
# Systemic Framework for Enterprise Architecture & Transformation

# Modularity

# Introduction

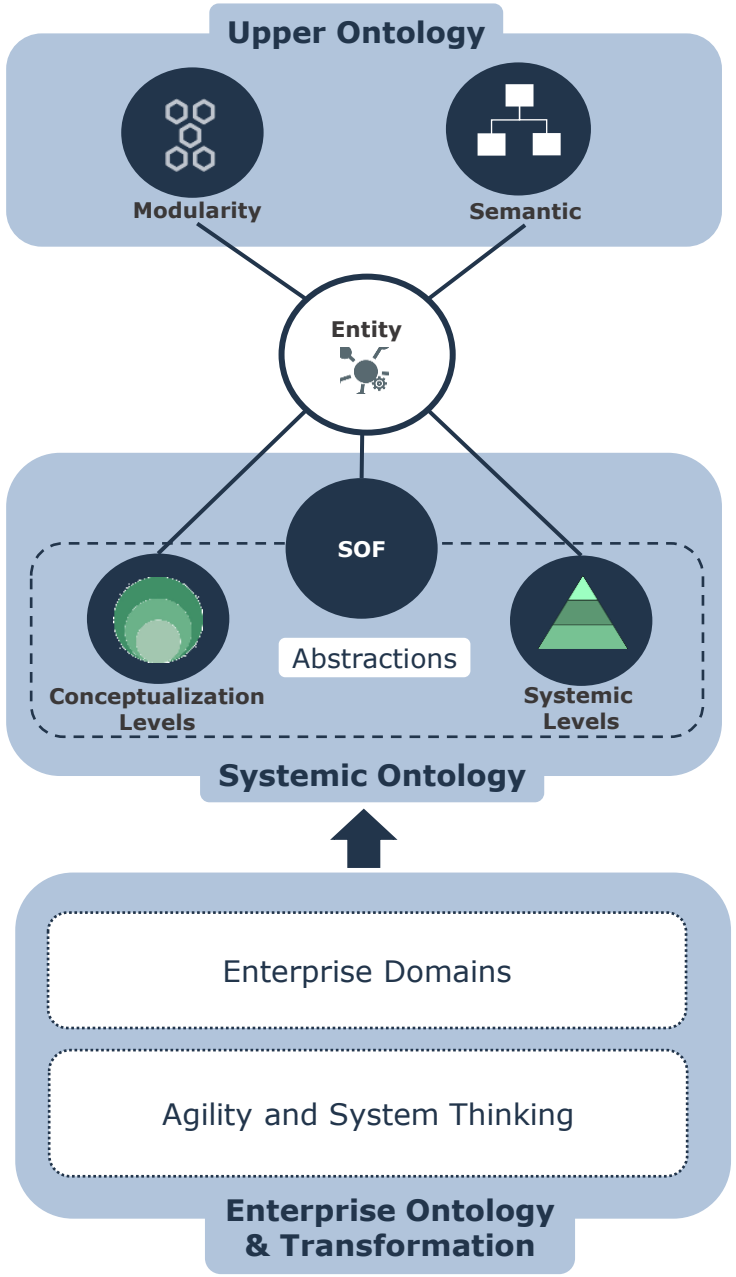
- This document is an integral component of the SysFEAT architectural framework. It provides foundations to address the challenges posed by Enterprise Architecture in the 21st century, which include :
  - Increasing complexity in system structures and behaviors.
  - Growing intricacy in architecture, management and governance of these systems.
  - The mission of the framework is to demystify these complexities, ensuring they are comprehensible to a broad audience, thereby facilitating the design and management of complex-systems across all scales, from micro-systems to enterprise level systems.

- Enterprise Modeling refers to the overarching language and conceptual framework used to describe, understand, and communicate the complex structures and dynamics of an enterprise.
- It integrates both the operating aspects of the enterprise (how it functions and interacts within its ecosystem), the transformational aspects (how it evolves and sustains over time through initiatives, asset management) and how these transformations are governed to ensure effectiveness, efficiency and reliability.

- The following slides present the foundations of enterprise modeling.

# Foundations of enterprise modeling

- **Modularity** provides the syntax for building robust, manageable, and scalable architectures, based on the principles of composability and packaging.
- **Semantic** provides robust capabilities for classifying and composing entities, from time-bound entities (individuals) to families of concepts, enabling effective representation of meaning.
- The **Systemic Operating Framework (SOF)** serves as the overarching language that describes why and how a system operates and interacts within its ecosystems.
- **Abstractions** organizes systems and concepts in degree of abstractions, including systemic levels and conceptualization levels.
- **Enterprise Domains** formalize the various disciplines that make-up EA, ranging from enterprise road-mapping to System ArcDevOps.
- **Agility and System Thinking** ensure that the enterprise evolves and sustains over time through governed initiatives, architected for flexibility and responsiveness in complex and dynamic business environments.



# Modularity & Modeling language

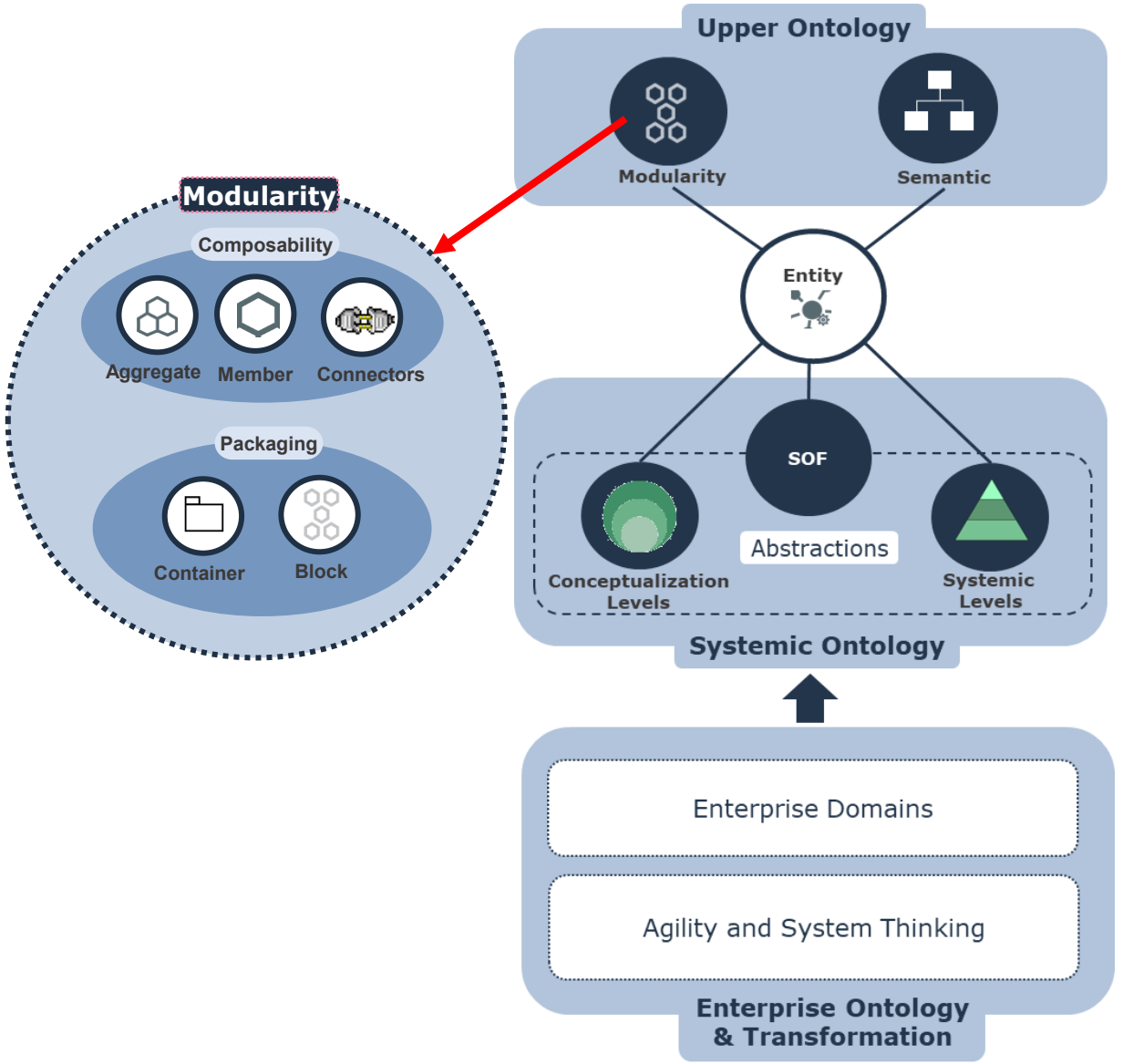
- As any language, modeling languages have three aspects:
  - Syntax is the required grammar and punctuation of the language.
  - Semantics is about meaning - what do we mean by a Capability?
  - The EA GRID is about semantic for Enterprise Architecture.
  - Pragmatics/Architecting has to do with:
    - How to use models (modeling technics).
    - What kind of model to use to address stakeholder concerns (method)
      - ✓ Example: how to use capability modeling in enterprise transformation initiatives.
- This document presents syntactic foundations for developing modular enterprise models.

# Modularity in the Architecture modeling landscape

- This document focuses on **modularity** at the **syntax level**, which is grounded on two complementary aspects: *composability* and *packaging*.

- **Composability** is the ability to assemble entities to form bigger constructs called aggregates.
  - Composability is a **syntactic** concern that does not carry inherent semantic meaning.
  - It can be applied to both semantic relationships of composition and typology.
- **Packaging** is the ability to group autonomous-reusable building blocks in **modules** also called Packages.

- These two disciplines come hands to hands but shall not be confused.



# Composability: Why?

*Complexity Is Good;*

*It Is Confusion That Is Bad*

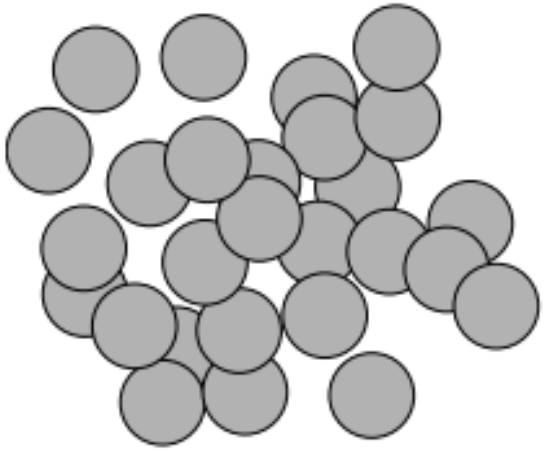
*Don Norman*

*The DESIGN of EVERYDAY THINGS*

# Modularity benefits – Don Norman illustration

- Source: Don Norman – Living with complexity - page 236
  - *Count the circles simply by looking at them: don't use your hands or a pointer to help. Difficult, isn't it?*

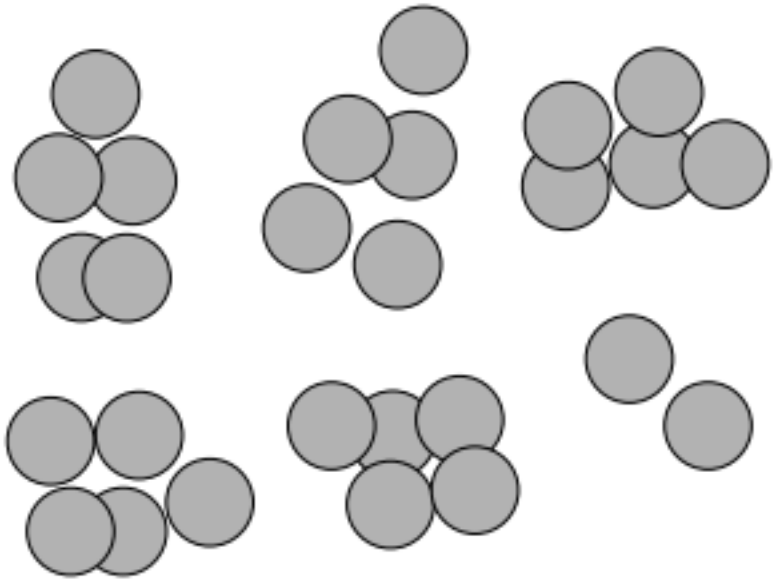
**Figure 8.3**



# Modularity benefits – Don Norman illustration

- Source: Don Norman – Living with complexity - page 236
  - *Now count the very same items shown in figure 8.4, again without using hands or other objects as aids: much easier, isn't it?*

**Figure 8.4**



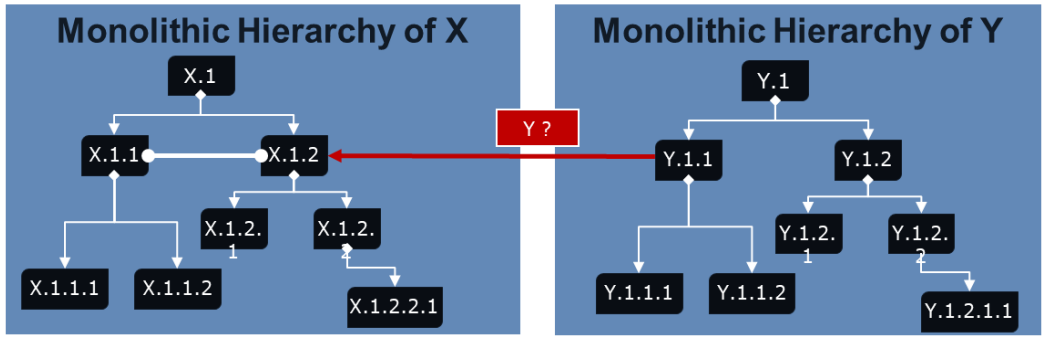
# Architecture & Composability

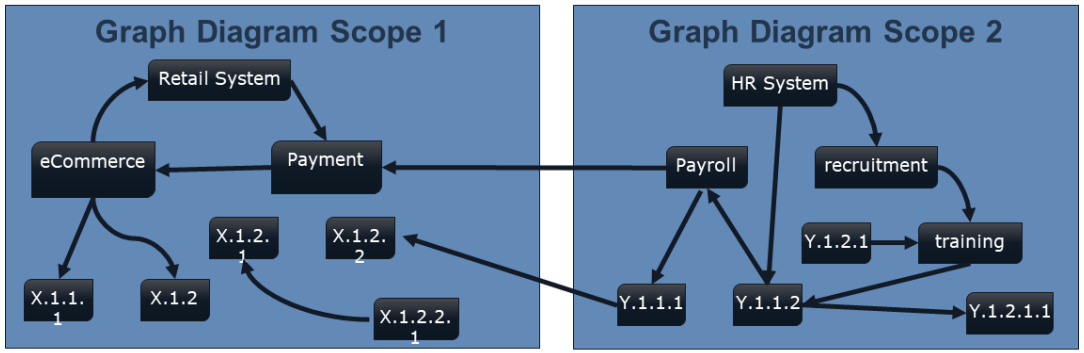
The issues of current frameworks

# What is the problem ?

- To provide value in driving transformations, architecture models must themselves be modular to deliver the following services:
  - Be able to build architecture alternatives.
  - Be able to manage catalogs/packages of reusable building blocks.
  - Be able to compare alternative architectures.
  - Be able to guide enterprise transformation, involving time and space perspectives.

- Problem:
  - The two common modeling syntaxes used for architecture descriptions – *monolithic hierarchies and flat models* - prevent from creating effective building block boundaries, thereby denying the notion of building block itself.
  - Without effective scoping principles, model-driven architecture cannot successfully help in designing complex adaptive systems while ensuring associated quality/security assurance.

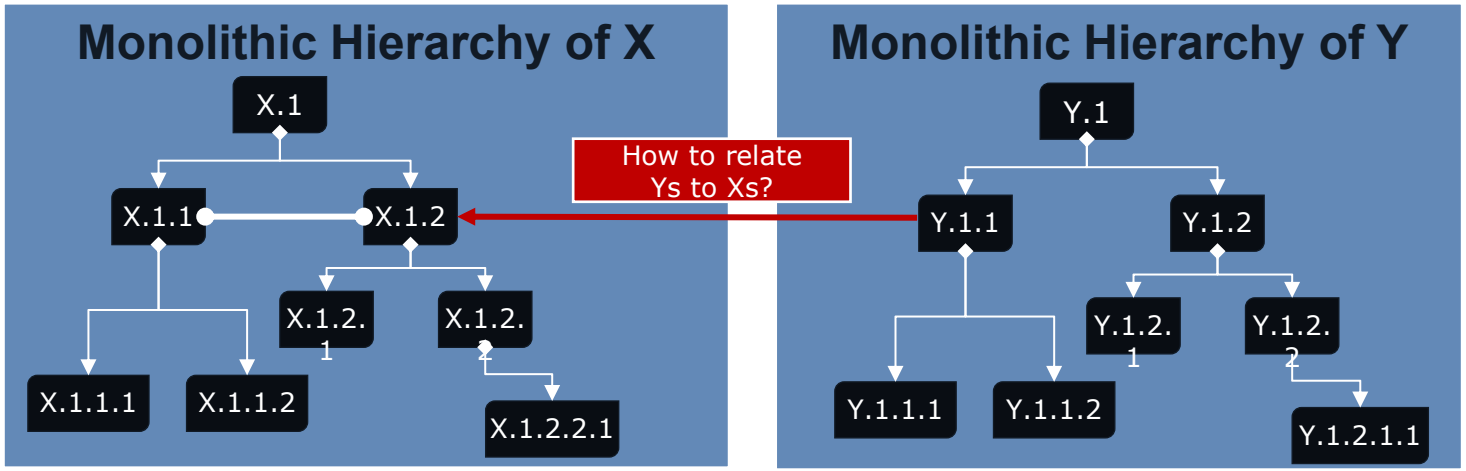


# Problem 1 : monolithic hierarchies & interconnections

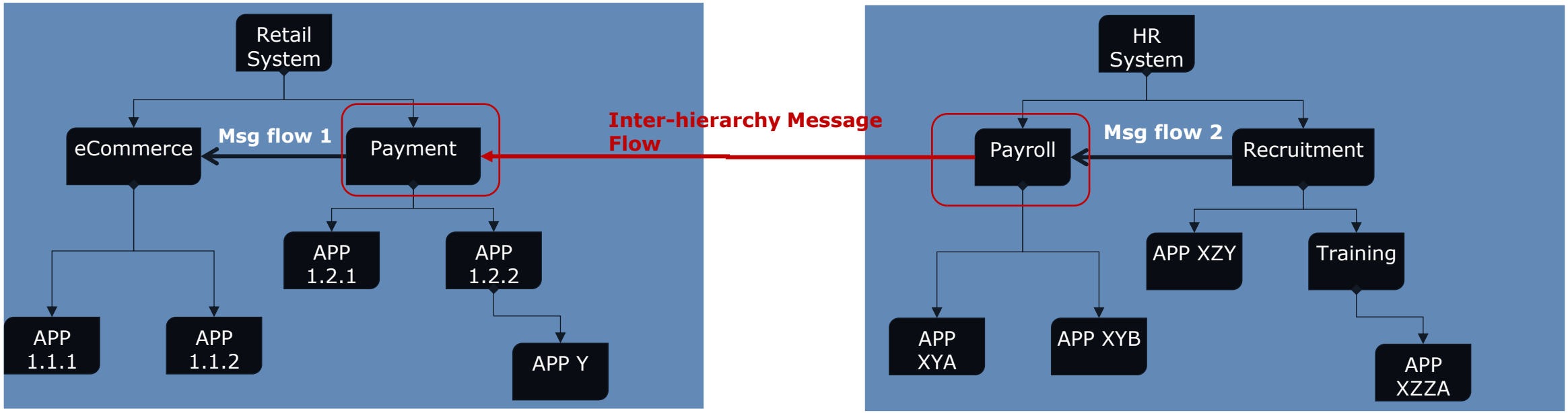
- Benefits of monolithic hierarchies.
  - They follow the usual breakdown practice (Cartesian approach).
  - They provide hierarchical scope for building blocks. This sometimes represented by naming conventions, such as, “X.1.1” and “X.1.2” are in “X.1”. IDEF notations are a good illustration of monolithic hierarchies.

- Issues: monolithic hierarchies hardwire building blocks together:
  - Blocks can only be part of  a single hierarchy: the single parent syndrome.
  - If multiple parent-relationship is allowed, inter-connections become undefined : the many to many relationship syndrome (see next slide).



# Illustration : monolithic hierarchies & interconnections

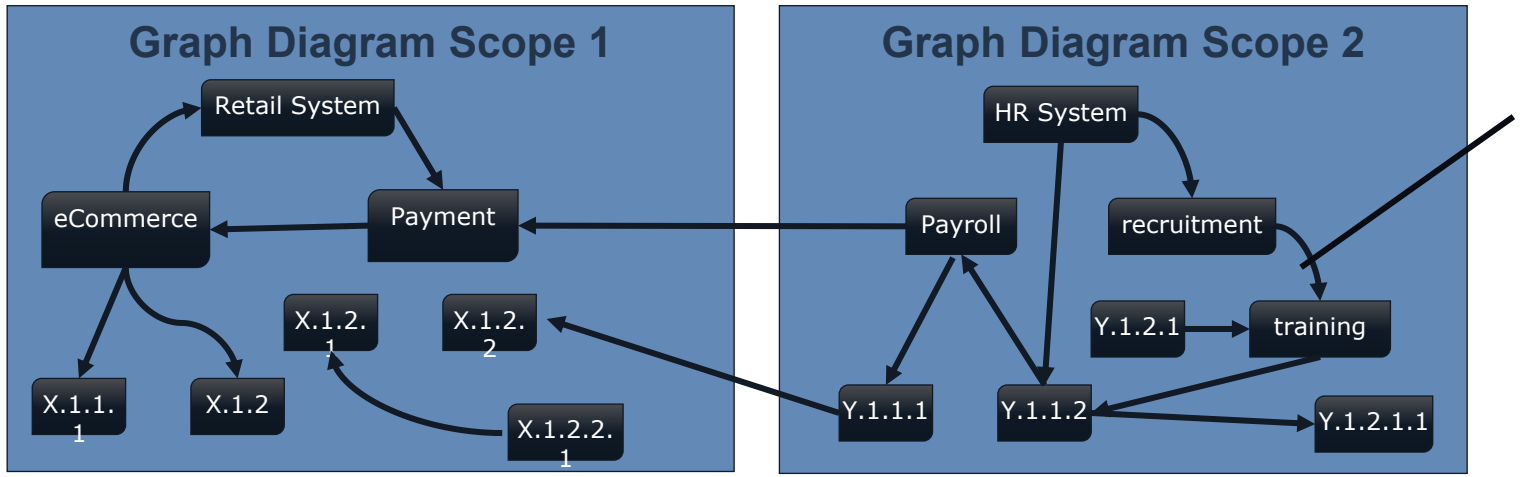
- Let’s consider two monolithic application hierarchies:
  - Retail-System (on the left below) and HR-System. (on the right below).
- If *Payroll* (from *HR System*) needs to send a message to *Payment* (from *Retail System*), does this implies that:
  - *Payment* becomes part of the *HR-System* hierarchy ?
  - or that The *HR-System* depends on the *Retail-System* (cross hierarchy dependency)?
- Similar issues occur on sequences between processes, flows between processes, etc. Strict hierarchical scope prevent from having reusable, autonomous building blocks.
- The benefit of autonomous monolithic hierarchies is lost because of the need to connect multiple hierarchies.



# Problem 2 : flat models – non-local relationships

- Benefits of flat E/R models:
  - They avoid the single parent, single scope syndrome of monolithic block hierarchies.
  - They enable a natural discovery of unitary building blocks and their dependencies through story boards. For instance, may architects look at  messages between software systems or events and commands through event storming.

- Issues:
  - Building Block assemblies have been lost: there are no more scoped relationships but a single global graph (*is Payment in HR system?*).
  - Adding a relationship at one end of the graph has undefined effects on the rest of the graph, hence building blocks do not have an autonomous definition.
  - Diagrams are often used for creating pseudo system boundaries. As mere pictures, diagrams do not provide an explicit definition of system-boundaries. We are back to Visio (See ArchiMate below)!

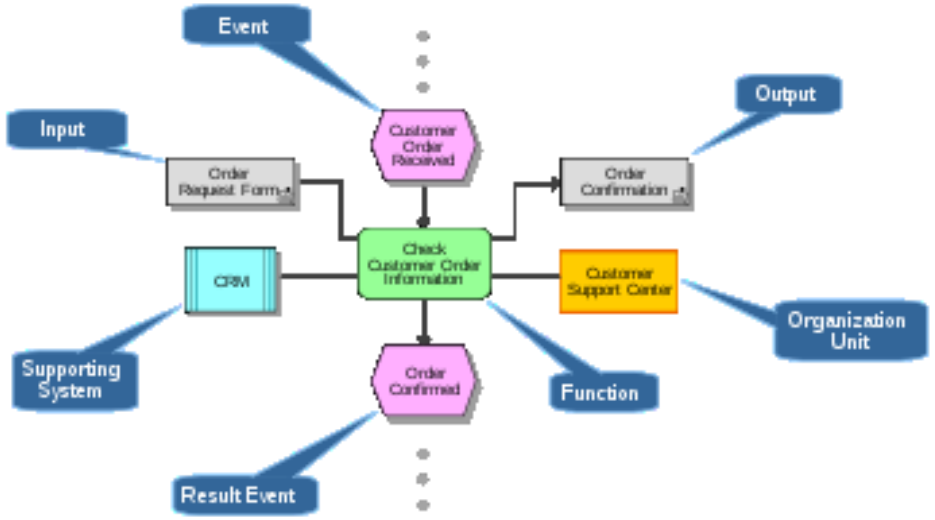


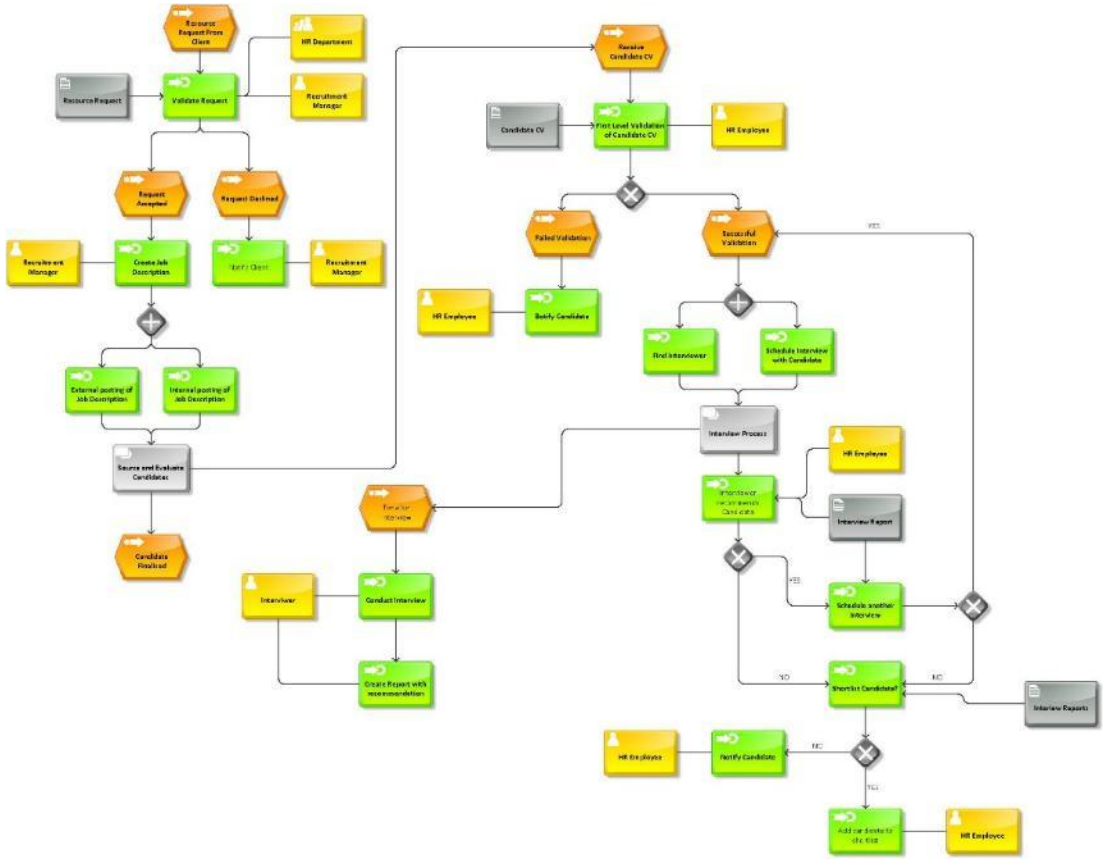

What is the impact of adding this connector ?
- Is *Recruitment* changed ?
- Is *Training* changed ?
- both ?

What is the change impact scope ? Recruitment, HR System, … the entire graph ?

# Problem 2: flat models – the case of EPC models

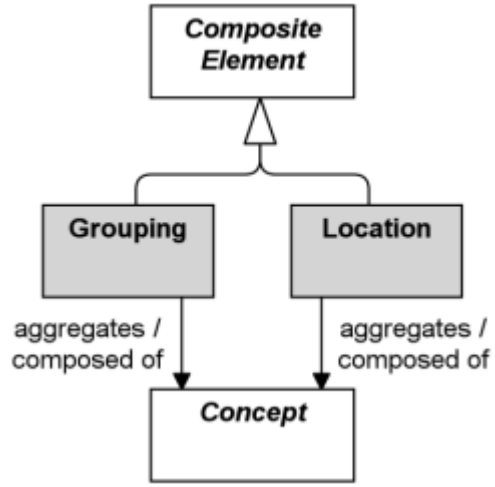
- EPC Models - Extract from Wikipedia (2008):
  - *<..>Unfortunately, neither the syntax nor the semantics of EPC are well-defined.[1] EPC requires non-local semantics,[2] so that the meaning of any portion of the diagram may depend on other portions arbitrarily far away.<..>*

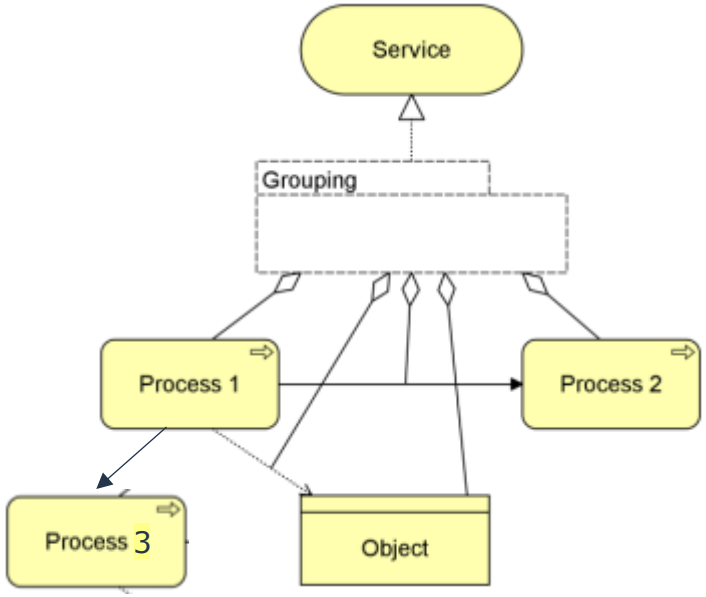


# Problem 2: flat models – the case of ArchiMate

- ArchiMate faces similar needs. It has introduced an intuitive mechanism to contextualize relationships, called *grouping* - with the following definition:
  - *The grouping element is used to group **an arbitrary group of concepts** (elements and/or relationships), which can be of the same type or of different types. The aggregation relationship is used to link the grouping element to the grouped concepts.*
- "Arbitrary" means fuzziness. It leads to an unlimited variety of interpretations in the hand of "ArchiMate Gurus". The issue becomes even trickier when it comes to have relationships between groups.
- The diagram on the right reveals the issue of non-local relationships:
  - The relationships between *Process 1*, *Process 2* and *Object* need to be contextualized for the relationship with *Service* to be established, excluding the relationship from *Process-1* to *Process-3*.



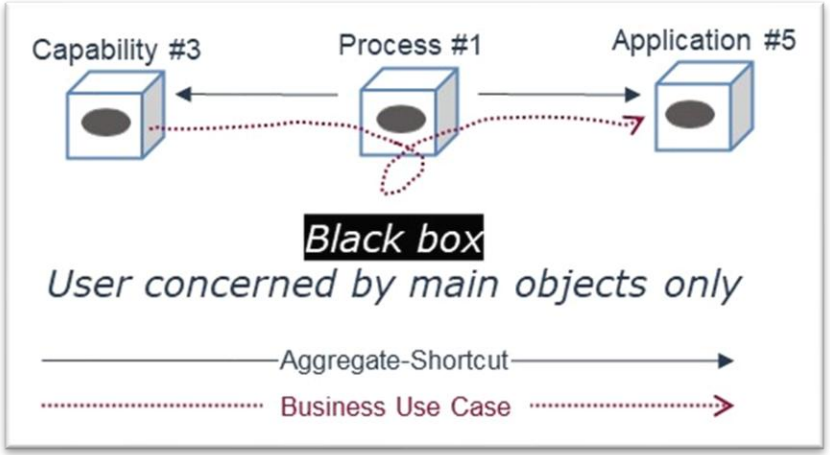
# Composability Principles

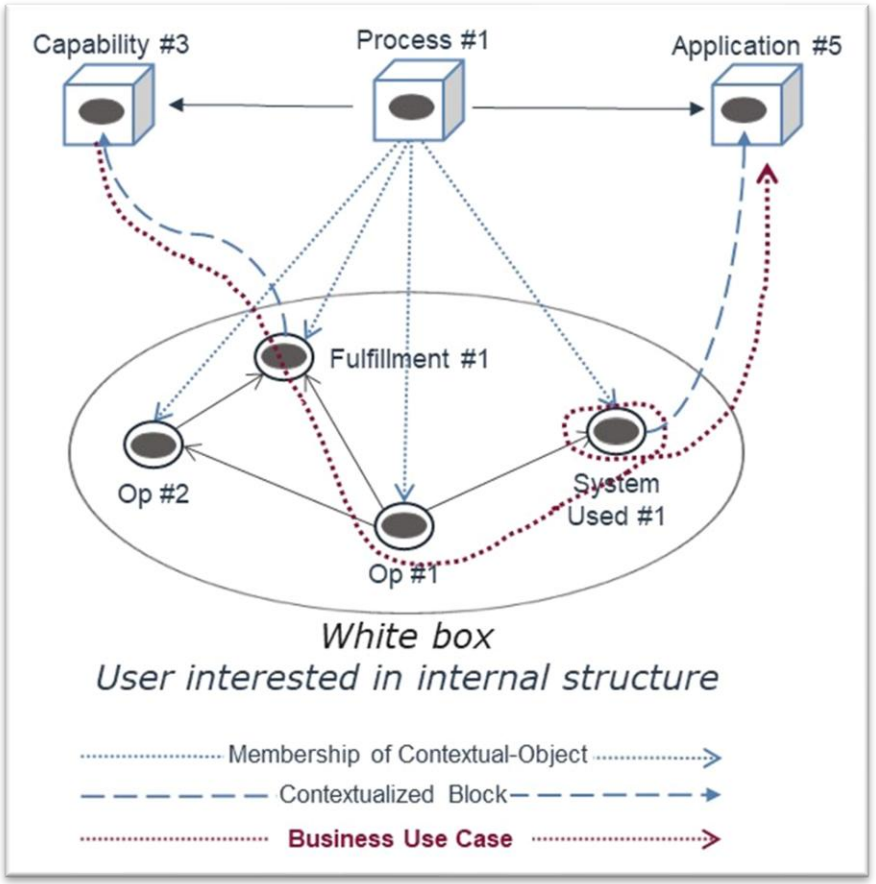
# Aggregates

- **Aggregates** are business level entities, with an **internal structure** made of aggregate members
- Depending on the working context, users are sometimes interested only in the main entity (Black box), and some other times in the internal structure (White box)
- Typically, Applications, Processes and Capabilities are aggregates.

- Aggregates are building blocks, they can be of different kinds, but the fundamentals remain the same; the list of aggregates & aggregate members is provided in the advanced detailed Vocabulary presentation

- Remark: Aggregate is a term defined in **standard Domain Driven Design approach** [DDD]



# WARNING

## COMPOSABILITY VERSUS COMPOSITION

- Composability is a **syntactic** concept that lacks inherent semantic meaning. It is an ability to assemble entities to form bigger constructs called aggregates, but it does not define the nature of their relationships.
- Composability can be applied to various semantic relationships, such as composition or specialization (e.g., "generalization" in UML).
- For more details on composition, please look at the presentation on semantic.

- A frequent misconception is conflating nesting, composability, and composition. While these concepts may intersect, they address distinct aspects of system design and should not be confused.

# Overview of composability principles

- Composability is the ability of constructing complex entities (Aggregates) by local assembly of related entities, creating a unified composite that exhibits emergent properties—qualities that surpass the simple sum of its individual parts.
- At its core, composability bridges two key perspectives: hierarchies (nesting of related entities) and networks (connections between Bounded Aggregates). To achieve effective composability, four essential characteristics must be present:
  - Reified Relationships: the transformation of a binary relationship between a source and a target entity into a distinct entity called an Aggregate Member, which represents the relationship itself, embedded within its source.
  - Bounded Aggregates : some aggregates encapsulate their internal structure behind a boundary, ensuring clarity and modularity.
  - Boundaries:  connection points that express as an ability to connect, in accordance with connection definitions: Connection Entities.
  - Connection Entities : these new kind of entities defines connectivity relationships between assembled aggregates, serving as the glue that binds them together. Examples include Events and Service Interfaces.

# Aggregate Blocks & Contextualization (Aggregation)

- An Aggregate is an Entity Element which has an internal structure made of Aggregate Members :
  - Typically, attributes of a class, steps of a process are aggregating members.
  - Agents, Processes, Capabilities, Data Objects are aggregates.

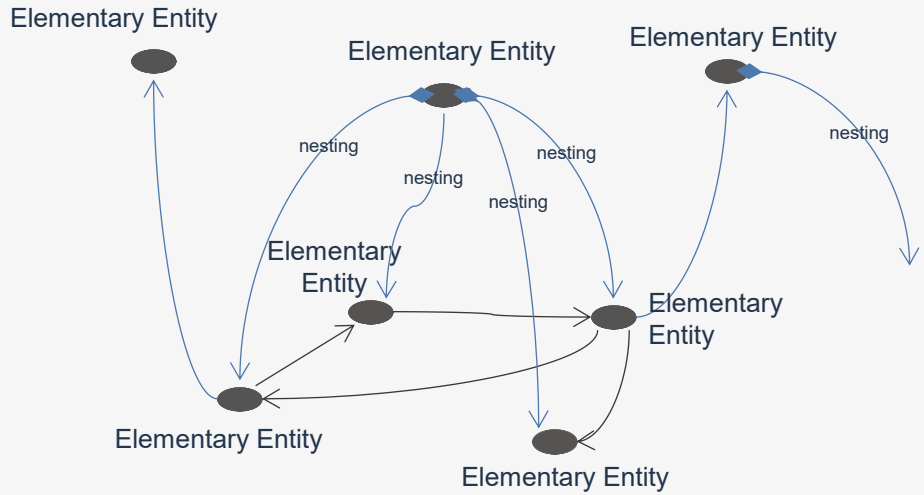
- Aggregation allows defining characteristics for the aggregated Entity Element that only apply within its parent Aggregate. This enables expressing the emergent properties of the composite structure.

*NOTE: The term "Aggregate" originates from the widely recognized Domain-Driven Design (DDD) framework.*

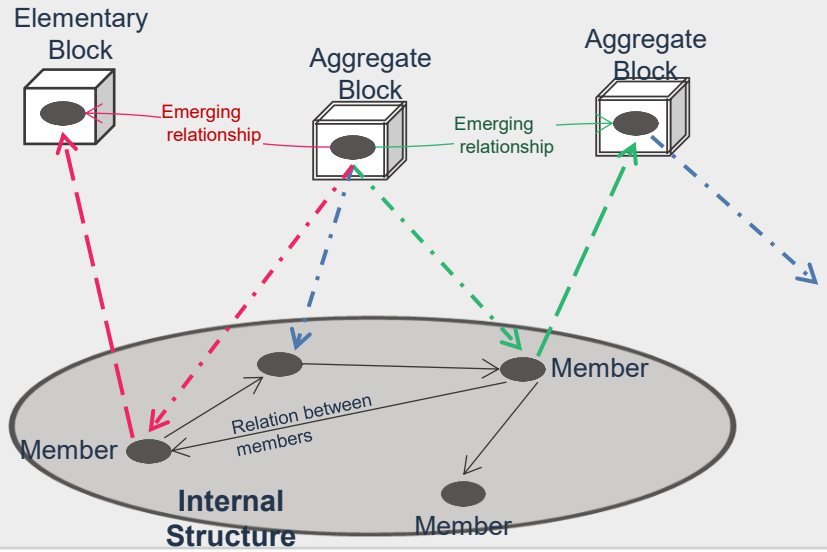
# Composability & layered graphs

- Composability induces a new method for structuring entities and relationships, replacing the conventional Entity/Relationship model with a dual-layered graph framework.
  - The Elementary graph is a traditional directed graph that includes nesting relationships.
  - The secondary graph employs nesting relationships as a mechanism to delineate the internal structure of aggregates and infer emerging relationships between them.

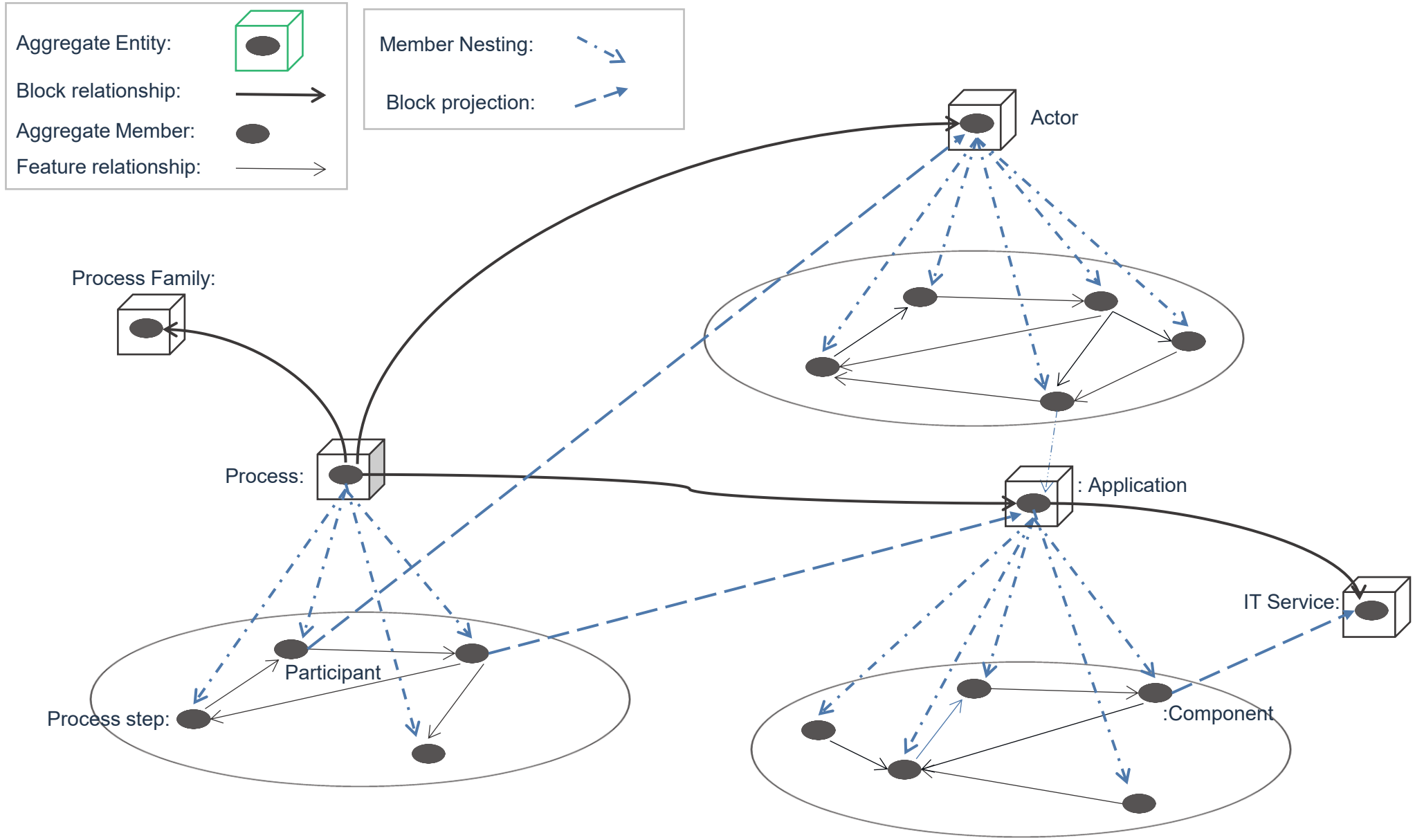
**Elementary Graph**




**Secondary Graph**

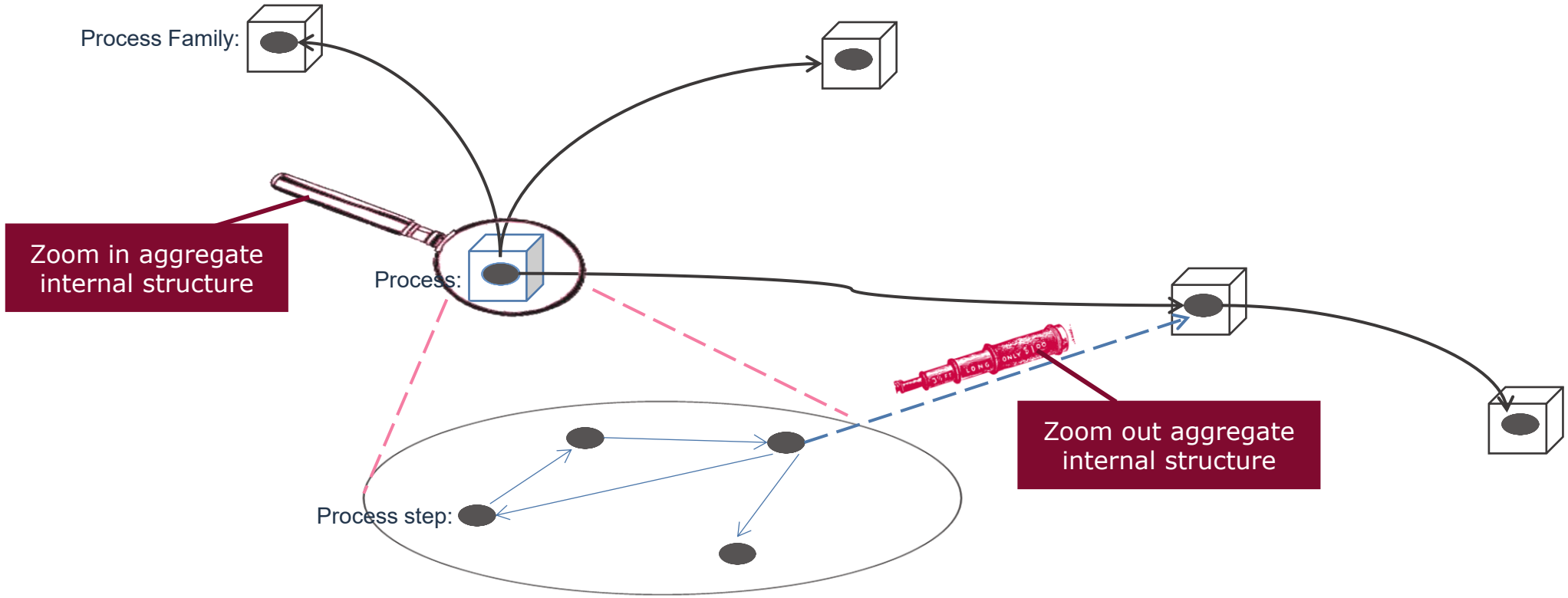


# Secondary Graph illustration - BPMN



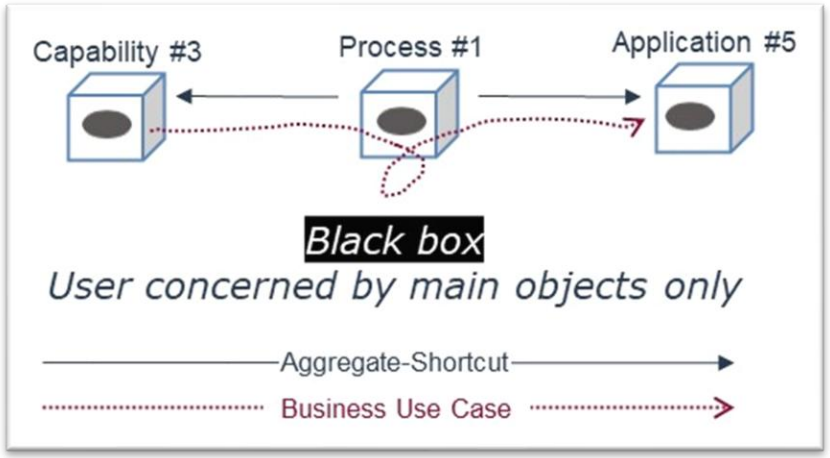
# Layered graphs and navigation

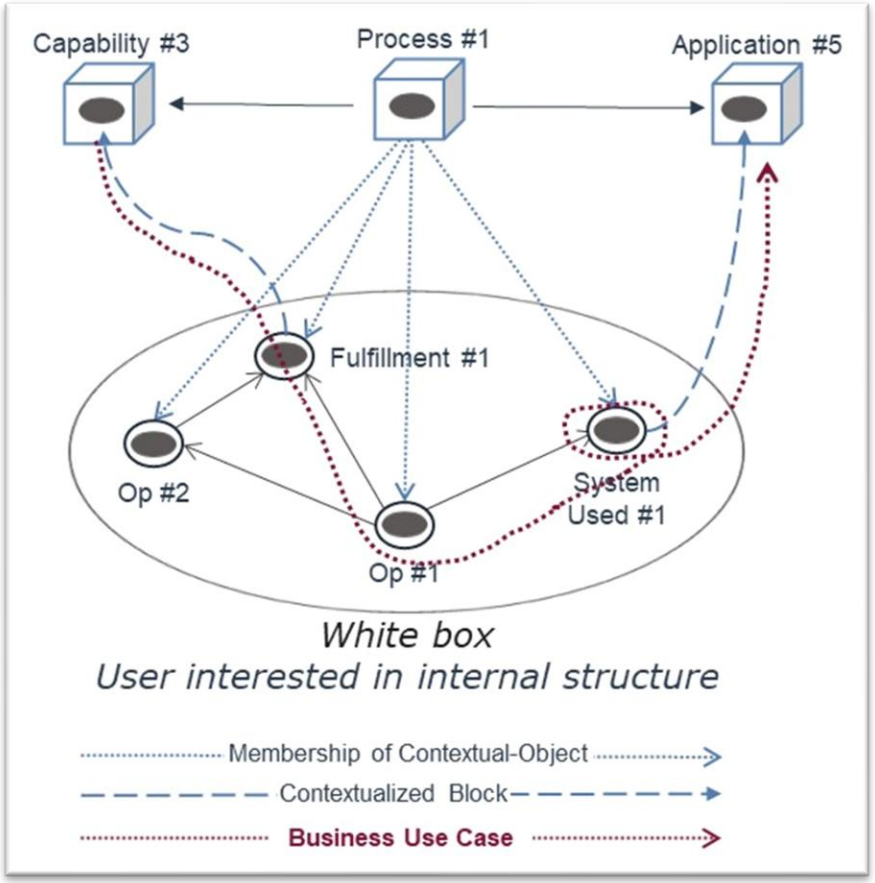
- Depending on the context of their work, users may prioritize either the external aspects of aggregates (Black box) or delve into their internal structure (White box) at different times.“
- Structural zooming is a novel approach of graph navigation to address these needs.
  - It integrates zoom, fusion, or morphing to enable users to scrutinize aggregate details while maintaining visibility of higher-level aggregates and their interconnections.
  - It encompasses two pivotal functions: zoom-in, which delves into the internal structure of aggregates, and zoom-out, which ascends to reveal direct relationships between aggregates.



# Aggregate & Views

- Emerging relationships should not be confused with relationship paths required to build view.
- View are structures of aggregates build to respond to specific processing of portions of concepts graphs.



# Packaging & Modules

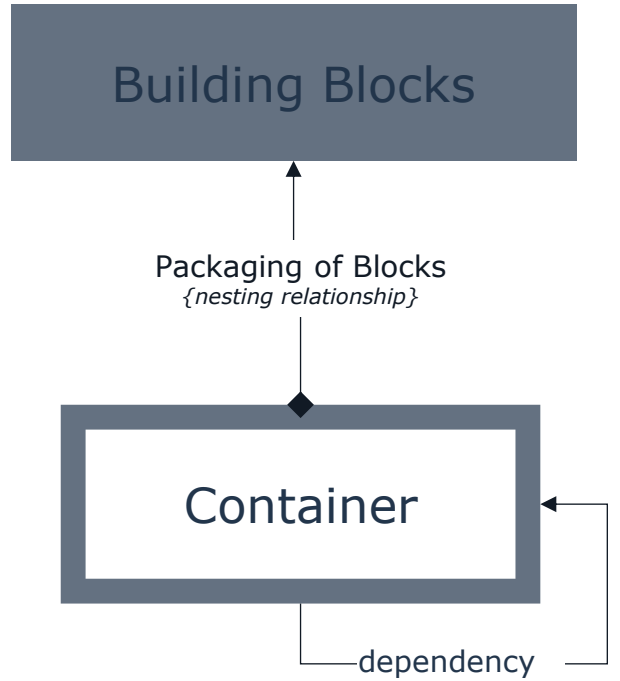
# Packaging

- Packaging is the ability to group building blocks into modules commonly referred to as "packages" or "packages" in software engineering. The general concept for packages in the concept of Container.
- Containers are the means to group and version the different components of a system.
  - In the case of software, this components comprise codes, data, configurations
  - In the case of models, this components comprise aggregate building blocks and their connections.
- Containers dependencies must be identified and mastered to automate the delivery of modules and ensure the deployment processes in different environments.
- Containers must be tested, built, and versioned to be ready for continuous deployment.

- Packaging is an essential aspect of incremental delivery and a key foundation for modular enterprise modeling and architecting.

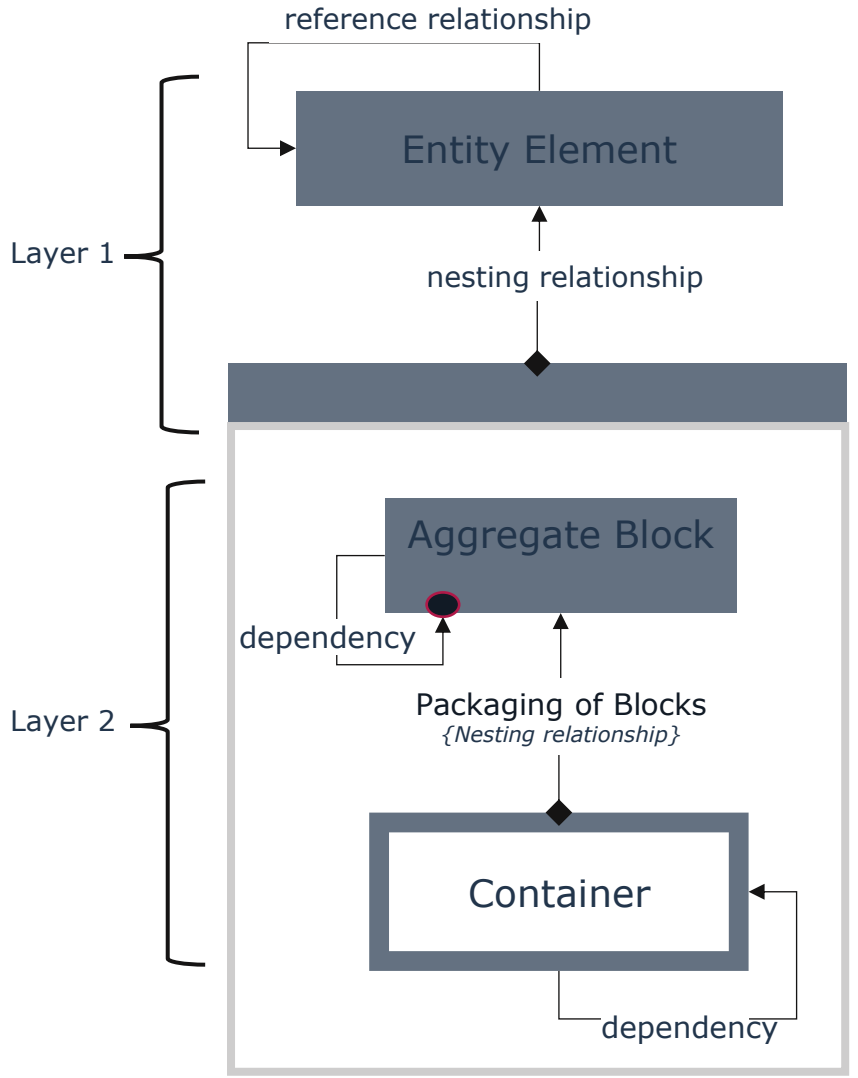
# Building Blocks & Containers

- As reusable units, Building Blocks have an independent existence. Thus, they cannot be nested into other Blocks that would hide their existence.
- Because of their independent existence, they must belong to an independent artifact whose sole purpose is the modular management of building blocks: containers.
- Containers are dedicated to the modular management of building blocks:
  - They hold the building blocks to which they provide existence.
  - They can provide a namespace for building blocks.
  - They potentially have dependencies on other containers (see next slide).
- Versioning of modules in EA is a separate concern that will be addressed in a future section on building block management.



# Summary of the dependency stack

- Dependencies are built on a stack of two nesting layers:
  1. Ownership of relationships and model elements
     1. Relationship must be directed (ownership of relationships)
     2. Model elements must be owned (nesting relationship) by a cluster of model elements (process owns operation, library owns process).
  2. Clusters & dependencies
     1. Clusters are either:
        1. building blocks: reusable, independent clusters (process owns operation).
        2. Containers: owner of building blocks (library owns process).
     2. dependencies between building blocks creates dependencies between modules.



# Container dependencies

- Dependencies between building blocks (red arrows below) result in inferred dependencies between modules that package them.
- Dependencies between modules must be made explicit through a **dependency analysis process**.