A Method for Composing Concerns into a Unified Domain Model in Domain-Driven Design

Van-Vinh Le[1,2], Nhat-Hoang Nguyen[2], Duc-Quyen Nguyen[2], and Duc-Hanh Dang[2*]

[1] Vinh University of Technology Education levanvinh@vuted.edu.vn
[2] Faculty of Information Technology, VNU University of Engineering and Technology, Hanoi, Viet Nam
{24025170, 21020388, hanhdd}@vnu.edu.vn

Abstract. Domain-Driven Design (DDD) advocates iterative software development centered on a semantically rich domain model to bridge the communication gap between domain experts and developers. Although the adoption of a ubiquitous language and Domain-Specific Languages (DSLs) enhances the expressiveness and maintainability of software systems, the increasing complexity and heterogeneity of modern systems necessitate the integration of multiple concern-specific DSLs to capture diverse aspects. However, current DDD approaches lack systematic mechanisms for composing such heterogeneous DSLs, leading to fragmented domain models, weak traceability, and limited automation throughout the development pipeline. To address these challenges, this paper proposes a method for composing concerns into a unified abstract syntax tree (AST) within the context of DDD. The approach specifies the abstract syntax, concrete syntax, and formal semantics of concern-specific DSLs in a consistent manner. Building on this foundation, we introduce an annotation-based composition mechanism that interrelates DSLs and integrates multiple concerns into a unified domain model. This mechanism ensures concern orthogonality, preserves model cohesion, and supports backward traceability across artifacts. We demonstrate the feasibility and practicality of the proposed methodology through a proof-of-concept implementation using the JetBrains MPS language workbench and the JDA framework, evaluated on representative case studies. Our contributions extend the state of the art in modular, extensible, and executable domain modeling for complex software systems.

Keywords: DDD · DSL · Domain Concern · Domain Model · MPS

1 Introduction

The increasing complexity, scale, and heterogeneity of modern software systems have intensified the need for methodologies that effectively bridge domain knowledge and technical implementation. Domain-Driven Design (DDD) has emerged as a prominent approach to this challenge, advocating iterative development

* Corresponding author

around a semantically rich domain model that encapsulates the core logic and rules of the problem domain [8, 14]. Central to DDD is the consistent use of a Ubiquitous Language (UL), which fosters effective communication and shared understanding between domain experts and developers throughout the software lifecycle.

To operationalize the UL and align it with implementation artifacts, Domain-Specific Languages (DSLs) have been widely adopted as expressive means of encoding domain concepts directly in source code [2, 10]. DSLs tightly couple design models with implementation, improving software correctness, maintainability, and extensibility [1, 2, 4, 9, 22, 24].

In the development of complex software systems, defining and integrating multiple concern-specific DSLs—such as those for business logic, security, user interfaces, and performance—is crucial to addressing diverse requirements and abstraction levels [11, 13]. A key challenge in applying DDD to such systems lies in the synchronous representation and integration of these heterogeneous concerns. The absence of systematic composition mechanisms fragments domain models and weakens UL consistency. Moreover, DSL development pipelines are often distributed across isolated tools for editing, verification, and code generation, thereby hindering automation. As domain models evolve, maintaining backward traceability between DSLs and generated code becomes increasingly difficult, further reducing the practical effectiveness of DDD.

Existing DDD approaches support domain modeling but often neglect the systematic formation and management of software modules derived from domain models [5, 12, 13, 23], leading to the absence of robust methodologies for their development and evolution. Furthermore, the lack of precise characterization of software artifacts generated from domain models hampers the effective adoption of DDD in real-world systems.

Current approaches often address concerns in isolation, resulting in tight coupling and limited extensibility. Recent efforts, including our approach to composing concerns into an executable unified domain model (UDML) [18], introduce a novel metamodel based on UML metaconcepts that supports the composition and unification of concern-specific DSLs. Nevertheless, significant limitations remain: integration processes are typically semi-automated and error-prone, rely heavily on high-quality mappings between DSLs, and provide limited support for complex or cross-cutting concerns. Scalability also becomes problematic as the number and complexity of concerns increase, while existing approaches still fall short in capturing sophisticated structural and behavioral aspects [6, 17].

In this paper, we propose a method for composing concerns into a unified domain model. Our approach extends UDML, a comprehensive DSL with well-defined abstract syntax (AS), concrete syntax (CS) and semantics, by unifying multiple concern-specific DSLs into a single abstract syntax tree (AST). This mechanism integrates concern-specific DSLs into the core model while ensuring orthogonality, cohesion, and consistency across concerns. Each concern is specified using its own DSL and systematically composed into the unified model, thereby facilitating complexity management and improving modularity and extensibility. We implement and validate the approach using the projectional editing capabilities of JetBrains MPS [3, 16] and the DDD framework JDA [7], demonstrating its feasibility and effectiveness in supporting composable, maintainable, and executable domain models through real-world case studies.

In brief, the contributions of this paper are as follows:

- A methodology for unifying multiple concern-specific DSLs (e.g., DCSL [17], AGL [6], RBAC [19]) into a single AST of UDML;
- A complete definition of the syntax and semantics of UDML, including AS, CS, and formal semantics;
- A tool-supported proof-of-concept implementation on the MPS platform, advancing the state of the art in modular and extensible domain modeling.

The rest of the paper is organized as follows: Section 2 motivates our work with examples. Section 3 explains the basic idea of our approach. Section 4 provides the syntax and semantics for UDML and the mechanism to compose concerns. Tool support and experiments are explained in Section 5, which discusses the proposed language. Section 6 surveys related work. Finally, Section 7 concludes the paper with a discussion of future work.

2 Motivating Example and Background

This section motivates our work through an example and provides the necessary background.

2.1 Motivating Example

We use the `CourseMan` domain, adapted from [17], as a motivating example. Fig. 1 presents the `CourseMan` domain model, based on OMG/UML, combin-

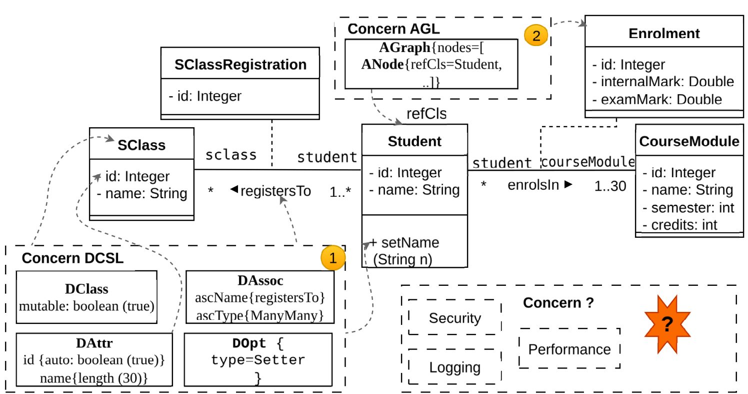

Fig. 1. UML class diagram and concern DSLs in CourseMan.

ing a class diagram with concern-specific DSLs. It includes three main classes (`Student`, `CourseModule`, `SClass`) and two association classes (`SClassRegistration`, `Enrolment`). The dashed shapes labeled (1) represent the DCSL concern, which captures the structural aspect through four concepts (`DClass`, `DAttr`, `DOpt`, `DAssoc`) defining entities and associations (e.g., `enrolsIn`, `registersTo`). The dashed shapes labeled (2) denote the AGL concern, which specifies behavioral aspects through `AGraph` defining the reachable state, while the star-like shape '`?`' indicates extensibility for future concerns.

The `CourseMan` system exemplifies a domain with intertwined concerns such as structure, enrollment, access control, and grading. While DCSL [17] supports structural modeling, it lacks modularity, expressiveness, and extensibility for addressing behavior, roles, constraints, and security. This highlights the need for a modeling approach that ensures separation of concerns, systematic integration, and continuous evolution of heterogeneous DSLs within a unified, executable model.

2.2 Background

To leverage domain expertise effectively, we revisit foundational concepts that underpin our concern-driven DSL integration approach.

Executable Domain Models in DDD. Domain-Driven Design (DDD) [8] advocates iterative development around a semantically rich domain model that encapsulates core logic and business rules. This model establishes a shared foundation between domain experts and developers through a consistent Ubiquitous Language (UL) [14, 25]. Domain-Specific Languages (DSLs) reinforce this alignment by encoding domain knowledge directly into software artifacts [2, 10], enhancing correctness, maintainability, and extensibility.

A key advancement in DDD is the executable domain model, which unifies descriptive and operational semantics [27]. By integrating concerns such as business logic and security into a runnable specification, executable models enable automation, verification, and adaptability [1]. However, achieving this vision requires automated composition of heterogeneous concerns—an ongoing challenge in DDD research.

Composing Concern DSLs in UDML. Concerns refer to orthogonal aspects of a system, such as structure, behavior, access control, or performance, that must be independently modeled and composed [15]. UDML [18] supports the composition of such concerns through internal and external DSLs. It extends UML metaconcepts to integrate concern DSLs within a cohesive metamodel.

Each concern is modeled using a dedicated DSL: DCSL [17] for structural aspects, AGL [6] for behavioral logic, and RBAC [19] for access control, with flexibility to incorporate additional DSLs as needed. However, integrating these heterogeneous DSLs into a cohesive domain model poses a significant challenge, as it requires well-defined composition mechanisms that operate at the AST level.

Our goal is to enhance the extensibility and expressiveness of UDML's syntax to enable executable domain modeling in complex software systems. To this end, we address two research questions: (1) How can concern-specific DSLs be systematically composed into a unified AST? (2) Which language composition mechanisms and tool-supported approaches best support the integration and evolution of heterogeneous concerns in modular, scalable domain models?

3 Overview of Our Approach

This section introduces our approach, termed composing concerns into a unified domain model.

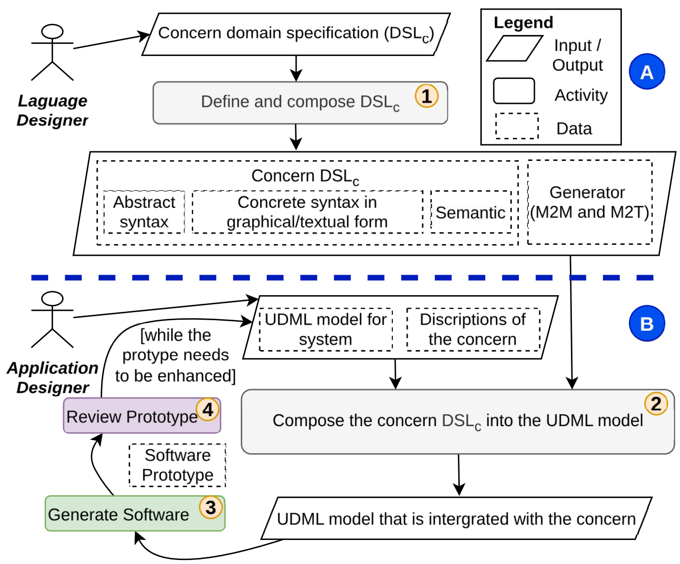

Fig. 2. Overview of our approach, structured into (A) language design level and (B) language application level.

Fig. 2 illustrates the method, which follows a four-step iterative process to systematically integrate and refine concern-specific DSLs for the automatic generation of software prototypes. *First*, at the language designer level, the designer specifies the concern domain using a domain-specific language (DSL_c) as input. For each DSL_c, we formally define its AS, CS, and semantics, resulting in a concern (DSL_c) that precisely captures the target concern. *Second*, at the application designer level, the designer leverages both the output from step one and existing UDML models, along with descriptions of new or evolving concern requirements. We compose the concern DSL_c into the UDML model to ensure coherent integration. Designers may use textual or graphical CS to construct a detailed model that represents the specific concern instance. The output of this step is a UDML model fully integrated with the given concern. *Third*, the integrated UDML model serves as the foundation for generating production software. The resulting software artifact is then evaluated by the designer to gather feedback. *Finally*, if feedback is provided, both the UDML model and the corresponding concern specification are updated accordingly. The process then repeats, supporting a continuous improvement cycle until the software system meets the specified requirements.

4 Composing Concern DSLs

This section introduces a method for composing concerns into a unified language, namely UDML, which serves as a modular language ecosystem where independently developed concern-specific DSLs are unified—enabling flexible integration and system-wide consistency.

4.1 Defining the Syntax and Semantics of UDML

This work defines UDML with complete AS, CS, and semantics, and integrates concern-driven domain modeling.

Definition 1. *The **UDML** language is constructed from a core DSL and a set of concern DSLs $\{DSL_i\}_{i=1}^{n}$, where each DSL_i describes a distinct aspect of the system.*
Each DSL is specified by a triplet (AS_i,CS_i,Sem_i), where:

- *AS_i: Abstract Syntax;*
- *CS_i: Concrete Syntax;*
- *Sem_i: Semantics*

We define the AS of UDML as a collection of domain concepts, attributes, instances, and relations organized into separate concern modules based on UML metaconcepts [21], as shown in Fig. 3. Each concern DSL defines its own AS but conforms to a shared metamodel, enabling inheritance, extension, and composition according to predefined formal rules. We extend the UDML metamodel [18] with the `RefAnnotation` and `Relationship` metaconcepts to support seamless integration of new concerns: the AS can be incrementally extended by adding new concepts, attributes, or relations without disrupting the global structure.

Definition 2. *The **AS** of UDML is defined as the tuple*
$AS_{UDML} = (C_{UDML}, A_{UDML}, R_{UDML}, P_{UDML})$*, where:*

- *C_{UDML}: Set of concepts (including core and concern DSL concepts)*
- *A_{UDML}: Set of attributes*
- *R_{UDML}: Set of relations between concepts*
- *P_{UDML}: Set of well-formedness rules*

The mapping of metaconcepts from the AS to the CS in a graphical User Interface (UI) is carried out systematically as follows:

- Each metaconcept, representing a core domain construct such as `Class`, `Property`, `RelationShip`, or `Annotation`, is directly mapped to one or more UI elements.

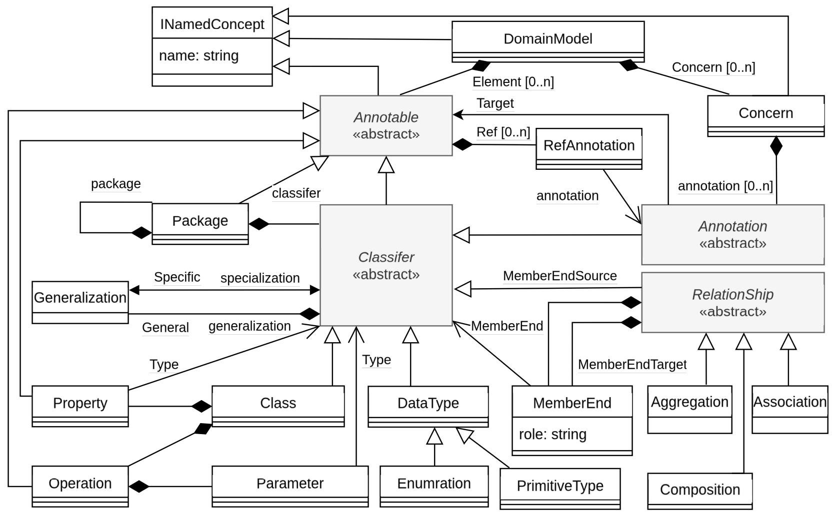

Fig. 3. Metamodel of UDML

- Entity-like metaconcepts are represented as distinct graphical blocks with labels and icons on the diagram.
- Relational metaconcepts are visualized as connectors or arrows linking the corresponding blocks.
- Attributes and features of a metaconcept are displayed as fields, columns, or labels within the blocks, or managed through separate configuration dialogs.
- Annotations are visualized using tags, labels, icons, or small graphical overlays attached to the target entities.

Definition 3. *The **CS** of UDML is defined as the tuple* $CS_{UDML} = (N, E, L, P, H)$*, where:*

- N*: The set of nodes representing instances of concepts in* C_{UDML}*. Each node corresponds to a concrete entity on the UI.*
- $E \subseteq N \times N$*: The set of edges between nodes, representing relationships in* R_{UDML} *displayed visually.*
- $L : N \cup E \to$ *Labels: A labeling function that assigns display labels to nodes or edges.*
- $P : N \cup E \to$ *Prop: A function mapping UI elements (nodes or edges) to a set of graphical properties to define their concrete visual representation.*
- $H = \{h_i\}$*: A set of handlers—rules or operations—that support interactive actions such as drag-and-drop, creation, modification, and deletion of nodes and edges on the interface.*

UDML's overall semantics derive from the individual semantics of each concern DSL, enabling independent control and verification of changes at the concern level. Constraint semantics are enforced through two mechanisms: (i) structural constraints embedded in metaconcepts for well-formedness rules [18], and

(ii) checking rules for more complex logic. For instance, a checking rule validates that an `Annotation` applies only to a `Class`, reporting an error if violated.

```
if (!(node.target.isInstanceOf(concept<Class>)))
{    error("Target must be a Class."); }
```

4.2 A Mechanism for Composing Concern DSLs in UDML

Concern DSLs are integrated into UDML via a systematic annotation-based approach. Core elements implement a generic `Annotable` interface, while each DSL defines its own `Annotation` types referencing these elements. This allows concern-specific semantics to be attached modularly and non-intrusively, preserving separation and traceability.

Algorithm 1 Tree-merging algorithm for integrating concern DSLs into UDML

Input: $D = \{DSL_1, DSL_2, \ldots, DSL_n\}$: A set of concern DSLs, each specified as a triplet (AS_i, CS_i, Sem_i).
Output: $UDML_{unified}$: The unified UDML comprising a global AS, CS, and Sem

Initialize: $AS_{UDML} \leftarrow AS_{core}$, $CS_{UDML} \leftarrow CS_{core}$, $Sem_{UDML} \leftarrow Sem_{core}$
for each $DSL_i = (AS_i, CS_i, Sem_i) \in D$ **do**
  for each concept $c \in AS_i$ where $c \notin AS_{UDML}$ **do** //*Merge Abstract Syntax*
    if c extends core concept **then**
      Integrate inheritance per metamodel
    end if
    Add attributes, relations, constraints to AS_{UDML}
  end for
  for each new or extended concept **do** //*Merge Concrete Syntax*
    Add graphical elements (nodes, edges, labels)
    Extend UI handlers (drag-drop, annotations)
  end for
  Integrate Sem_i into Sem_{UDML} modularly //*Merge Semantics*
  Compose new constraints into P_{UDML} and update validation rules.
end for
return $UDML_{unified} = (AS_{UDML}, CS_{UDML}, Sem_{UDML})$

Algorithm 1 presents a tree-merging process that incrementally unifies concern-specific DSLs into the UDML framework. The algorithm merges AS by incorporating new concepts and extending core types. CS is augmented with graphical elements and interactive behaviors associated with the extended concepts. Formal semantics are composed modularly, integrating validation rules and behavioral constraints defined by each concern. This yields a unified, executable model that consolidates diverse concerns while preserving their modularity and traceability.

5 Tool Support and Experiments

We developed a support tool for UDML that enables the modular separation of concern-specific DSLs, allowing designers to directly model all relevant aspects of the domain. Our approach is implemented using JetBrains MPS [3,26], which provides a flexible platform for AST-based language composition and seamless integration into existing development workflows. The tool facilitates the design, extension, and integration of concern DSLs, validating the approach's expressiveness, feasibility, and satisfiability. The practical applicability of our method is illustrated in Figure 4, and the implementation is publicly available at the Git repository[3].

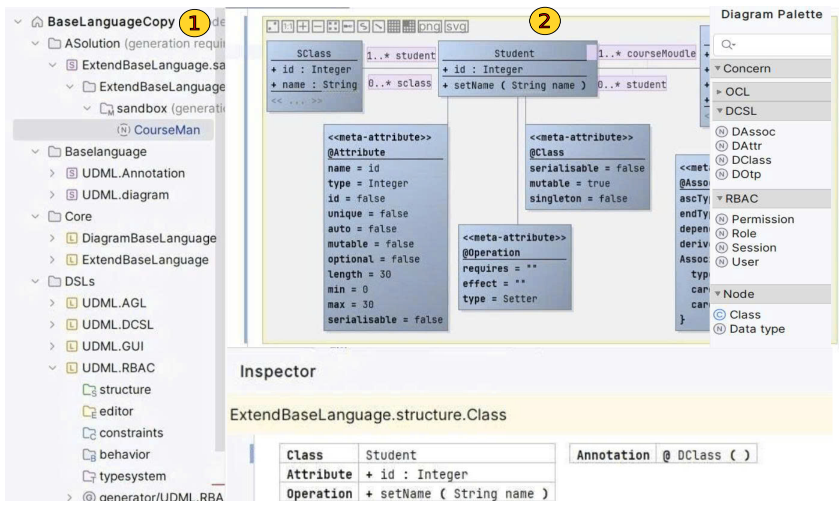

Fig. 4. MPS-based realization of concern DSL integration in UDML

Figure 4 illustrates our implementation. On the left, label (1) defines three core components: (i) Instructor, specifying metaconcepts for UDML and concern DSLs; (ii) Editor, providing a graphical concrete syntax with drag-and-drop capabilities; and (iii) Generator, transforming the AST into executable code via the JDA framework [7]. We designed the UDML language as modular packages: `UDML.core` provides the integration foundation, while `UDML.dcsl`, `UDML.agl`, and `UDML.rbac` capture structural, behavioral, and security concerns, respectively. The central area offers a graphical interface for domain modeling, supported by a palette that enables intuitive construction through rapid insertion of concern-specific elements.

We applied the support tool to the `CourseMan` case study. On the right of Figure 4, label (2) illustrates the UML class diagram along with the integrated concern DSLs: DCSL, AGL, and RBAC. The designer implements the design using drag-and-drop operations. The tool subsequently generates the corresponding source code, and a software prototype is then produced using the JDA framework, with further technical details provided in the Git repository[4].

[3] https://github.com/vinhskv/udml-syntax-soict2025.git
[4] https://github.com/vinhskv/udml-syntax-soict2025/tree/main/technical

Discussion. We have successfully applied UDML in developing software for real-world problem domains. However, several validity concerns arise. First, correctness depends on the expressive adequacy of individual concern DSLs and their coherent integration via UDML, in line with DDD principles. Second, integrating heterogeneous external DSLs requires sufficient syntactic support, which is currently ensured through systematic metaconcept validation, interface design, transformation testing, and verification of generator rules. Third, the generated Java code may misalign with requirements if generation errors occur within the JDA framework. To mitigate this, we conduct careful review and verification of the generated output.

6 Related Work

We position our work at the intersection between the following areas: DSL Engineering, DDD concern DSLs, Integrating concern DSLs.

DSL Engineering and Language Workbenches. DSL engineering has matured significantly [27], with language workbenches such as JetBrains MPS [3] playing a pivotal role in enabling the design, implementation, and maintenance of DSLs for both academic and industrial use. The ability to define, extend, and compose DSLs in MPS has been demonstrated in various domains, including finance, healthcare, safety-critical systems, and industrial automation, where tailored DSLs have led to increased productivity, improved quality, and more effective leveraging of domain expertise.

DDD and Concern-Specific DSLs. Within the DDD paradigm [25], the centrality of the domain model and the use of a ubiquitous language are well established. However, as software systems grow in scale and complexity, they often require the modeling and integration of multiple concerns—such as business logic, security, user interface, and performance—each potentially realized as a separate DSL [1]. Prior work has shown that handling these concerns in isolation can lead to fragmentation, inconsistencies, and difficulties in maintaining traceability between the domain model and implementation artifacts.

Integrating Concern DSLs. Recent research has emphasized systematic methods for integrating heterogeneous concern DSLs [13,20]. Platforms like JetBrains MPS [3] have advanced this field by enabling modular language development through reuse and composition, thereby enhancing efficiency and quality.

In our previous work [18], we proposed an approach to composing concerns into an executable unified domain model. This approach leverages a metamodel based on UML metaconcepts to support the composition and unification of concern-specific DSLs. Each concern is specified using an external DSL, which is then mapped to an internal DSL to produce an executable domain model embedded in an object-oriented programming language for source code generation. In contrast, the present work defines the full syntax of concern-specific DSLs and introduces an integration method based on tree merging at the AST level. Moreover, we define a unified syntax for composing concern-specific DSLs.

Nevertheless, existing approaches often fall short in supporting the complete DSL development pipeline—spanning editing, verification, code generation, and automated testing. Backward traceability between models and generated code remains a critical challenge, particularly as systems evolve.

7 Conclusion and Future Work

This paper introduces a methodology for integrating multiple concern-specific DSLs into a unified domain model. Building upon this foundation, we define the complete syntax and semantics of UDML, an external DSL designed to support domain-driven development. UDML is structured as an AST, where each node corresponds to a concept from a specific concern DSL. This structure enables modular development and seamless integration of heterogeneous concerns, while preserving consistency across structural, behavioral, and security dimensions of the domain model. The proposed approach is realized through a prototype tool developed using JetBrains MPS and the JDA framework. The tool demonstrates the feasibility, expressiveness, modularity, and executability of UDML in modeling complex software systems aligned with DDD principles.

In the future, we plan to develop an MPS plug-in for our method for many domain concerns. We also intend to develop a technique for automatically transforming a specification requirement into complex software systems.

References

1. Andrzej Wąsowski, Thorsten Berger: Domain-Specific Languages Effective Modeling, Automation, and Reuse. Springer Cham (2023)
2. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan&Claypool, 2nd edn. (2017)
3. Bucchiarone, A., Cicchetti, A., Ciccozzi, F., Pierantonio, A.: Domain-specific languages in practice: with JetBrains MPS. Springer Nature (2021)
4. Buede, D.M., Miller, W.D.: The Engineering Design of Systems: Models and Methods. Wiley, 4th edn. (2024)
5. Dan Haywood: Apache Isis - Developing Domain-Driven Java Apps. Methods & Tools: Practical knowledge source for software development professionals **21**(2), 40–59 (2013)
6. Dang, D.H., Le, D.M., Le, V.V.: AGL: Incorporating behavioral aspects into domain-driven design. Information and Software Technology **163**, 107284 (Nov 2023)
7. D.M. Le, D.-H. Dang, V.-H. Nguyen: Generative software module development for domain-driven design with annotation-based domain specific language. Information and Software Technology **120**, 106–239 (2022)
8. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional (2004)
9. Foster, E., Jr, B.T.: Software Engineering: A Methodical Approach, 2nd Edition. Auerbach Publications, New York, 2 edn. (Jul 2021). https://doi.org/10.1201/9780367746025
10. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, 1st edn. (Sep 2010)

11. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE'07). pp. 37–54. IEEE (2007)
12. Gerasimov, A., Michael, J., Netz, L., Rumpe, B.: Agile Generator-Based GUI Modeling for Information Systems. In: Modelling to Program, vol. 1401, pp. 113–126. Springer International Publishing, Cham (2021)
13. Haber, A., Look, M., Perez, A.N., Nazari, P.M.S., Rumpe, B., Völkel, S., Wortmann, A.: Integration of heterogeneous modeling languages via extensible and composable language components. In: 3rd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD). pp. 19–31. IEEE (2015)
14. Jaiswal, S.K., Agrawal, R.: Domain-Driven Design (DDD)- Bridging the Gap between Business Requirements and Object-Oriented Modeling. Int. Innovative Research in Engineering and Management **11**(2), 79–83 (2024), number: 2
15. Katara, M., Katz, S.: A concern architecture view for aspect-oriented software design. Software & Systems Modeling **6**(3), 247–265 (Aug 2007). https://doi.org/10.1007/s10270-006-0032-x
16. Latifaj, M., Taha, H., Ciccozzi, F., Cicchetti, A.: Cross-Platform Blended Modelling with JetBrains MPS and Eclipse Modeling Framework. In: ITNG 2022 19th Int. Conf. Information Technology-New Generations. pp. 3–10. Springer (2022)
17. Le, D.M., Dang, D.H., Nguyen, V.H.: On domain driven design using annotation-based domain specific language. Computer Languages, Systems & Structures **54**, 199–235 (Dec 2018)
18. Le, V.V., Dang, D.H.: An Approach to Composing Concerns for an Executable Unified Domain Model. In: Int. Conf. On Computing and Communication Technologies (RIVF). pp. 424–428 (Dec 2024). https://doi.org/10.1109/RIVF64335.2024.11009109, iSSN: 2473-0130
19. Li, Y., Du, Z., Fu, Y., Liu, L.: Role-based access control model for inter-system cross-domain in multi-domain environment. Applied Sciences **12**(24), 13036 (2022), publisher: MDPI
20. Nosál', M., Sulír, M., Juhár, J.: Language composition using source code annotations. Computer Science and Information Systems **13**(3), 707–729 (2016)
21. OMG: Unified Modeling Language 2.5.1. Object Management Group (2017)
22. Ouchani, S., Debbabi, M.: Specification, verification, and quantification of security in model-based systems. Computing **97**(7), 691–711 (Jul 2015). https://doi.org/10.1007/s00607-015-0445-x
23. Paniza, J.: Learn OpenXava by example. CreateSpace, 1.1 edn. (2011)
24. Suryn, W.: Software Quality Engineering: A Practitioner's Approach. John Wiley & Sons (Dec 2013), google-Books-ID: 1dNiAgAAQBAJ
25. Vernon, V.: Implementing domain-driven design. Addison-Wesley Professional, 1 er edn. (2013)
26. Voelter, M.: Language and IDE Modularization and Composition with MPS. Lecture Notes in Computer Science (LNPSE), vol. 7680, pp. 383–430. Springer, Berlin, Heidelberg (2013)
27. Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, implementing and using domain-specific languages. M Volter / DSLBook.org, Stuttgart, Germany (2013)