



Model-Driven Software Engineering: An NST Architecture

Normalized Systems White Paper

June 12, 2026 | Herwig Mannaert



Contents

1	Introduction	2
2	Model-Driven Software Engineering	2
2.1	Code Generation and Model-Driven Development	2
2.2	Expansion Based on Normalized Systems Elements	2
2.3	The Need for Flexible Modeling Frameworks	3
3	Multilevel Meta-modeling and Ontologies	4
3.1	Meta-Object Facility for Meta-modeling	4
3.2	Integrating Ontologies with Meta-modeling	6
4	The NST Multi-level Modeling Architecture	7
4.1	Embedding NS Expansion in MOF	7
4.1.1	A Lightweight Meta-meta-model	8
4.1.2	The NS Elements Meta-ontology	8
4.2	Empowering Meta-model Ontologies	8
4.2.1	Extending and Creating Meta-models	8
4.2.2	Supporting Meta-models Downward	8
4.3	The Role of of Domain Ontologies	9
4.4	Integrating Modeling Languages	9
5	Conclusions	9

1 Introduction

2 Model-Driven Software Engineering

2.1 Code Generation and Model-Driven Development

The act of automatically generating source code from a model or template is nearly as old as programming itself. While often called *automatic programming*, it is also referred to a *generative* or *meta-programming*, and sometimes distinguished from code generation, as performed by a compiler. The fact that automatic programming *has always been a euphemism for programming in a higher-level language than was then available to the programmer*, as stated by Dave Parnas [1], already implies its model-based nature.

While the main goal is and has always been to improve programmer productivity, automatic programming entails multiple benefits. More generally, the purpose is to manufacture software – or entire software components – in an automated way, in the same way as manufacturing automation in the industrial revolution. Therefore, the envisioned advantages are:

- to increase programming productivity
- to consolidate programming knowledge
- to eliminate human programming errors

The automatic generation of source code is in general based on the combination of a source template with parameters. A simple example would be a template for a *Data Transfer Object (DTO)* class, where the parameters consist of the name of the data entity, e.g., *Invoice*, and the various names and types of the attributes, e.g., an *Invoice number* and *reference*. Therefore, the field of automatic programming has become inextricably linked to the use of models for software development, i.e., model-driven development. This area of software engineering is known through various names and/or trends like:

- Model-Driven Architecture (MDA)
- Model-Driven Engineering (MDE)
- Model-Driven Software Development (MDSD)
- Low-Code Development Platforms (LCDP)
- No-Code Development Platforms (NCDP)

Despite the variety of names, it is important to keep in mind that these various trends and disciplines all share the use of models to structure requirements and/or to represent domain knowledge.

Recently, *AI (Assisted) Coding or Code Generation* has stormed the software development scene. Ranging from so-called *vibe coding* to full agentic AI coding, results have been impressive, but have clearly yet to reach the state of fully autonomous software development. We have argued in a previous *White Paper* [2] that AI-Based software development needs additional structured guardrails to control the sheer infinite amount of possibilities. The use of models, to establish descriptions or specifications in intermediate or higher-level languages, is clearly an example of such structured guardrails.

2.2 Expansion Based on Normalized Systems Elements

Normalized Systems Theory (NST) is a theory on modular structures under change [3]. Its operationalization in the context of information systems consists of two parts. First, four necessary – but not necessarily sufficient – **principles** are derived to avoid so-called *combinatorial effects* in software, i.e., effects

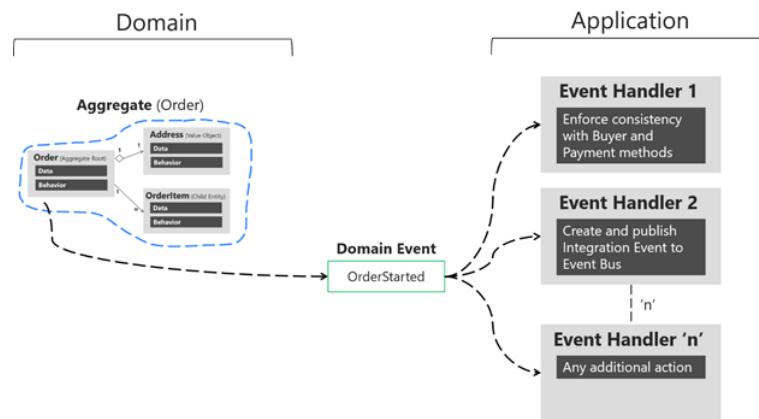


Figure 1: Representation of *Domain-Driven Design* modeling concepts and their relationships [4].

where the impact of a change ripples through the code structure and becomes dependent on the size of the system. Second, a number of so-called **elements** are proposed, detailed design patterns that can be instantiated to serve as the basic building blocks of information systems.

For these detailed design patterns, i.e., the element structures, it is proven that they mitigate these combinatorial effects, thereby providing systems theoretic stability with respect to basic changes [3]. The element structures also serve to interconnect the functional primitives, such as data entities and tasks, with the various technologies for the so-called *cross-cutting concerns*, such as persistency and data access. More specifically, NST defines five types of elements, aligned with the basic software programming concepts, to provide the basic functionality of information systems.

- *Data elements*, to represent data variables and structures, e.g., an *Invoice* with attributes.
- *Task elements*, to represent instructions and/or functions, e.g., a *PeppolInvoiceGenerator*.
- *Flow elements*, to handle control flow and orchestrations, e.g., an *InvoicingFlow*.
- *Connector elements*, to allow for input/output commands, e.g., an *InvoiceImporter*.
- *Trigger elements*, to offer periodic clock-like processing control, e.g., an *InvoicingEngine*.

To provide consistency and productivity, and to become less prone to human errors, code generation techniques are used to create instances of these recurrent element structures. Due to its simple and deterministic nature, NST refers to this process as *expansion*, and to the modular code generators as expanders. This expansion process of NST elements, i.e., *the parameter-based instantiation of detailed design patterns*, is clearly a form of model-driven development.

2.3 The Need for Flexible Modeling Frameworks

Though NST information systems, based on models that consist of NST elements, exhibit systems theoretic stability with respect to basic changes, many software designers might – and do – prefer other kinds of models for software systems. And while most programming models somewhat related to NST models, as they typically use data models and workflow process models, alternative types of models and modeling techniques exist as well.

In his seminal book on *Domain-Driven Design*, Eric Evans proposes to bring the modeling of information systems closer to the users or domain experts by identifying so-called *domain objects* [4]. As represented schematically in Figure 1 [4], these models are based on other types of primitive building blocks, like:

- *Domain Objects*, e.g., *Order*.

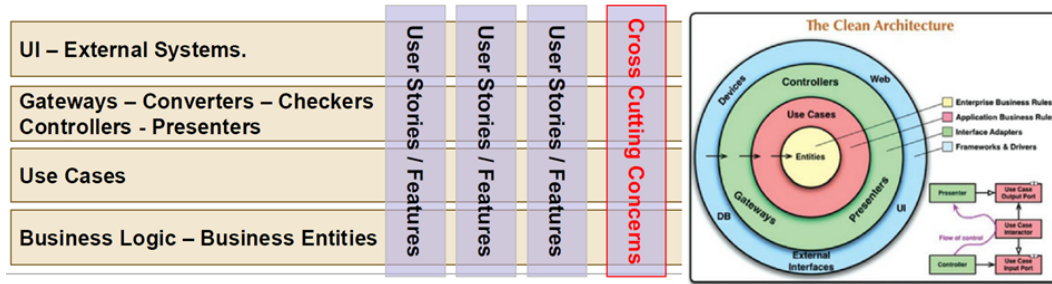


Figure 2: Schematic representation of the *Clean Architecture* as proposed by Robert Martin [5].

- *Aggregates*, e.g., an *Order* with *OrderItem* and *Address*.
- *Domain Events*, e.g., *OrderStarted*.
- *Event Handlers*, e.g., *Enforce Consistency*.

In his well-known books, Robert Martin investigates the creation of so-called *clean code* to obtain systems that are better structured and more easily maintainable. Martin does not only present *five principles* to guide the software engineer to develop clean code, but also proposes an entire *clean architecture* for the development of information systems [5]. As can be seen in the schematic representation in Figure 2, the high-level architecture distinguishes *horizontal layers*, *vertical slices*, and *cross-cutting concerns*. Martin stresses the decoupling, for instance of the horizontal layers:

- *Business Entities*, similar to DDD, and *Business Logic*, or the rules what the business allows to do.
- *Use cases*, representing workflows and sequences of business logic.
- *Gateways*, *Converters*, *Checkers*, *Controllers*, and *Presenters* to organize the processing and pre-processing in a systematic and structured way.
- *User Interfaces* to visualize, and *External Systems* to cut dependencies with these external systems and the various technologies.

The vertical slices, i.e., the user stories and features, run orthogonal through these horizontal layers.

3 Multilevel Meta-modeling and Ontologies

3.1 Meta-Object Facility for Meta-modeling

As explained in Section 2, the use model-driven development implies a modeling framework that defines the different building blocks or primitives of the model. Using the NST framework, we have to specify for instance data elements and task elements, while the use of a domain-driven design framework implies the definition of domain objects, aggregates, and events. Adhering to the clean architecture of Robert Martin on the other hand, would require the identification of business entities, business rules and workflows, gateways, converters, controllers, etcetera.

To be able to specify the appropriate set of model primitives, we need to specify what these primitives are. This corresponds to the specification of the so-called *meta-model*, i.e., the model of the model. Besides the primitives or concepts, such a meta-model needs to define the rules, syntax, and constraints of the modeling framework or language, and to establish the "how-to" guidelines for creating the actual models. In addition, we also need to be able to *represent* these meta-models and their specification. As argued by Herbert Simon in his seminal book on *The Sciences of the Artificial*, solving this so-called *representation problem* is both a necessary and non-trivial issue.

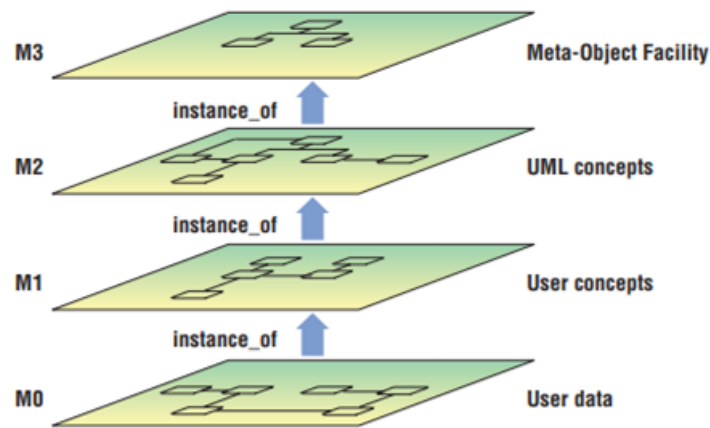


Figure 3: Schematic representations of *Meta-Object Facility* multi-level meta-modeling [10].

In their seminal paper, Jean Bézivin and Richard Lemesle investigated model-driven engineering principles, proposing a foundational, multi-level meta-modeling approach [7]. Bézivin and Lemesle outline how *Object-Oriented Analysis and Design (OA&D) models* can be rigorously defined using a layered architecture where the relations between the layers are based on instantiation. Addressing the issue of how many layers are needed, the authors propose a four-layer architecture to organize models, metamodels, and meta-metamodels, laying early groundwork for *Model-Driven Architecture (MDA)*.

To foster these *multi-level meta-modeling* environments, the *Object Management Group (OMG)* established the *Meta-Object Facility (MOF)* standard [8]. As represented schematically in Figure 3, and in accordance with the layers proposed in [7], this architecture distinguishes four modeling levels or layers.

- **M3** – *Meta-meta-model*, the language to describe meta-models.
- **M2** – *Meta-model*, the language to describe models.
- **M1** – *Model*, any entity of a type defined in the meta-model.
- **M0** – *Data*, any instance of an entity defined in the model.

The lower layers are the realm of model-driven software developers, and contain the *user concepts* in *M1*, e.g., an *Invoice*, and the *user data* in *M0*, e.g., *Invoice-001*. Meta-models, languages to describe models, are located at layer *M2*. This is the layer of the modeling languages, and the OMG explicitly includes the standardized and widely used *Unified Modeling Language (UML)*. This is also the layer where the data and task elements of NST, the domain objects and events of DDD, and the gateways and controllers of *clean architecture* reside. We have not included UML in Section 2, because it is in general limited to the realm of modelers, without being used to drive code generation. This detachment of modeling languages from code – and code generation – is both prominent and problematic in our opinion.

The meta-meta-model, i.e., the language to describe meta-models, is located at layer *M3*, and called the *Meta-Object Facility* [8]. This level requires a language to specify the *meta-entities* and *meta-relationships*. Moreover, such a language needs to be bootstrapped or *meta-circular*, i.e., able to be defined in itself. Otherwise, another layer would be needed to define this meta-meta-model language.

The MOF serves as the foundation for implementations like the *Eclipse Modeling Framework (EMF)* and is aligned with standards like UML and XMI (*XML Metadata Interchange*) for model interchange. The EMF *M3* or meta-meta-model, the highest level of abstraction in the Eclipse Modeling Framework (EMF), is called the *Ecore*. Ecore defines the constructs used to create meta-models at *M2* level. Its key elements are the following.

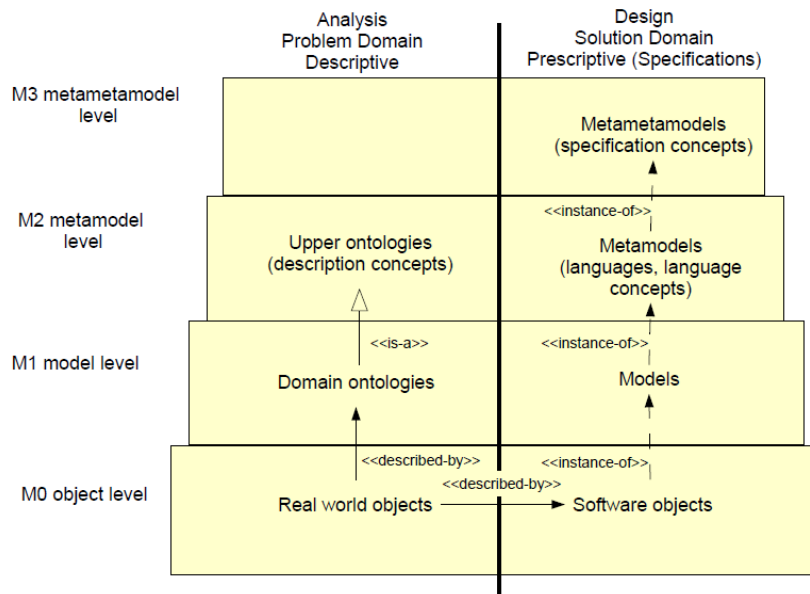


Figure 4: Schematic representation of the ontology-aware meta-pyramid. [11].

- *EPackage*: Represents a package or namespace for grouping meta-model elements.
- *EClass*: Defines a class or type in the meta-model, with attributes and references.
- *EAttribute*: Represents a simple attribute (e.g., a string or integer) of a class.
- *EReference*: Defines a relationship (containment or non-containment) between classes.
- *EDataType*: Represents primitive or custom data types (e.g., EString, EInt).
- *EOperation*: Defines operations or methods on a class.

In accordance with the MOF standard, Ecore is bootstrapped or *self-defining*, i.e., the Ecore meta-model is itself defined using Ecore, and provides *interoperability*, i.e., Ecore supports XMI for model serialization, aligning with other MOF-based tools.

3.2 Integrating Ontologies with Meta-modeling

While the MOF standard established a clear multi-level architecture, there was less consensus on the precise form it should take, and the role it should play. Researchers like Atkinson and Kuhne studying the role of meta-modeling in MDA [9], identified a number of shortcomings and/or ambiguities that made it difficult for model-driven development techniques to scale up [10]. They stated for instance that 'there is no explanation of how entities within the infrastructure relate to the real world', 'there is an implication that all instance-of relationships between elements are of fundamentally the same kind', and 'there is no explicit principle for judging at which level a particular element should reside' [10]. Atkinson and Kuhne emphasize the need for *ontological meta-modeling* in addition to *linguistic meta-modeling*. They refer to meta-modeling as ontological metamodeling when it is concerned with describing what concepts exist in a certain domain and what properties they have.

In general, ontologies are concerned with the nature and relations of being. An *ontology* encompasses a representation, formal naming, and definitions of the categories, properties, and relations between the concepts, data, or entities that pertain to one, many, or all domains of discourse. While a metamodel defines the rules and language for constructing models, it is a "model of a model", an ontology represents the actual knowledge, concepts, and relationships within a specific domain.

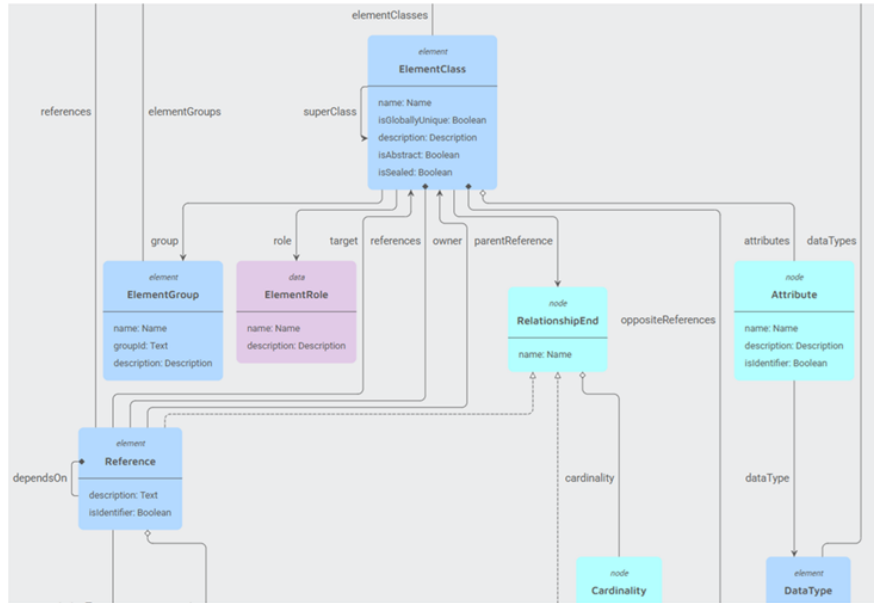


Figure 5: Schematic representation of lightweight M3 meta-meta-model.

Though ontologies are sometimes associated with the meta-modeling or M2 level, authors like Atkinson and Kuhne make it clear that the concept of ontologies is orthogonal to the multi-level meta-modeling hierarchy of Figure 3. An in-depth discussion on the role of descriptive and structural models, in particular ontologies, in the model-driven process can be found in [11]. Assmann, Zschaler, and Wagner propose an *ontology-aware meta-pyramid*, schematically represented in Figure 4. They distinguish *domain ontologies*, located at M1 and related to categories of things that exist or may exist in some domain, and *upper-level ontologies*, located at M2 and limited to concepts that are meta, generic, or abstract. Assmann et al. also argue that ontologies can be used as language descriptions, which has multiple consequences. As the same meta-language can be used on the ontology as well, this leads to *ontology meta-languages* that live on level M3. They also propose to employ domain ontologies into the MDA process, as it seems convenient to center software modeling around concepts of a domain, or structure of a domain, while adding behavior to it step by step.

This reasoning brings domain ontologies close to the concept of a *Domain-Specific Language (DSL)*. Such a DSL, a specialized language for domain-specific computation, is often considered to be different and complementary with respect to a domain ontology. However, both concepts become nearly identical in the context of software modeling in a model-driven development environment, fitting the description of the higher-level language as described by Dave Parnas in the context of automatic programming [1].

4 The NST Multi-level Modeling Architecture

In essence, the purpose of the NST architecture is to create an *executable architecture for model-driven development providing an ontology-aware multi-level meta-modeling architecture*.

4.1 Embedding NS Expansion in MOF

As explained in Section 2, the expansion of NS elements is a form of model-driven software development. In this subsection we describe how it is embedded in the MOF-based multi-level modeling architecture.

4.1.1 A Lightweight Meta-meta-model

For the M3 meta-meta-model, we adhere as close as possible to the *EMF Ecore* described in Section 3. To avoid unnecessary complexity in this multi-level executable architecture, EMF Ecore is quite an elaborate implementation, we have developed a lightweight version of the EMF Ecore standard implementation. The key elements, schematically represented in Figure 5, are nearly identical.

- *ElementGroup*: Represents a package or namespace for grouping meta-model elements.
- *ElementClass*: Defines a class or type in the meta-model, with attributes and references.
- *Attribute*: Represents a simple attribute (e.g., a string or integer) of a class.
- *Reference*: Defines a relationship (containment or non-containment) between classes.
- *DataType*: Represents primitive or custom data types (e.g., String, Int).
- *Operation*: Defines operations or methods on a class.

Similar to EMF Ecore and in accordance with the MOF standard, this lightweight M3 model is bootstrapped or *self-defining*, and supports XMI for model serialization, aligning with other MOF-based tools.

4.1.2 The NS Elements Meta-ontology

The meta-model to specify NS element models is located at level M2. In accordance with the need for ontological meta-modeling expressed by Atkinson and Kuhne [10], it is defined in terms of the lightweight M3 meta-meta-model, and is consistent with an upper-level ontology as discussed by Assmann et al. [11]. It is crucial to *close the engineering loop* and to *enforce consistency* between the definition of the meta-model at level M2, and the specification of the M1 models for the model-driven development.

As with most model-driven development environments, tooling is provided to create and modify M1 models of NS elements, and to drive the code generation process. The tool allowing software designers to specify M1 models for NS elements is called the *μRadiant*. To close the engineering loop, this tool has been extended in two ways.

- A separate functionality has been developed to visualize and author the M2 meta-model.
- The functional module to visualize and author M1 models has been re-engineered, and can be generated based on the definition of the M2 meta-model, the model of the model.

In this way, we are able to assure that the M1 models that are entered in the *μRadiant*, and used to drive the NS expansion, are consistent with the M2 meta-model.

4.2 Empowering Meta-model Ontologies

4.2.1 Extending and Creating Meta-models

The functionality to visualize and author the NST meta-model implies that it is both possible to extend this meta-model and to create different meta-models at the M2 level.

ddd - clean

hla

4.2.2 Supporting Meta-models Downward

The second part of the extended functionality means that

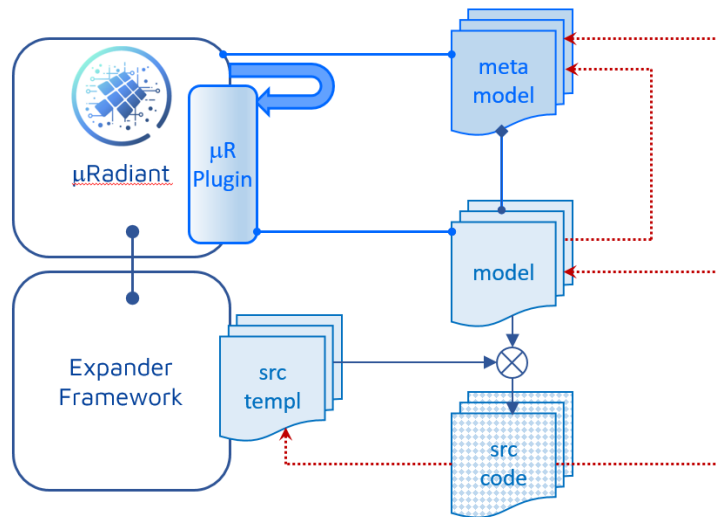


Figure 6: Schematic representation of supporting meta-models with plugins.

4.3 The Role of of Domain Ontologies

The actual models of the model-driven development are specified at the M1 level, and obviously contain domain information. This is also the modeling level where according to [11] the domain ontologies are located. Though we embrace the notion of ontological modeling as identified in [10], it is not clear that these domain models, based on the NST M2 meta-model, qualify as domain ontologies. However, the fact that these models, expressed in terms of the NST M2 meta-model, can be transformed into models expressed in terms of the M3 meta-meta-model, seems to validate this.

iot - ot dsp

4.4 Integrating Modeling Languages

[12]

5 Conclusions

References

- [1] D. L. Parnas. "Software Aspects of Strategic Defense Systems." *American Scientist*. November 1985.
- [2] Herwig Mannaert. "Controlling Artificial Intelligence: An NST Perspectiev", NSX White Paper. June 2026.
- [3] Mannaert, H., Verelst, J., De Bruyn, P., "Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design", NSI Press, 2016.
- [4] Eric Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software", Pearson, 2003.
- [5] Robert C. Martin, "Clean Architecture: A Craftsman's Guide to Software Structure and Design", Prentice Hall, 2017.
- [6] Simon, Herbert, "The Sciences of the Artificial", MIT Press, 1969.
- [7] Jean Bézivin, Richard Lemesle, "Ontology-Based Layered Semantics for Precise OA&D Modeling", Proceedings of the ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques, editors Haim Kilov and Bernhard Rumpe, 1997.
- [8] Object Management Group, "Meta-Object Facility 1.4 specification", April 2002.
- [9] Atkinson, C., Kuhne, T. "The Essence of Multilevel Metamodeling". International Conference on the Unified Modeling Language. 19-33, 2001.
- [10] Atkinson, C., Kuhne, T. "Model-driven development: a metamodeling foundation". *IEEE software*. [Online] 20 (5), 36–41, 2003.
- [11] Uwe Assmann, Steffen Zschaler, Gerd Wagner, "Ontologies, Meta-models, and the Model-Driven Paradigm", Conference on Ontologies for Software Engineering and Software Technology, 2006
- [12] Suchánek Marek, "Towards a normalized systems gateway ontology for conceptual models", Czech Technical University Prague & University of Antwerp, 2023.

Glossary

Systems Engineering

System: Refers to a set of interconnected components that work together to achieve a specific purpose or function.

Functional view: Refers to a core architectural perspective in systems engineering and software architecture, i.e., the *what* or the system's operational behavior and functionality.

Construction view: Refers to a core architectural perspective in systems engineering and software architecture, i.e., the *how* or the system's physical structure and implementation.

Functional Modeling

M3 Model: Refers to a meta-meta-model to define various meta-models or ontologies. Such a meta-meta-model needs to be meta-circular, i.e., defined within itself.

M2 Model: Refers to a meta-model, ontology or Domain Specific Language (DSL), specified in an M3.

M1 Model: Refers to an actual model, i.e., an instantiation of an M2 meta-model or ontology.

M0 Model: Refers to actual data, i.e., an instantiation of an M1 model in an information system.

Functional Type: Refers to the M2 ontology used to specify the functional view, identified by the name of its root element, e.g., 'Application' for the 'elements' ontology.

Functional Element: Refers to an 'ElementClass' or node in the M2 ontology specifying the functional view, e.g., 'component' or 'data element'.

Functional Model: Refers to an M1 model or an instantiation of an ontology from the root element, e.g., the 'Application' named 'E-Crystal'.

Functional Module: Refers to an M1 model or an instantiation of an ontology from a specific node, high enough in the hierarchy to be re-usable, e.g., the 'component' named 'invoicing'.

Construction Architectures

High-Level Architecture: Refers to the high-level specification of the construction architecture, including the interconnected components or subsystems, and their interconnections.

High-Level Subsystem: Refers to a subsystem identified in the high-level architecture, including the detailed specification of the interfaces and interactions with the other subsystems.

Tier: Refers to the physical or logical separation of components in a system, typically across different servers, machines, or processes. Tiers are about the deployment structure and how components are distributed.

Layer: Refers to the logical separation of functionality within a single system or application. Layers are about organizing code or components within the same process or application.

Client Library: Refers to a runtime client library provided to give developers in-memory access to remote information systems while encapsulating the remoting technology.

Architectural Style: Refers to a specific high-level architecture that describes the construction view, e.g., 'MIS – multi-tier information system'.

Program Type: Refers to an architectural style where some technologies and/or standards have been predefined, e.g., 'JEE Application'.

Program Layer Type: Refers to a high-level subsystem or layer where a technology or a set of technologies have been predefined on top of the program type, e.g., 'JAX-RS Control Layer'.

Element Structure: Refers to the detailed construction design for the implementation of a functional element, e.g., 'Data Element class structure' in a program type or program layer type.

Systems Expansion

Expander: Refers to a unit of generation that combines an instance of a data tree, i.e., a tree or subtree of a functional M1 model, with a template, to generate an artifact.

Expansion Resource: Refers to a set of expanders that share a specific repository, and are packaged into an expansion resource or archive, which may contain other resources, e.g. code or data resources.

Expander Bundle or Expansion Resource Bundle: Refers to a set of expanders, packaged in one or multiple expansion resource archives. Considered to be rather *informal language*.

Expander Beam: Refers to a packaged and certified set of expanders, in one or multiple resource archives, that is able to generate the various element structures of a functional model for a specific program layer type, e.g., 'MIS Control Layer JAXRS', or an entire program type, e.g., 'MIS JEE Application'.

Expander Beam Aggregate: Refers to a specified collection of expander beams, that aggregates specific complementary variants of expander beams for different program layer types within a program type architecture, e.g., {'MIS View Layer Knockout', 'MIS Control Layer JAXRS', 'MIS Logic-Data Layer JEE'}.

System Architecture: Refers to the combined specification of an ontology or M2 meta-model to specify the functional view, and an architecture style or program type that describes the construction view.

Software Program: Refers to an instantiation of a system architecture, the combined specification of a functional M1 model, and an instance of a program type, e.g., 'E-Crystal JEE Application'.

Expansion Technology Settings: Refers to a set of technology settings, specifying the utility frameworks, that will be added to the data tree of the functional model to drive the actual expansion.

Software Program Expansion: Refers to a software program that has been expanded with specific expansion technology settings, and is eligible for building and deployment.

Software Project Composition: Refers to a collection of software programs, that can be expanded, built and deployed as an integrated system, e.g., {'C2C JEE Application', 'Telemetry MQTT MicroServer'}.