

---

# **AI-ASSISTED SYSTEMS ENGINEERING WITH OPM**

**VOJTĚCH MERUNKA**

CZECH UNIVERSITY OF LIFE SCIENCES, PRAGUE, CZECHIA

DECEMBER 2025

# CONTENTS

- a) **emergence**
- b) **LLM** (generative AI) – benefits, problems, limitations
- c) **FP** and **COCOMO** methods
- d) **OPM** – principles, example

# EMERGENCE

**Emergence** refers to the **phenomenon** in which new properties or behaviors arise at the level of a whole system even though they are absent in the individual components.

Emergent properties appear only through the interactions of many simple units, meaning that the whole exhibits capabilities that no part possesses on its own.

**Emergence is fundamental in nature, in computer science, in LLMs, and in systems engineering.**

Underestimating emergence gives rise to dangerous forms of **reductionism** in modern science.

## **example:**

In an ant colony, no single ant understands the colony's goals or strategy. Each ant follows only simple local rules. Yet, the colony as a whole behaves in surprisingly intelligent and adaptive ways - optimizing foraging, regulating temperature, or responding to threats - far beyond the cognitive capacity of any individual ant.

This phenomenon was already examined by **Aristotle**, and modern science has begun to examine emergent phenomena intensively. Carlo Rovelli (\*1956):

*Time is not only relative in Einstein's sense, but also emergently created by microscopic interactions between quantum systems, and therefore time may not a fundamental structure of reality itself.*

# BENEFITS OF LLM

## **Rapid transformation of informal knowledge into structured models**

- Bridges the gap between domain experts (who speak in natural language) and modelers (who need formal structures). Extracts objects, processes, constraints, and relationships from messy documents.

## **Powerful assistant for systems architects**

- Summarizes requirements, identifies dependencies, and detects ambiguities.
- Helps validate completeness and discover missing components.
- Proposes consistent refinements across multiple system layers.

## **Acts as an intelligent query interface**

- Lets humans ask and responds with structured information extracted from models or documentation.

## **Automates documentation**

- Keeps manuals, reports... synchronized with evolving models.

# THERE IS NO TIME IN LLM

## LLMs do not have physical time.

- Token operations have position but no temporal dimension.
- Nothing in the token-processing architecture enforces that causes must precede their effects, which can contribute to hallucinations.
- LLM attention mechanism treats token relations in a largely symmetric manner: it allows bidirectional influence, making the internal token-transformations effectively reversible and order-invariant with respect to causal direction.

Time occurs in the LLM only as a sequence, not as a process with duration.

The causal logic of “if x, then y” is encoded only through positional proximity of tokens within the LLM’s internal configuration; no non-commutative structures exist in the LLMs that could give rise to an emergent flow of physical time.

Understanding that LLMs are not temporal automata helps in interpreting hallucinations: LLMs cannot reconstruct the temporal continuity of the world.

# RUSSELL'S PARADOX

## Russell's Paradox (self-reference inconsistency)

*Q: a barber in a village shaves all and only those men who do not shave themselves; does the barber shave himself?*

*if a barber shaves himself, he should not to shave himself*

*if a barber does not shave himself, he should to shave himself*

Russell's paradox exposes a fundamental problem that occurs whenever a system allows unrestricted abstraction without any constraints.

## Consequences

- LLMs have no mechanism to block paradoxical constructs.
- LLMs can output instable contradictory rules, loops, or self-negating statements which are interpreted as hallucinations.
- **External structure is required to avoid paradox-driven errors.**

# WITTGENSTEINIAN NONSENSE AND TARSKI'S SOLUTION FOR LLM

**LLMs can produce sentences that are syntactically perfect but semantically impossible.** These sentences are grammatically correct but cannot picture any possible configuration of the real world:

*Baron Münchhausen pulled himself out of the swamp by his own hair.*  
(physical nonsense)

*The database query returned results that were inserted after this query.*  
(temporal nonsense)

For some statements it is not even possible to verify whether a statement is true or false:

*The computer regretted deleting the user's feelings.*  
(semantic nonsense)

**Tarski's solution of this LLMs problem:**

LLMs cannot evaluate themselves internally. **Semantic correctness requires stepping outside the LLM system that produced the statement.**

# LIMITATIONS OF LLM

## LLMs are not capable of recognizing their own errors because

- LLMs compute the probability of linguistic continuation, not epistemic certainty.
- LLMs lack an explicit model of the world; they only operate on patterns in text, not on ontological structures or states of reality.
- LLMs possess no mechanism for verification: no constraints, no logic engine, no model checking, no grounding in external truth.
- Apparent causal sentences (“*because*”, “*therefore*”, ...) reflect statistical correlations only, not causal mechanisms.
- LLMs cannot detect contradictions that require state tracking (e.g., identity, persistence, object continuity).
- Confidence estimates from logits reflect distributional fit, not correctness; high probability  $\neq$  truth.

# FUNCTION POINT METHOD

**Function Point Method** (FPM) is a standardized software-engineering technique for measuring the software complexity from user requirements based on what the system does for the user, not how it is technically implemented. It originates from Allan Albrecht (IBM, 1979) and is maintained in ISO/IEC 20926. Its purpose is to provide an implementation-independent metric.

FPM is based on five function types (EI, EO, EQ, ILF, EIF), which can be derived from the user-requirements specification, each multiplied by predefined weighting factors and then summed.

**For different programming languages, empirical averages are known regarding how many lines of code are typically required to implement a program corresponding to one function point of complexity.**

for example,

in the standard C programming language one must write about 130 lines of code, whereas in a modern high-level language such as Python only about 20 lines of code are needed to implement the same unit of one function point.

# COCOMO

The **Constructive Cost Model** (COCOMO) was introduced by Barry W. Boehm in 1981 as a quantitative, empirically grounded method for estimating software-development effort. COCOMO II (Boehm et al., 2000; updated 2009) generalizes the original model to accommodate modern development practices.

**COCOMO II starts from an estimated size expressed in source lines of code (SLOC) or function points (FP), and then adjusts the required effort using 17 cost drivers and 5 scale factors that represent project-specific conditions.**

Using COCOMO II, we can derive:

- **the estimated project development effort in person-months,**
- **the recommended project duration in months, and**
- **the corresponding recommended size of a project-development team.**

# OPM – OBJECT PROCESS METHODOLOGY

**OPM** is the **ISO 19450:2024** standard intended for rigorous specification, analysis, and verification of systems, and it is particularly well suited to simulation-based and other semantics-driven validation techniques.

**OPM is bimodal**, every model is expressed simultaneously in a natural-like language **OPL** = *Object-Process Language* and in its fully equivalent graphical form, the **OPD** = *Object-Process Diagram*. Both forms describe the same underlying model. Available modelling tools maintain full synchronization between them.

**OPL** provides a human-readable textual view that accelerates understanding of **OPD** graphical diagrams, and it has also proven exceptionally effective for processing by **LLMs**. The combination of **OPM** and **LLM** enables a new paradigm in systems and software engineering often referred to as **AI-assisted system engineering**.

# OPM COMPARED WITH UML AND BPMN

**OPM differs fundamentally from both UML and BPMN in scope, expressive principles, and conceptual integration.**

Whereas UML is primarily oriented toward describing the software aspects, **OPM models the entire system**, including elements, structures, and behaviors that exist outside the software boundary. This makes OPM suitable for socio-technical, cyber-physical, and organizational systems where software is only one subsystem within a broader operational context.

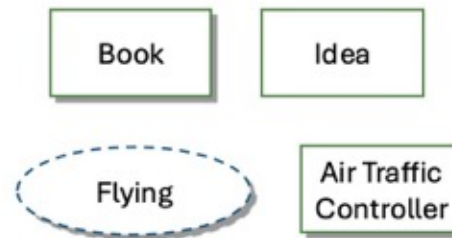
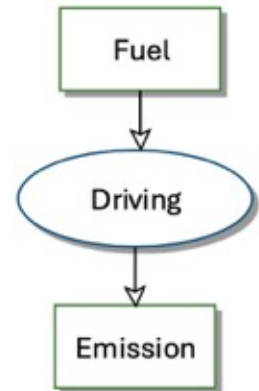
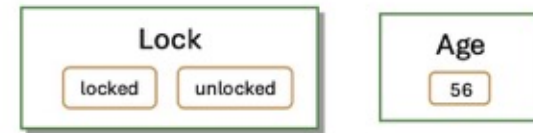
**OPM differs from UML by using a single diagram** type built on a minimal, well-defined ontology of only objects and processes. This ontology creates a symmetrical modeling structure, where relations among objects and relations among processes follow parallel principles and integrate seamlessly into one coherent model. Despite its simplicity, the formal semantics of OPM are Turing complete, allowing it to express any computationally definable behavior. This combination of ontological clarity, symmetries, and full computational expressiveness gives OPM a unified modeling power that UML's heterogeneous diagram set cannot match. OPM supports **semantic zooming and refinement across levels of detail**, OPM paradigm fully replaces the entire UML family, eliminating the need to learn numerous notational conventions.

**OPM also serves as an alternative to BPMN** which concentrate exclusively on workflows, tasks, and control flow, but omit the explicit interplay between the system structure and behavior.

# OPM – OBJECTS AND PROCESSES

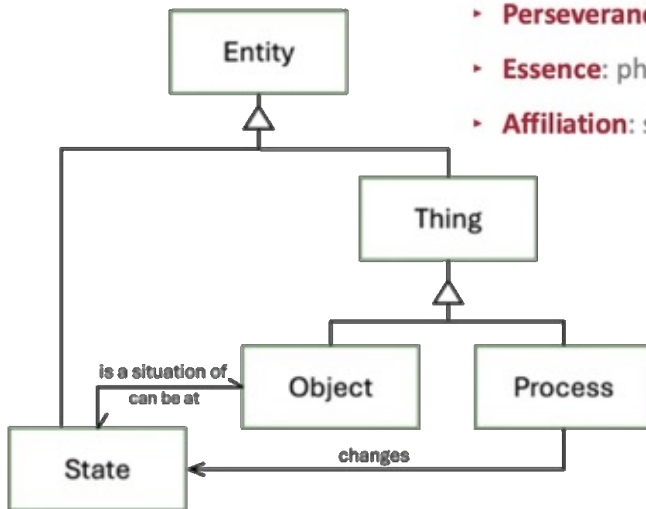
The foundation of **OPM** (both the diagrammatic graphical form **OPD** and textual form **OPL**) consists of things and the relationships between them. Things come in two kinds: **Objects** and **Processes**.

- Objects and Processes are **Things**
- An Object is a Thing that **exists**
- Objects can have **states**, which describe their situation or value
- States exist only in the context of the Object, and they are **mutually exclusive**
- A Process is a Thing that **transforms** an Object
- Transformation can be **creation, consumption, or effect**
- Things have three independent properties:
  - **Perseverance**: persistent vs. transient
  - **Essence**: physical vs. informatical
  - **Affiliation**: systemic vs. environmental



© OptimISE Academy

19



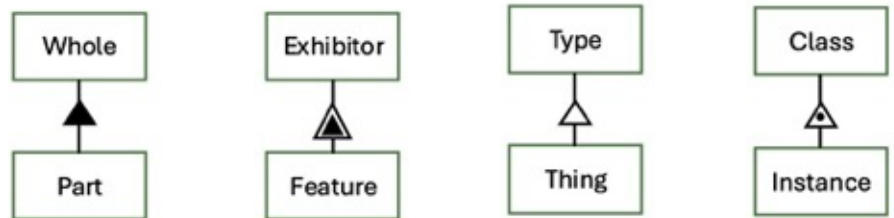
# OPM - LINKS

There are links between things. The fundamental ones are **structural links**, which apply equally to both Objects and Processes.

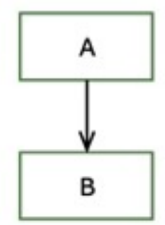
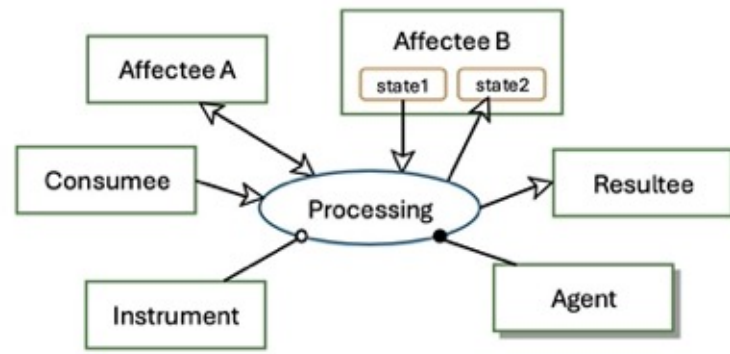
Other **procedural links** follow their own rules.

**In general, each Object in the model should be procedurally linked to at least one Process, and each Process should be procedurally linked to at least one Object.**

- Structural links
  - Fundamental Structural links
    - Aggregation-participation
    - Exhibition-characterisation
    - Generalisation-specialisation
    - Classification-instantiation
  - Tagged structural links
- Procedural links
  - Procedural transforming links
    - Consumption link
    - Result link
    - Effect link / in-out link pair
  - Procedural enabling links
    - Instrument link
    - Agent link



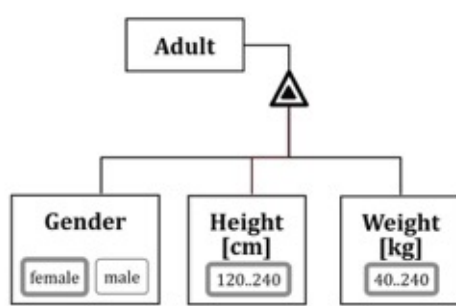
Whole consists of Part. Exhibitor exhibits Feature. Thing is a Type. Instance is an instance of Class.



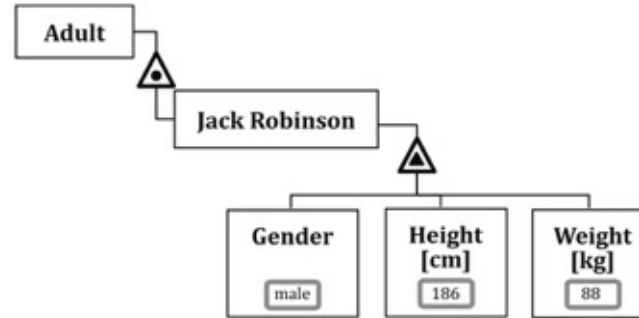
A relates to B.

Processing consumes Consumee. Processing yields Resultee. Processing affects Affectee A. Processing changes Affectee B from state1 to state2. Processing requires Instrument. Agent handles Processing.

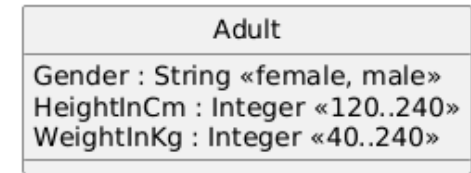
# COMPARING EXAMPLE OPM AND UML



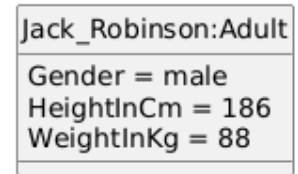
a) Class



b) Instance



UML class



UML instance

**Adult** exhibits **Gender**, **Height in cm**, and **Weight in Kg**. **Jack Robinson** is an instance of **Adult**.  
**Gender** of **Adult** can be **female** or **male**.  
**Height in cm** of **Adult** ranges from **120** to **240**.  
**Weight in Kg** of **Adult** range from **40** to **240**.

**Gender** of **Jack Robinson** is **male**.  
**Height in cm** of **Jack Robinson** is **185**.  
**Weight in kg** of **Jack Robinson** is **88**.

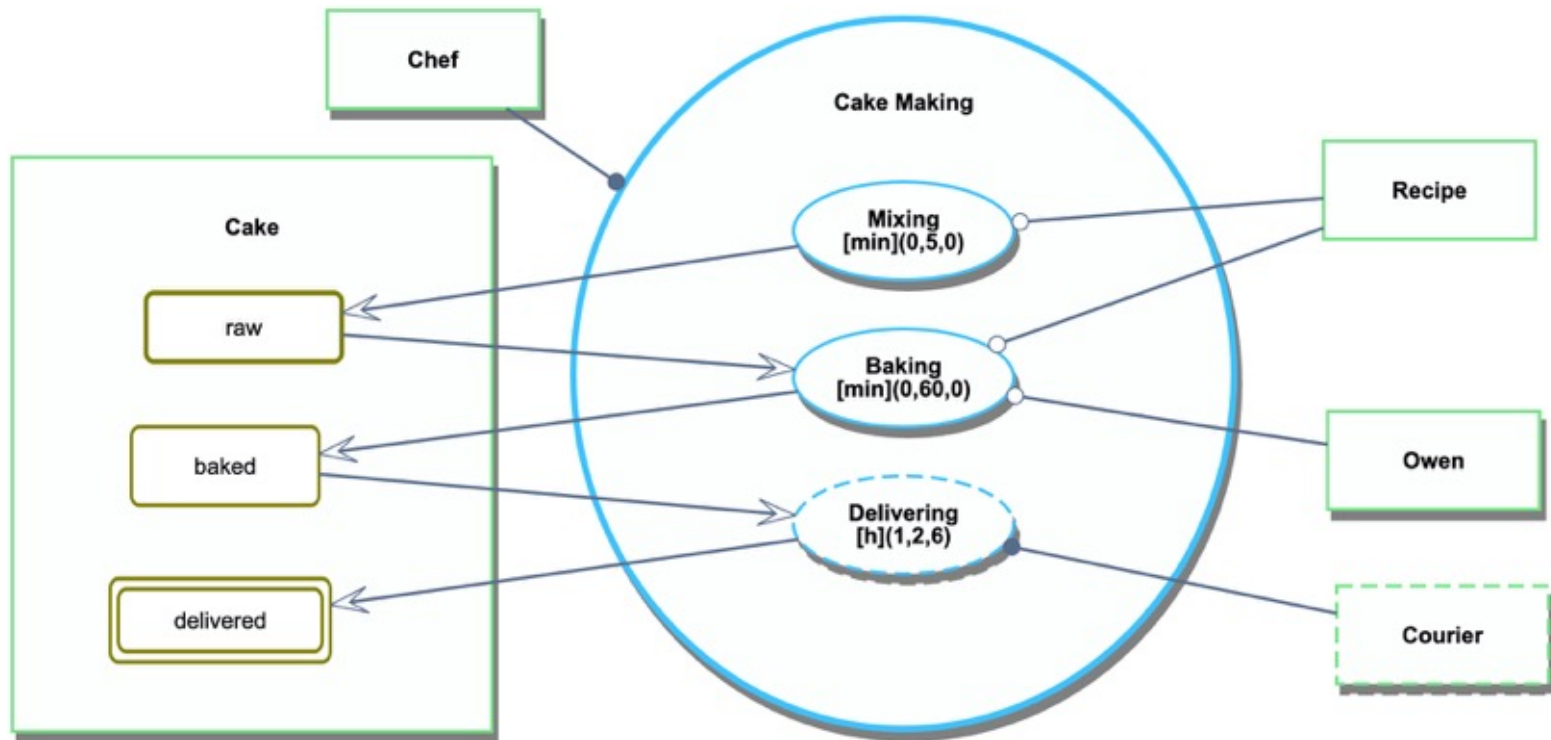
© ISO/PAS 19450:2024, fig. 26

In **OPM**, the **exhibits** link is used to connect an object with its **attributes** (represented as Objects) and optionally with its **behavior** (represented as Processes) that the object performs. This is a fundamental structural link.

In **UML**, attributes and behavior are considered only as internal components of a class, without the possibility of more detailed modeling, because they are not independent conceptual entities. For example, UML does not allow expressing that attributes can change over time or that behavior is a process in itself.

In addition, **OPM is simpler because it uses only a single type of diagram**. This diagram is highly symmetrical, meaning that links between processes and links between objects work according to the same principles. Thanks to its ability to easily zoom in and out between levels of detail, OPM fully replaces the 14 diverse UML diagrams, so there is no need to learn different notations — everything is integrated into one coherent, unified view.

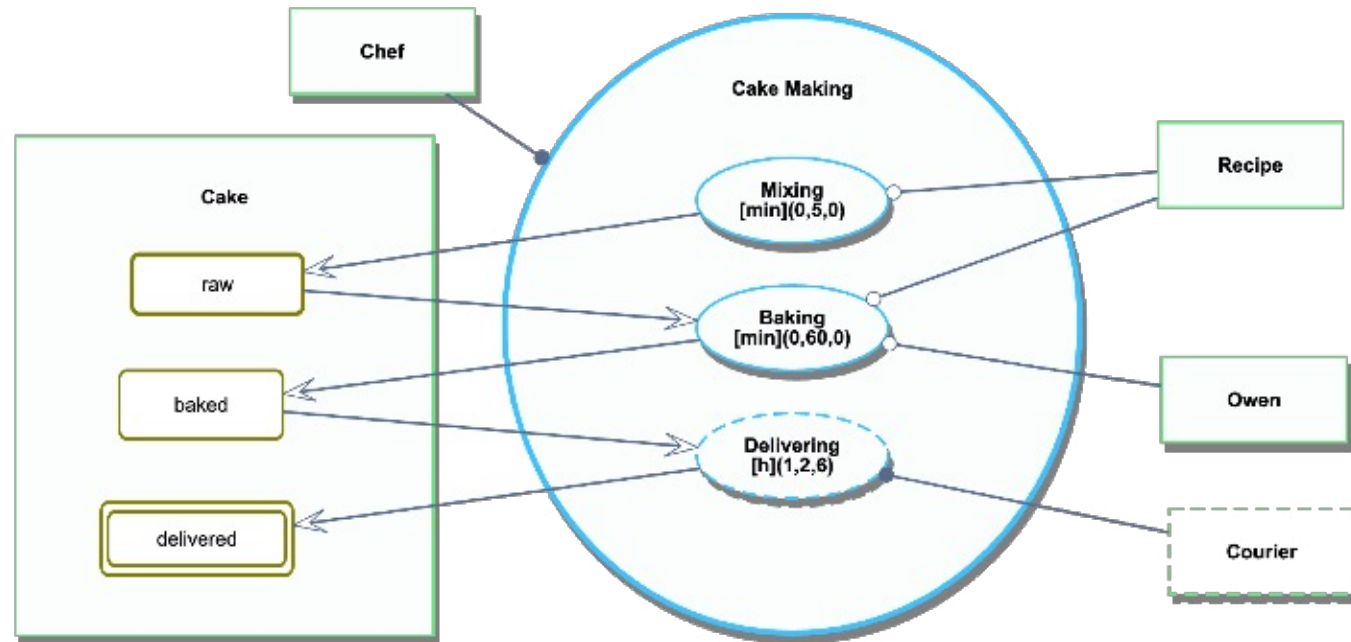
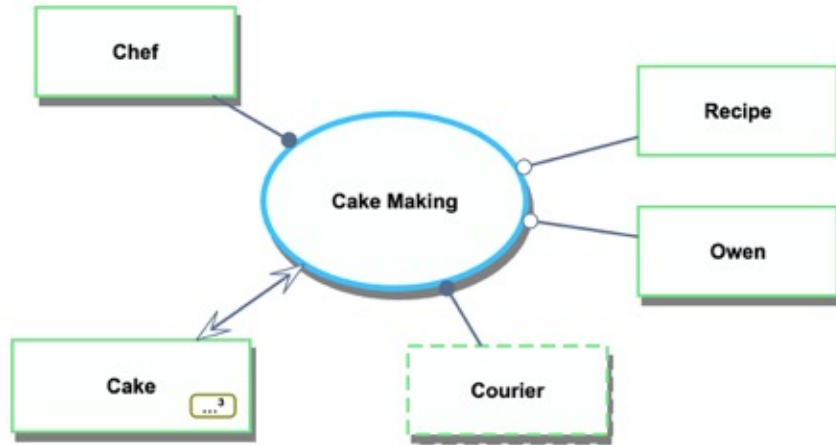
# OPD AND OPL EXAMPLE



1. **Cake Making** zooms into **Mixing**, **Baking**, and **Delivering**, which occur in that time sequence.
2. **Cake** is a physical and systemic object.
3. **Cake** can be **baked**, **delivered** or **raw**. State **raw** is initial. State **delivered** is final.
4. **Recipe** is an informatical and systemic object.
5. **Courier** is a physical and environmental object.
6. **Oven** is a physical and systemic object.
7. **Chef** is a physical and systemic object.
8. **Cake Making** is an physical and systemic process.
9. **Chef** handles **Cake Making**.
10. **Mixing** is an physical and systemic process.

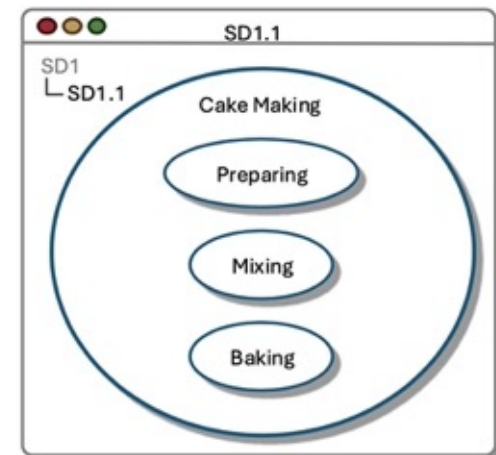
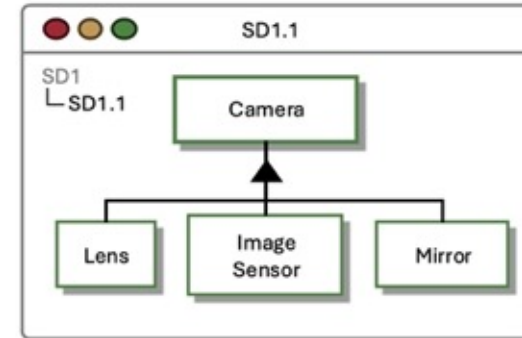
11. The expected duration of **Mixing** is 5 min.
12. **Mixing** requires **Recipe**.
13. **Mixing** yields **Cake** at state **raw**.
14. **Baking** is an physical and systemic process.
15. The expected duration of **Baking** is 60 min.
16. **Baking** changes **Cake** from **raw** to **baked**.
17. **Baking** requires **Oven** and **Recipe**.
18. **Delivering** is an physical and environmental process.
19. The minimal duration, expected duration, and maximal duration of **Delivering** are 1 h, 2 h, and 6 h, respectively.
20. **Delivering** changes **Cake** from **baked** to **delivered**.
21. **Courier** handles **Delivering**.

# OPD ZOOMING



# OPD COMPLEXITY MANAGEMENT

- ▶ OPM offers three complexity management mechanisms
  - ▶ **Unfolding/folding – structure of the system**
    - ▶ Links precedence: Aggregation-participation > Exhibition-characterisation > Generalisation-specialisation > Classification-instantiation
  - ▶ **In-zooming/out-zooming – behaviour (processes) or spatial layout (objects)**
    - ▶ Timeline from top to bottom, parallel = synchronous
    - ▶ In-zooming implies aggregation-participation or exhibition-characterisation between objects/process
    - ▶ Links are distributive
    - ▶ Links precedence: Consumption = Result > Effect > Agent > Instrument
  - ▶ **View creating – other perspectives**
    - ▶ E.g. Other structural hierarchies, functional allocation, requirements mapping, interface diagrams...



# OPD SIMULATION

The screenshot shows the OPM simulation interface for a 'Cake Making' process. The interface includes a top navigation bar with the OPM logo and user information (Vojtech Merunka, OptimISE). A central workspace displays a simulation diagram with a 'Chef' actor, a 'Cake' object, and a 'Cake Making' process. The 'Cake' object has states: 'raw', 'baked', and 'delivered'. The 'Cake Making' process consists of three sequential activities: 'Mixing [min](0,5,0)', 'Baking [min](0,60,0)', and 'Delivering [h](1,2,6)'. The 'Recipe' object is connected to the 'Mixing' activity, and the 'Owen' object is connected to the 'Baking' activity. The 'Courier' object is connected to the 'Delivering' activity. A control panel on the right allows for settings such as 'Headless Runner', 'Include Sub Models', and 'Reset Values Each Run', along with an 'Animation Speed' slider set to 50%. A left sidebar shows a 'Druggable OPM Things' list with 'Cake', 'Chef', 'Courier', and 'Owen'. A bottom panel displays the OPL (Object Process Language) description for the simulation.

**OPL**

1. Cake Making from SD zooms in SD1 into Mixing, Baking, and Delivering, which occur in that time sequence.
2. Cake is a physical and systemic object.
3. Cake can be baked, delivered or raw.  
State raw is initial.  
Cake is currently at state delivered.  
State delivered is final.
4. Recipe is an informatical and systemic object.

# ARISTOTLE'S PHILOSOPHY IN OPM

**The Four Causes of Aristotle form a philosophical framework for understanding what a system is, how it works, why it works, and what purpose it serves.**

## **1. Material Cause – What is it made of?**

OPM view: *Objects that are consumed or transformed by processes.*

example:

**Cake** starts in state **raw**, implying the presence of ingredients according to **Recipe**.

## **2. Formal Cause – What is its form or structure? (How it looks and is structured?)**

OPM view: *The structure of the OPD (what form, what structure).*

example:

**Cake** follows information according to **Recipe**.

## **3. Efficient Cause – What brings it into being? (Who or what causes it?)**

OPM view: *Objects connected to processes via agent and instrument links.*

example:

**Chef** handles **Cake Making**. **Owen** is an instrument of **Baking**.

## **4. Final Cause – What is its purpose?**

OPM view: *The intended final state of a systemic object.*

example:

**Cake** reaches the final state **delivered**.

# **BENEFITS OF OPM IN AI-SUPPORTED SYSTEM MODELING**

**OPM, through its bimodal representation of OPD and OPL, provides an explicit conceptual structure that effectively extends the benefits of LLMs while compensating for their inherent limitations.**

The LLM's capabilities are harnessed in a controlled and interpretable manner, while human conceptual judgment ensures verification, grounding, and architectural coherence — together mitigating hallucinations and reinforcing ontological consistency across the evolving system model.

**OPD** of the model allows human stakeholders and system engineers to verify and validate LLM-generated conceptual structures through visual inspection and simulation.

**OPL** of the same model, in turn, enables LLMs to operate within a grounded and semantically constrained conceptual frame, improving the precision and consistency of LLM-generated system descriptions as well as the estimation of **FP** and **COCOMO**.

**By integrating OPD for human oversight with OPL for LLM-based processing, it creates a hybrid neuro-conceptual metasystem that unifies the complementary strengths of human reasoning and AI-driven generation.**

# THANK YOU FOR YOUR ATTENTION

