European Journal of Technical and Natural Sciences 2026, No 4
ISSN 2414-2352

PREMIER Publishing
ppublishing.org

Section 3. Computer science

DOI:10.29013/EJTNS-26-4-19-26

DOMAIN-DRIVEN DESIGN AS A BASIS FOR ALIGNING BUSINESS REQUIREMENTS AND IT PRODUCT ARCHITECTURE

Daria Kolesnikova [1]

[1] Senior IT Consultant, DB Fernverkehr AG, Frankfurtsa. Main, Germany

Cite: *Kolesnikova D. (2026). Domain-driven design as a basis for aligning business requirements and IT product architecture. European Journal of Technical and Natural Sciences 2026, No 4.* https://doi.org/10.29013/EJTNS-26-4-19-26

Abstract

The article examines domain-driven design as a methodological basis for aligning business requirements, analytical descriptions of processes, and IT product architecture. The problem of the gap between business logic and technical implementation is revealed, arising from the ambiguity of requirements, insufficient formalization of business rules, and weak connection between analytical artifacts and architectural decisions. It is shown that the domain model functions as an intermediate semantic structure that makes it possible to identify bounded contexts, localize component responsibility, clarify service boundaries, and reduce the risk of technical debt accumulation. It is substantiated that the application of such a model improves the modularity, maintainability, and resilience of the architecture of complex digital systems under changing functional, organizational, and technological requirements.

Keywords: *domain-driven design, domain model, IT product, business requirements, software architecture, technical debt*

Introduction

The development of IT products is increasingly associated with the need to simultaneously consider business objectives, user scenarios, regulatory constraints, and technological parameters of the future system. Business stakeholders, as a rule, describe expected outcomes through applied tasks, operational processes, requirements for efficiency, and the quality of user interaction, whereas engineering teams must transform these expectations into architectural decisions, software components, interfaces, and data models. At this stage, there is a risk of semantic and methodological misalignment between the business description of the product and the logic of its technical implementation, which leads to ambiguous interpretation of requirements, more complex project communication, an increase in architectural rework, and reduced resilience of the IT product to subsequent changes.

In this regard, domain-driven design (DDD) can be considered a methodological

basis for aligning business logic with the architecture of a software system. Its significance is determined by the fact that the subject area acts not as an external source of separate requirements, but as a structure-forming element of design that defines the semantic boundaries of the product, key entities, interaction rules, constraints, and scenarios of system change. The aim of the article is to analyze the role of DDD as a tool for formalizing the subject area, reducing the ambiguity of business requirements, and ensuring a more stable relationship between the architecture of an IT product and the logic of its practical application.

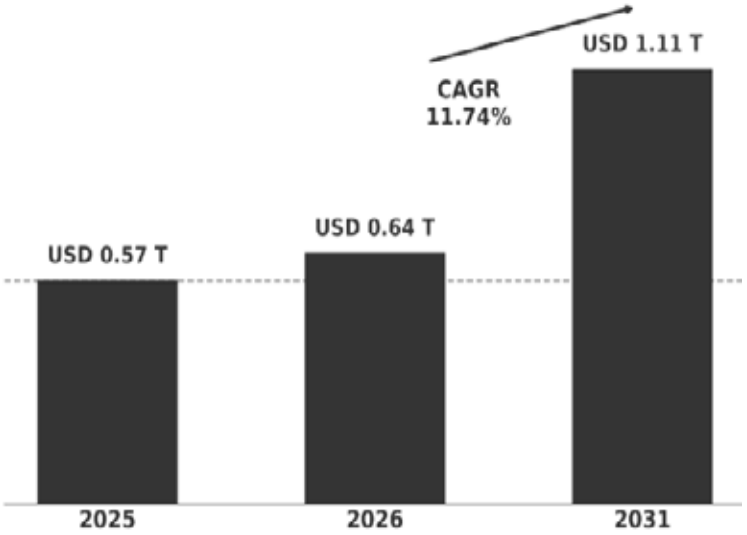
Main part

The problem of the gap between business requirements and IT product architecture

The gap between business requirements and IT product architecture arises when expected business outcomes are not transformed into explicit architectural principles, areas of responsibility, and technical models. Business stakeholders usually describe a product through objectives, processes, user scenarios, and economic effects, whereas engineering teams must translate these expectations into modules, interfaces, data structures, and integration mechanisms (Kovalevskyi, 2026). In the absence of a shared semantic basis, architecture begins to reflect not the logic of the subject area, but a set of separate functional requests.

The relevance of this problem increases as the scale of software product development continues to grow. According to Mordor Intelligence, the global software development market is estimated at approximately $0.64 trillion in 2026 and may reach $1.11 trillion by 2031, with a compound annual growth rate of 11.74% (fig. 1).

Figure 1. *Dynamics of the global software development market size in 2025–2031, trillion dollars (Mordor Intelligence, 2026)*



These data indicate that the alignment between requirements and architecture has direct economic significance. As software development scales, errors in interpreting business logic increasingly affect implementation timelines, maintenance costs, modernization complexity, and the resilience of digital solutions.

One of the main causes of this gap is the ambiguity of requirements. Business formulations often contain implicit rules, exceptions, dependencies, and constraints that are clear to domain experts but insufficiently formalized for engineering implementation. For example, a requirement to automate application processing may include statuses, user roles, data validation procedures, execution deadlines, access rights, and exceptional scenarios. If these elements are not recorded in a shared model, technical implementation is based on assumptions, which increases the risk of duplicated logic and architectural inconsistencies.

Thus, the gap between business requirements and IT product architecture is formed at the intersection of semantic, analytical,

and technical factors. Ambiguous requirements, insufficient formalization of business rules, and weak linkage between business descriptions and architectural decisions create conditions under which the system accumulates duplicated logic, excessive dependencies, and contradictory implementation patterns. This substantiates the need for an approach that can use the subject area as a common basis for interpreting requirements and structuring software architecture.

The domain model as a tool for semantic formalization of requirements

Generally, DDD responds to this need by placing the subject area at the center of IT product design. It is based on the formalization of domain concepts, business rules, processes, and bounded contexts, around which software architecture is subsequently constructed. Its main purpose is to ensure consistency between the product's business logic, the analytical description of requirements, and the technical implementation of the software system (Kumar et al., 2025). This consistency is achieved through the domain model, which acts as an intermediate semantic structure between business requirements and architectural decisions.

In this context, the domain model specifies the internal organization of the subject area through entities, processes, rules, roles, constraints, states, and relationships. Unlike isolated functional requirements, it represents product logic as an integrated system of concepts and dependencies, which makes it possible to compare functional descriptions with domain-based modeling principles (table 1).

Table 1. *Comparative characteristics of functional requirements and the domain model in IT product design*

Comparison criterion	Functional requirements	Domain model
Main focus	Define the actions and functions that the system must perform.	Describe the internal logic of the subject area and its structural elements.
Level of representation	User stories, functional scenarios, interface behavior, and expected outputs.	Entities, value objects, business rules, states, relationships, and constraints.
Treatment of business rules	Business rules may remain implicit, fragmented, or distributed across separate requirements.	Business rules are formalized as part of the domain logic and linked to specific entities or processes.
Connection with architecture	Do not always define component boundaries, ownership of logic, or responsibility areas.	Support the identification of bounded contexts, service boundaries, and zones of responsibility.
Risk of ambiguity	Ambiguity may persist due to different interpretations by business and engineering teams.	Ambiguity is reduced through a shared semantic structure and a unified understanding of the subject area.
Practical role in design	Specify what the system is expected to do.	Clarify how business logic should be represented, structured, and preserved in technical implementation.

The comparison presented in Table 1 shows that the domain model performs a broader role than a conventional functional description of requirements. While functional requirements define the expected actions of the system, the domain model establishes the semantic structure within which these actions acquire logical, business, and architectural justification. Therefore, it is necessary to consider not only the differences between these two forms of representation, but also the specific functions performed by the domain model in the formalization of requirements (table 2).

Table 2. *Key functions of the domain model in the semantic formalization of requirements (Nandi & Dey, 2025; D'Anna, 2026)*

Function of the domain model	Content of the function	Significance for IT product design
Semantic structuring	Defines the key concepts, entities, relationships, states, and constraints of the subject area.	Forms a unified understanding of product logic among business, analytical, and engineering teams.
Formalization of business rules	Records rules, exceptions, permissible operations, and transitions between system states.	Reduces the risk of implicit, fragmented, or contradictory interpretation of requirements.
Context delimitation	Separates areas of responsibility through bounded contexts and clarifies the meaning of entities within each context.	Supports more accurate decomposition of modules, services, data models, and architectural boundaries.
Alignment of requirements and architecture	Links business logic with software components, interfaces, APIs, and interaction patterns.	Ensures that architectural decisions are derived from domain logic rather than isolated technical assumptions.
Support for system evolution	Makes it possible to assess how changes in business processes affect entities, rules, contexts, and technical components.	Increases maintainability and adaptability during further product development.

The functions presented in Table 2 show that the domain model acts as a methodological mechanism for translating domain semantics into architectural structure. It defines how business rules, constraints, and contextual meanings should be preserved during implementation and further product evolution.

The relevance of such formalization is confirmed by recent research in product management and business analysis. The ProductPlan State of Product Management 2025 report, based on data from nearly 400 product professionals from different countries, indicates that product strategy is considered one of the most significant responsibilities of product managers, while 59% of teams are focused on consolidating product management tools (ProductPlan, 2025). These data show that modern IT product development increasingly requires a unified management and analytical framework in which strategy, roadmap, prioritization, and requirements are interconnected rather than treated as separate project artifacts.

The importance of the domain model also increases as product development becomes more distributed and cross-functional. According to Atlassian State of Product 2026, which surveyed more than 1,000 product professionals in the United States and Europe, 85% of product teams believe that they influence product strategy; however, 49% report a lack of time for strategic planning and roadmap development, and another 49% indicate insufficient time for data analysis and metric tracking (Crusson, 2025). At the same time, 84% of teams express concern that their current products may not succeed in the market. These findings demonstrate that the growing strategic role of product teams does not eliminate the need for semantic coordination. Under conditions of high organizational workload and limited analytical capacity, teams need a shared model of the subject area that establishes a common language for discussing the product and reduces dependence on fragmented interpretations of requirements.

Architectural effects of DDD

The architectural significance of DDD lies in its ability to align the technical structure of an IT product with the stable semantic boundaries of the subject area. In this approach, modules, services, data models, and interfaces are designed not as isolated technical elements, but as representations of domain entities, rules, and processes. This

reduces the risk of dispersing business logic across different system layers and creates a basis for more manageable architectural decomposition (table 3).

Table 3. *Architectural effects of DDD in IT product development (Lytvynov et al., 2026; Özkan et al., 2023)*

Architectural effect	Content of the effect	Significance for an IT product
Decomposition by semantic boundaries	Modules and services are structured according to bounded contexts of the subject area.	Reduces the risk of arbitrary system division based only on technical layers or isolated functions.
Localization of business logic	Business rules, states, constraints, and domain operations are placed within the relevant domain components.	Prevents the dispersion of business logic across the user interface, database layer, and integration services.
Reduction of component coupling	Acceptable dependencies between contexts, services, and data models are explicitly defined.	Improves the controllability of changes and reduces the probability of cascading modifications.
Architectural resilience	Technical decisions are aligned with the long-term logic of domain development.	Allows the product to adapt to new requirements without disrupting previously established architectural decisions.
Limitation of technical debt	Implicit dependencies, duplicated business rules, and uncontrolled growth of logic are reduced.	Decreases the costs of maintenance, modernization, and further system development.
Improved maintainability	Areas of responsibility between components are defined more clearly.	Simplifies change analysis, functional extension, and long-term product support.

Thus, the architectural effects of DDD are manifested not only in increased system modularity, but also in a more accurate alignment of the technical structure of the product with the logic of the subject area. The identification of semantic boundaries, the localization of business rules, and the reduction of component coupling make it possible to view architecture as a manageable system of domain contexts rather than as a set of fragmented technical decisions. As a result, the predictability of changes increases, the risk of technical debt accumulation decreases, and conditions are created for the long-term development of an IT product without disrupting its internal logic.

Applied interpretation of the approach in the design of complex digital systems

The applied significance of DDD is most evident in systems where architecture must simultaneously reflect business goals, operational constraints, scalability requirements, data security, and long-term maintainability. In such conditions, sustainable design cannot be reduced to the choice of a technology stack, database, or integration protocol. The decisive factor is the correspondence between the technical structure and the subject-area logic; otherwise, business rules become duplicated, changes remain local and inconsistent, and the product gradually loses architectural controllability.

In practice, DDD enables a transition from function-centered development to domain-boundary-based design, in which each system component is associated with a specific domain context. This is particularly important when designing digital products that include user management, billing, application processing, analytics, notifications, compliance, and integrations with external platforms. Each of these contexts has its own data model, set of business rules, and correctness criteria. Therefore, domain decomposition makes it possible to more

precisely define data ownership, the location of business rules, permissible dependencies between services, and the boundaries of changes that can be implemented locally without affecting the entire system.

The practical significance of correctly identifying domain boundaries is confirmed by the results of an empirical evaluation of DDD in the modular architecture of an information system. In a 2025 study presented in the IEEE ICoCSETI proceedings, the domain-driven approach was analyzed using an academic information system as a case study and assessed through metrics of modularity, coupling, granularity, and stability. The evaluation was aimed not at an abstract description of DDD, but at measuring the extent to which domain decomposition contributes to the formation of a more manageable software system structure (table 4).

Table 4. *Results of modularity evaluation in the application of DDD to an academic information system (Putra et al., 2025)*

Metric	Obtained result	Interpretation for DDD
Service Granularity Metric, SGM	0.18	Indicates the need to assess whether the selected service granularity corresponds to domain boundaries.
Average Lack of Cohesion Metric, ALCOM	0.14	Reflects the degree of insufficient cohesion within the modular structure.
Relational Cohesion, H	0.38; 0.35; 0.21	Shows that internal relationships between types within individual modules require further strengthening.
Number of Operations, NOO	Used to assess the operational density of modules	Helps identify components that are either overloaded with operations or excessively fragmented.
Instability, I	Used to analyze module stability in relation to dependencies	Makes it possible to assess the risk of architectural instability when changes are introduced.

The obtained results show that DDD is most effective when domain decomposition is complemented by quantitative assessment of modularity, granularity, and component cohesion. Its practical value lies not only in identifying domain contexts, but also in verifying whether the proposed service boundaries correspond to the actual structural characteristics of the system. In this sense, the domain model functions not only as a means of describing business logic, but also as an instrument of architectural validation: it makes it possible to compare intended service boundaries with measurable system metrics and to identify areas where component responsibilities should be refined.

The practical relevance of this approach is illustrated by engineering cases from large technology companies. **Uber** describes its platform principle as the use of modular building blocks that allow different business verticals – rides, eats, freight, and autonomous – to operate on a shared infrastructure. In the Unified Checkout case, the company centralized payment orchestration across business lines, which reduced duplication of payment logic and resulted in a 3% increase in conversion, a 4.5% improvement in session recovery, and hundreds of millions of dollars in additional gross bookings (Fiali de Sá, 2024). In another case, related to executive travel, Uber introduced a participant model covering booking, tracking, and billing across more than 30 backend services and 5 client platforms. These examples show that the domain model can serve as a practical mechanism for aligning business scenarios, service boundaries, and reusable architectural components.

A similar principle can be observed in **Square**'s engineering practice, where the Mobile Developer Experience team treated build logic as an independent engineering domain

with its own model, responsibility boundaries, and testable rules. During the modernization of the Point of Sale Android repository, the company formalized the build model through convention plugins and separated the build domain from the feature domain. The quantitative results demonstrate the practical effect of this approach: the cyclomatic complexity of the root build script was reduced from 41 to 16, or by 61%, while the complex square_gradle_module.gradle script was reduced from 75 to 0 through removal (Robalik, 2023). At the same time, the repository grew from approximately 3 million to more than 4 million lines of code and from 3,500 to almost 4,400 Gradle modules, while total local build time did not increase significantly. This case shows that domain-oriented thinking is applicable not only to user-facing business logic, but also to internal engineering platforms, where model formalization reduces complexity and improves maintainability.

Conclusion

Thus, DDD makes it possible to consider IT product architecture as the result of the consistent formalization of the subject area rather than as a set of isolated technical decisions. In this context, the domain model performs a coordinating function between business requirements, analytical descriptions of processes, and software implementation. It reduces ambiguity in the interpretation of requirements and increases the consistency of design decisions by linking product logic with architectural structure.

The analysis shows that DDD is especially significant for complex digital systems that evolve under conditions of changing business rules, user scenarios, technological constraints, and increasing requirements for maintainability. The identification of bounded contexts, the formalization of domain entities, and the specification of interaction rules create a basis for a more resilient and modular architecture. Such an architecture can adapt to further changes while preserving the internal logic of the IT product and maintaining alignment between business meaning and technical implementation.

References

Crusson, T. (2025). The State of Product in 2026: Navigating Change, Challenge, and Opportunity. *Atlassian*. [cited: 15.05.2026]. Available from URL: https://www.atlassian.com/blog/announcements/state-of-product-2026

D'Anna, D. (2026). Domain-Driven Design (DDD). In Building Distributed Systems with Go and NATS: A Comprehensive Guide. *Berkeley, CA: Apress*, – P. 171–215. URL: https://doi.org/10.1007/979–8–8688–2089–2_6

Fiali de Sá, F. (2024). Unified Checkout: Streamlining Uber's Payment Ecosystem. *Uber*. [cited: 18.05.2026]. Available from URL: https://www.uber.com/us/en/blog/unified-checkout/

Kovalevskyi, K. (2026). Impact of architectural optimization of distributed systems on reducing operating costs and improving the economic efficiency of digital services. *International Journal of Engineering in Computer Science*, – 8 (2). – P. 63–67. URL: https://doi.org/10.33545/26633582.2026.v8.i2a.278

Kumar, A., Gupta, L., Alam, S., Khanna, P., & Raman, A. (2025). Bridging the Gap Between Problem and Solution Space with Domain-Driven Design. *World Conference on Information Systems and Technologies. Cham: Springer Nature Switzerland*, – P. 179–192. URL: https://doi.org/10.1007/978-3-032-01234-0_16

Lytvynov, O. A., Khandetskyi, V. S., & Lytvynov, M. O. (2026). Estimation of effort of migration among domain-driven design architectural variations. *Radio Electronics*, Computer Science, Control, – 1. – P. 159–175. URL: https://doi.org/10.15588/1607-3274-2026-1-14

Mordor Intelligence. (2026). Software development market size & share analysis - growth trends and forecast (2026–2031). *Mordor Intelligence*. [cited: 14.05.2026]. Available from URL: https://www.mordorintelligence.com/industry-reports/software-development-market

Nandi, K., & Dey, K. (2025) Designing scalable multi-agent AI systems: Leveraging domain-driven design and event storming. *International Journal of Computer Science and*

Engineering, – 12 (3). – P. 10–16. URL: https://doi.org/10.14445/23488387/IJCSE-V12I3P102

Özkan, O., Babur, Ö., & van den Brand, M. (2023). Refactoring with domain-driven design in an industrial context: An action research report. *Empirical Software Engineering*, – 28 (4). – 94 p. URL: https://doi.org/10.1007/s10664-023-10310-1

ProductPlan. (2025). The 2025 State of Product Management Report. *ProductPlan*. [cited: 15.05.2026]. Available from URL: https://www.productplan.com/ebooks/the-2025-state-of-product-management-report

Putra, Z. L., Sarno, R., Septiyanto, A. F., Akbar, R. J., & Taufany, F. (2025). Evaluating The Modularity of Domain-Driven Design Approach: A Case Study of Academic Information System. 2025 *International Conference on Computer Sciences*, Engineering, and Technology Innovation (ICoCSETI). IEEE, – P. 507–512. URL: https://doi.org/10.1109/ICoCSETI63724.2025.11019730

Robalik, T. (2023). Stampeding Elephants: The data-driven case for best practices and against silver bullets. *Square*. [cited: 18.05.2026]. Available from URL: https://developer.squareup.com/blog/stampeding-elephants

submitted 27.06.2026;
accepted for publication 11.07.2026;
published 31.07.2026

Contact: kolesnikovadaria32@gmail.com