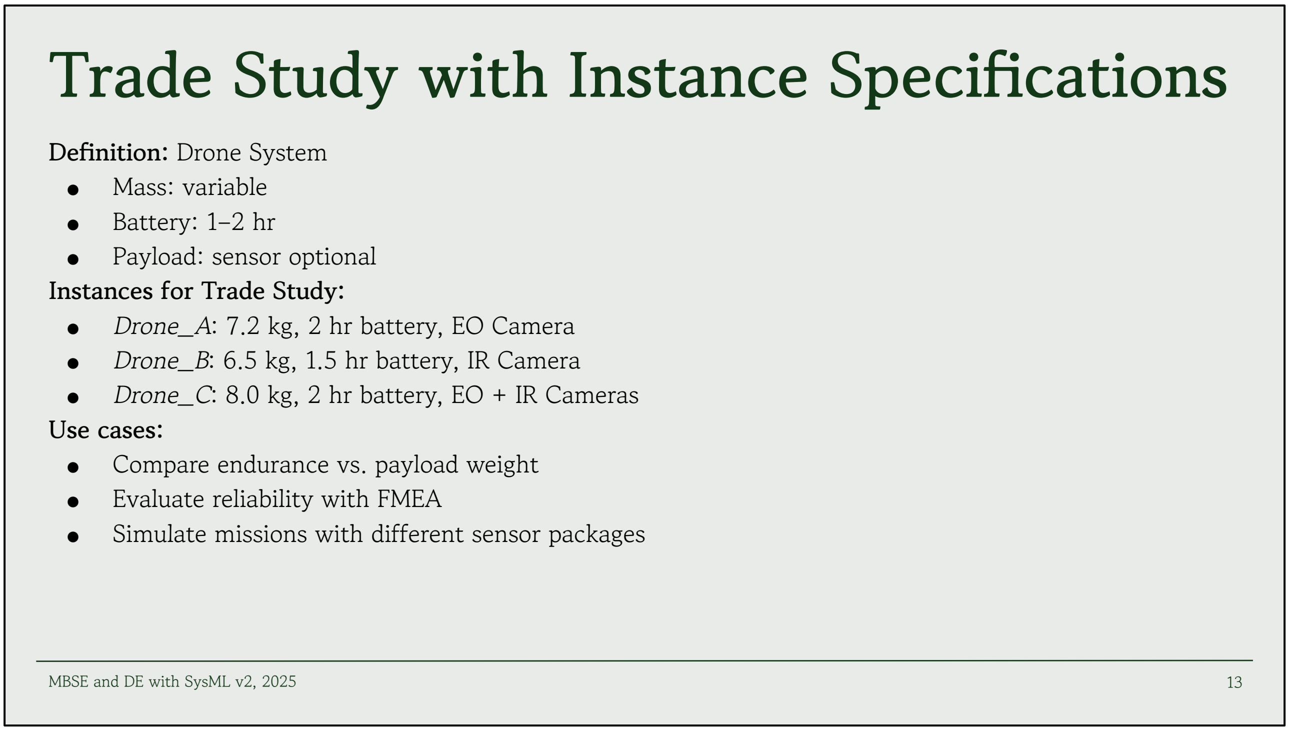
- **Configuration**: capture exact system builds (e.g., Drone SN-001 vs. Drone SN-002)
- **Trade Studies**: compare different configurations with fixed values
- **FMEA / FTA**: analyze failure modes and fault paths on specific instances
- **Simulation**: feed concrete values into Human-in-the-Loop, Software-in-the-Loop, and Hardware-in-the-Loop setups



Instance specifications are not just theory — they make analysis and simulation meaningful.

- For **configuration management**, an instance lets you say, “This is Drone SN-001 with a 2-hour battery and EO camera,” distinct from Drone SN-002 with different equipment.
- In **trade studies**, you can create multiple instances from the same definition, each with different parameter values, and then compare them.
- For **FMEA (Failure Mode and Effects Analysis)** and **FTA (Fault Tree Analysis)**, you need concrete instances. These analyses explore how specific configurations behave under faults, not just the generic type.
- For **simulation**, tools require real numbers. Instance specifications provide the inputs for Human-in-the-Loop, Software-in-the-Loop, and Hardware-in-the-Loop validation. Without instances, you cannot bridge the abstract model to executable scenarios.

In short, instance specifications connect SysML models to the world of real engineering decisions — configurations, analyses, and testbeds.



Trade Study with Instance Specifications

Definition: Drone System
- Mass: variable
- Battery: 1–2 hr
- Payload: sensor optional

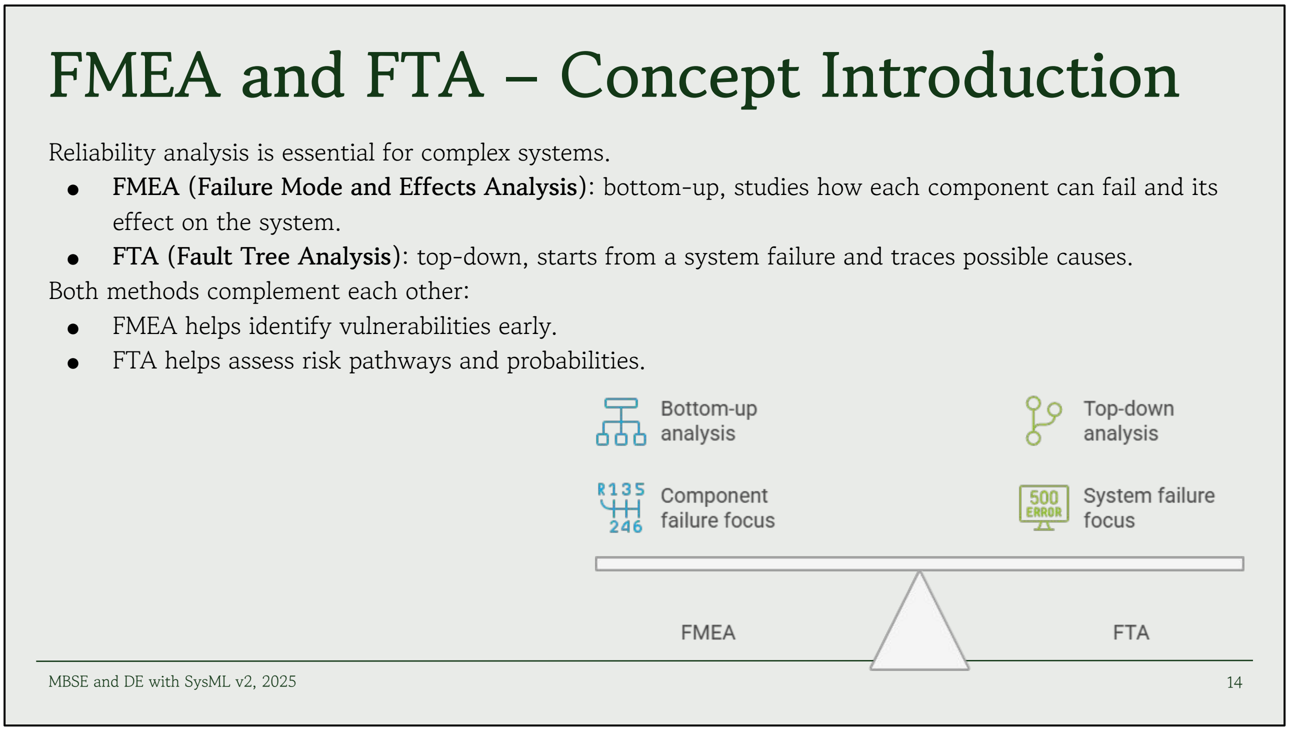
Instances for Trade Study:
- *Drone_A*: 7.2 kg, 2 hr battery, EO Camera
- *Drone_B*: 6.5 kg, 1.5 hr battery, IR Camera
- *Drone_C*: 8.0 kg, 2 hr battery, EO + IR Cameras

Use cases:
- Compare endurance vs. payload weight
- Evaluate reliability with FMEA
- Simulate missions with different sensor packages

The **definition** *Drone System* is general. It only says drones have mass, a battery range, and an optional payload. By itself, this definition cannot answer engineering questions like “Which configuration has better endurance?”
To explore design options, we create **instances**:
- *Drone_A* is a lighter drone with a 2-hour battery and EO camera.
- *Drone_B* trades endurance for reduced mass and an IR camera.
- *Drone_C* carries both sensors, at the cost of higher weight.

By modeling these as **instance specifications**, we can:
- Run trade studies that compare endurance vs. payload.
- Perform **FMEA** to see which configuration is more failure-prone.
- Run simulations of real missions, testing each option under Human-in-the-Loop or Software-in-the-Loop environments.

This is the value of instance specifications: they let us test *concrete choices* rather than abstract types.



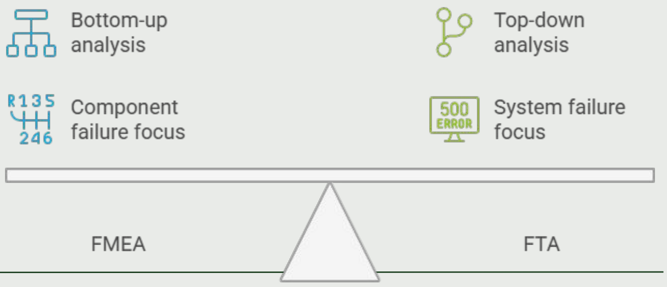
FMEA and FTA – Concept Introduction

Reliability analysis is essential for complex systems.

- **FMEA (Failure Mode and Effects Analysis)**: bottom-up, studies how each component can fail and its effect on the system.
- **FTA (Fault Tree Analysis)**: top-down, starts from a system failure and traces possible causes.

Both methods complement each other:

- FMEA helps identify vulnerabilities early.
- FTA helps assess risk pathways and probabilities.



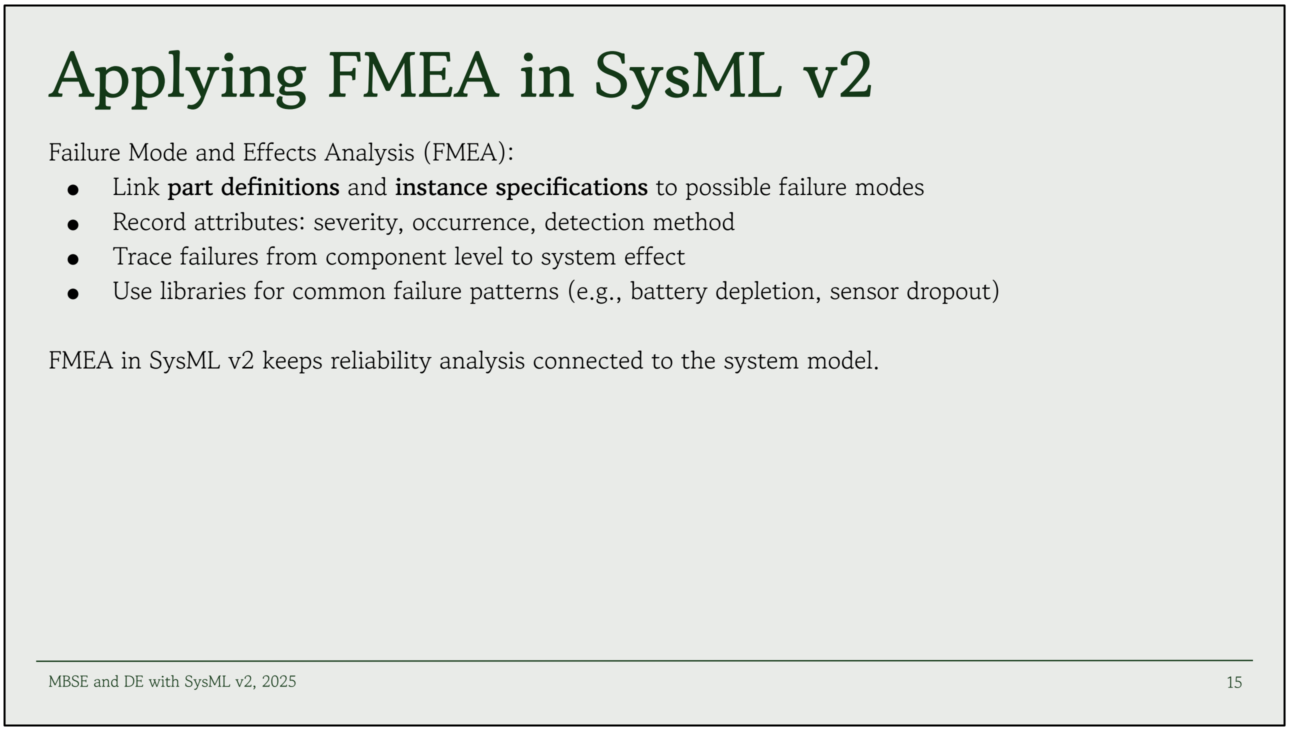
As our models grow, we must think about reliability, not just functionality. Two common methods are **FMEA** and **FTA**.

- **FMEA** is a *bottom-up* method. You look at each part — battery, motor, sensor — and ask, "How could this fail, and what happens if it does?" You then record the effects, severity, likelihood, and detection methods.
- **FTA** is a *top-down* method. You start with a system failure, like "Drone crashes during mission," and build a logical tree of contributing faults, such as motor loss, sensor blackout, or power failure.

The two methods complement each other:

- FMEA is good at finding weak spots early in design.
- FTA is good at analyzing risk chains and calculating probabilities of top-level failures.

In SysML v2, both can be integrated into the model, making them traceable and reusable rather than stand-alone spreadsheets.



Applying FMEA in SysML v2

Failure Mode and Effects Analysis (FMEA):

- Link **part definitions** and **instance specifications** to possible failure modes
- Record attributes: severity, occurrence, detection method
- Trace failures from component level to system effect
- Use libraries for common failure patterns (e.g., battery depletion, sensor dropout)

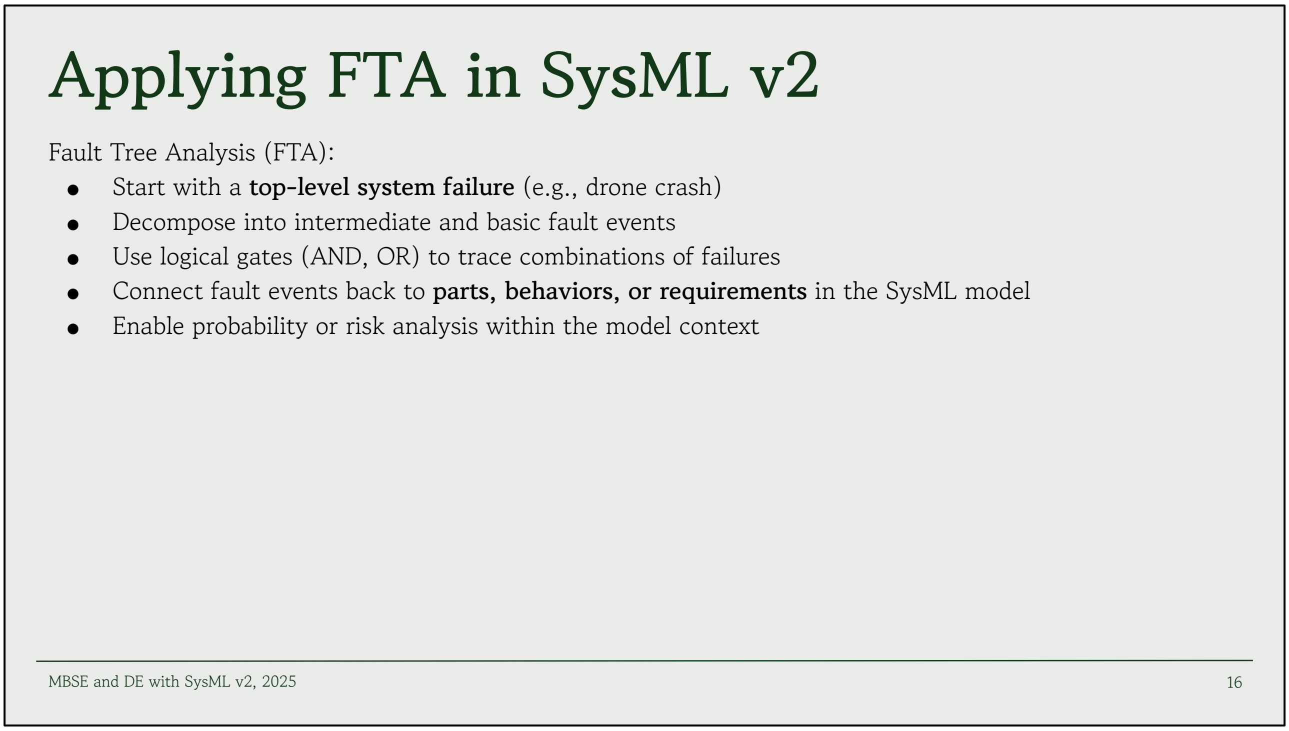
FMEA in SysML v2 keeps reliability analysis connected to the system model.

In traditional practice, FMEA is often a separate spreadsheet. The drawback is that it quickly drifts out of sync with the system design. SysML v2 solves this by embedding FMEA inside the model.

Here's how it works:

- For each **part definition**, you create failure modes. For example, a battery might fail by overheating, depletion, or short circuit.
- For each **instance specification**, you can record the actual conditions — such as the battery in *Drone #001* that has a 2-hour endurance but a higher overheating risk in hot climates.
- You annotate each failure mode with severity, occurrence, and detection. This lets you assess risk systematically.
- Libraries can capture common failure modes so that teams reuse knowledge across projects.

By modeling FMEA in SysML v2, you link failures directly to system structure and behavior. That way, when the system evolves, the analysis evolves too. It is no longer just a static document — it becomes part of the engineering model.



Applying FTA in SysML v2

Fault Tree Analysis (FTA):

- Start with a **top-level system failure** (e.g., drone crash)
- Decompose into intermediate and basic fault events
- Use logical gates (AND, OR) to trace combinations of failures
- Connect fault events back to **parts, behaviors, or requirements** in the SysML model
- Enable probability or risk analysis within the model context

While FMEA works bottom-up, **FTA works top-down**.
We begin with a system-level failure, such as “Drone crash during mission.” From there, we build a tree of possible causes:

- A **Power Loss** branch, which may result from battery overheating or connector failure.
- A **Navigation Failure** branch, which could stem from GPS signal loss or sensor malfunction.

Each branch is connected with **logic gates**:

- An **OR gate** means any one fault is enough to cause the failure.
- An **AND gate** means multiple conditions must occur together.

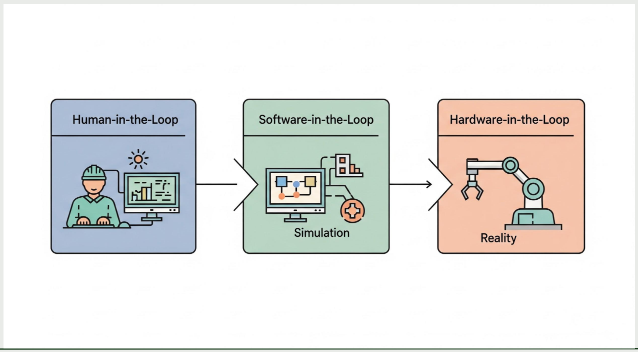
In SysML v2, these fault events are not just boxes in a diagram. They can be linked directly to parts, behaviors, or requirements in the model. This keeps the analysis consistent with design data. You can also attach probability values to basic events, making it possible to calculate overall system risk.
By capturing FTA inside SysML, we integrate safety analysis into the same environment where requirements, structures, and behaviors are modeled. That reduces duplication and improves traceability.

X-in-the-Loop – Concept Introduction

“X-in-the-Loop” = progressive testing and validation:

- **Human-in-the-Loop** (HITL): operators interact with the model or simulation
- **Software-in-the-Loop** (SITL): real code runs in the simulated environment
- **Hardware-in-the-Loop** (HIL): physical components integrated into the loop

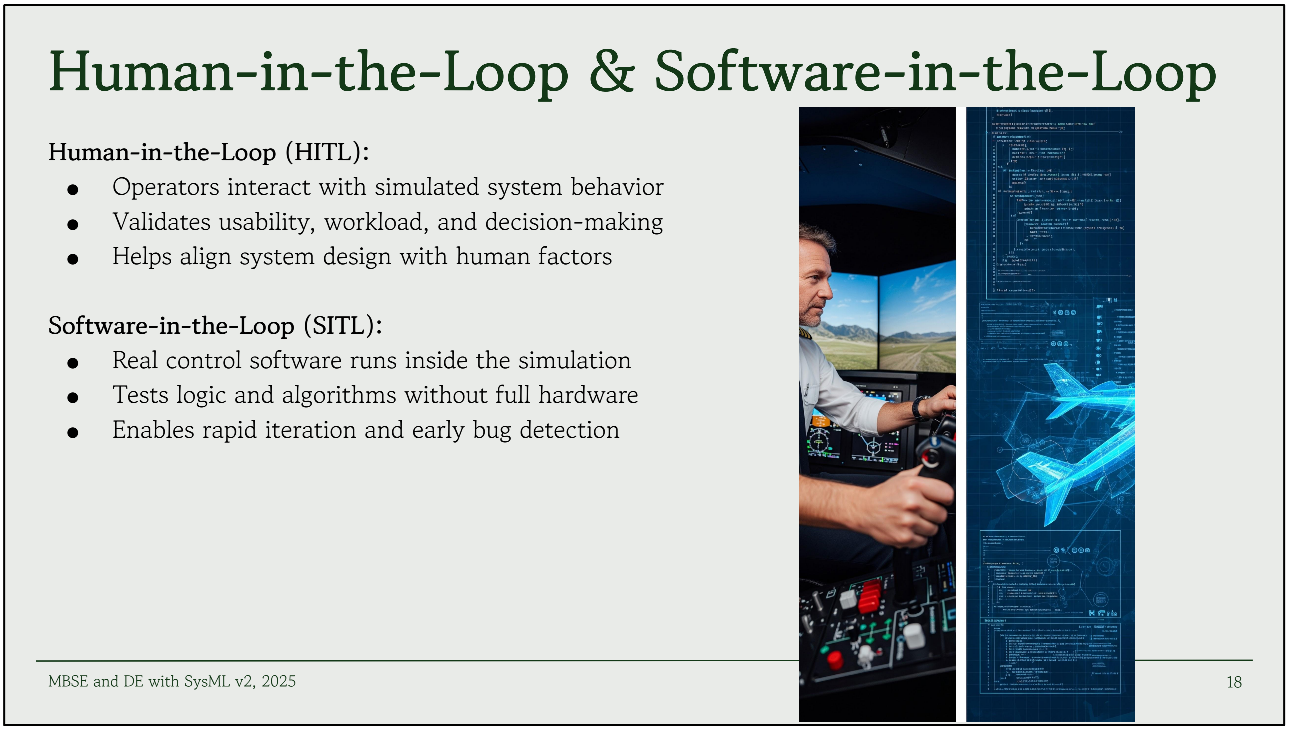
Goal: close the gap between abstract models and real-world performance.



The phrase “X-in-the-Loop” refers to a family of methods where we progressively bring the model closer to reality.

- With **Human-in-the-Loop**, we put operators in the loop early. For example, a pilot flies a simulated UAV mission. This checks usability and decision-making, not just system logic.
- With **Software-in-the-Loop**, the actual control code runs inside a simulation. Instead of abstract behavior, we test the real software in a safe, controlled environment.
- With **Hardware-in-the-Loop**, we plug real components into the system model. For example, connecting a physical flight controller or sensor into a drone simulation.

The common goal is to move step by step from model to reality. This reduces risk, catches integration issues early, and makes validation more realistic without waiting for a full prototype.



Human-in-the-Loop & Software-in-the-Loop

Human-in-the-Loop (HITL):

- Operators interact with simulated system behavior
- Validates usability, workload, and decision-making
- Helps align system design with human factors

Software-in-the-Loop (SITL):

- Real control software runs inside the simulation
- Tests logic and algorithms without full hardware
- Enables rapid iteration and early bug detection

The first two stages of X-in-the-Loop are powerful because they come early, before hardware is even ready.

- In **Human-in-the-Loop**, we connect people directly to the simulation. A UAV operator can run missions in a virtual environment and give feedback on interface, workload, or decision paths. This exposes human-system interaction issues that a pure model would never reveal.
- In **Software-in-the-Loop**, we drop in the real code. Instead of abstract activity diagrams, the UAV's actual control software runs against the simulation of the airframe and environment. This validates algorithms, catches bugs, and allows safe stress testing before connecting to real hardware.

Together, HITL and SITL bring realism into the model long before hardware is available. They ensure both humans and software are ready for integration.

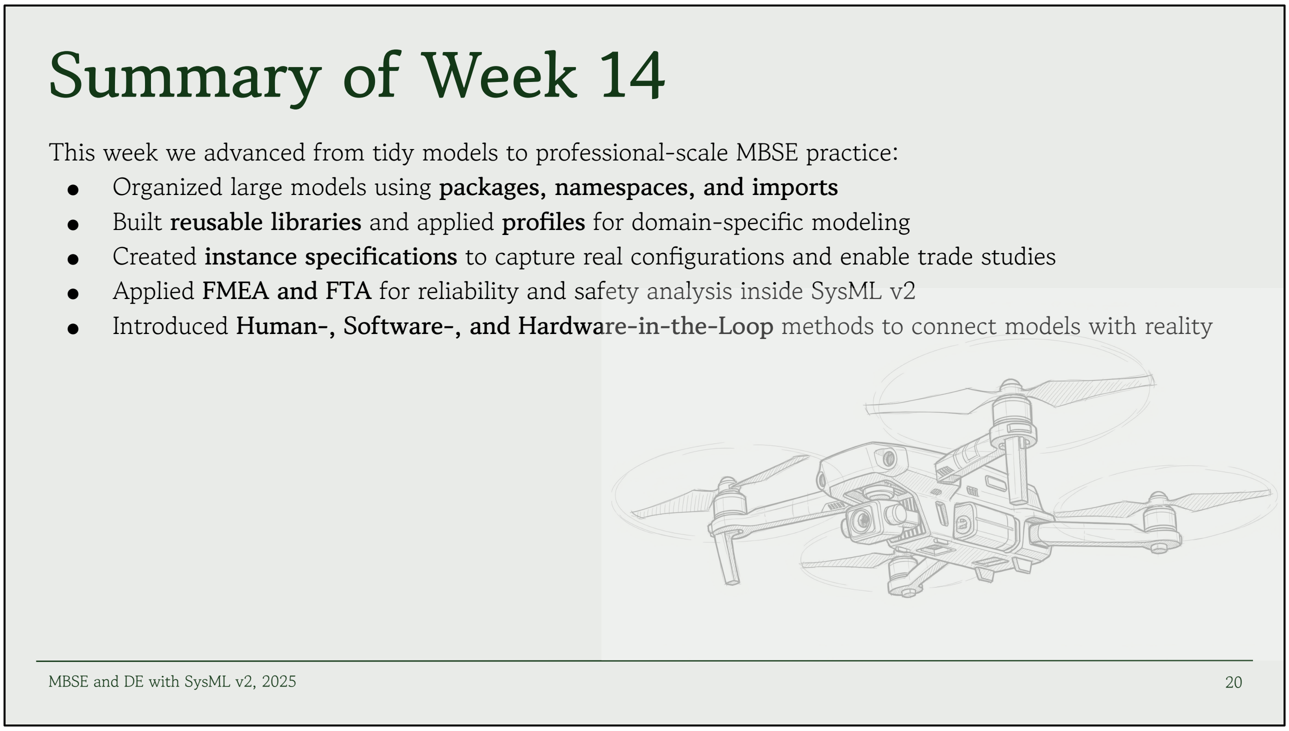
Hardware-in-the-Loop (HIL)

- Integrates real hardware with the simulated system
- Validates interfaces, timing, and performance under realistic conditions
- Finds issues before full system assembly
- Bridges the final step from digital model to physical prototype

The final stage is **Hardware-in-the-Loop (HIL)**.
Here we connect real components to the simulation. For example, instead of modeling the flight controller, we plug in the actual unit. The simulated drone provides inputs like sensor signals, and the hardware outputs control commands.
This setup validates:

- **Interfaces**: Are signals and data exchanges handled correctly?
- **Timing**: Does the hardware respond fast enough under mission conditions?
- **Performance**: How does the real hardware behave under edge cases simulated in software?

HIL allows engineers to uncover problems before building the entire prototype. It reduces integration surprises, lowers cost, and improves confidence in the design.
By combining Human-in-the-Loop, Software-in-the-Loop, and Hardware-in-the-Loop, we create a staged validation pipeline that gradually shifts from abstract modeling to real-world operation.



Summary of Week 14

This week we advanced from tidy models to professional-scale MBSE practice:

- Organized large models using **packages, namespaces, and imports**
- Built **reusable libraries** and applied **profiles** for domain-specific modeling
- Created **instance specifications** to capture real configurations and enable trade studies
- Applied **FMEA and FTA** for reliability and safety analysis inside SysML v2
- Introduced **Human-, Software-, and Hardware-in-the-Loop** methods to connect models with reality

This week was about moving beyond "just modeling" into practices that make MBSE scalable, reusable, and directly tied to engineering reality.

- You learned to **organize models** so they stay clear as they grow.
- You saw how **libraries** promote reuse and how **profiles** tailor SysML to your domain.
- You explored **instance specifications**, the bridge from general design to concrete configurations.
- You applied **FMEA** and **FTA** as part of the model, integrating safety and reliability into the same framework as design.
- You discovered the role of **X-in-the-Loop** — staged methods that validate humans, software, and hardware progressively.

Together, these practices transform SysML v2 from a drawing tool into a professional MBSE environment.

QUESTION!