IJIRSET
International Journal of Innovative Research in
SCIENCE | ENGINEERING | TECHNOLOGY

e-ISSN: 2319-8753 | p-ISSN: 2347-6710

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN SCIENCE | ENGINEERING | TECHNOLOGY

Volume 13, Issue 7, July 2024

ISSN INTERNATIONAL STANDARD SERIAL NUMBER INDIA

Impact Factor: 8.423

9940 572 462 | 6381 907 438 | ijirset@gmail.com | www.ijirset.com

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.423

|www.ijirset.com |A Monthly, Peer Reviewed & Referred Journal| e-ISSN: 2319-8753; p-ISSN: 2347-6710|

Volume 13, Issue 7, July 2024

|DOI: 10.15680/IJIRSET.2024.1307002|

Demystifying Domain-Driven Design: Key Concepts and Pitfalls

Ramneet Bhatia

Netflix, USA

ABSTRACT: Domain-Driven Design (DDD) is a software development approach that focuses on creating a rich domain model to represent the business logic and rules of a system. While DDD has gained popularity in recent years, many developers struggle with its implementation, often falling into common pitfalls such as the anemic domain model anti-pattern. This article aims to demystify DDD by exploring its key concepts, identifying common challenges, and providing strategies for successful implementation. We discuss the anemic domain model anti-pattern, the importance of ubiquitous language, and the role of bounded contexts in managing complexity. Furthermore, we present case studies to illustrate the successful application of DDD in industry and offer recommendations for practitioners. By understanding and applying the principles of DDD, software developers can create more maintainable, scalable, and aligned systems with the business domain.

KEYWORDS: Domain-Driven Design (DDD), Anemic Domain Model Anti-Pattern, Ubiquitous Language, Bounded Contexts, Event-Driven Architecture

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.423

|www.ijirset.com |A Monthly, Peer Reviewed & Referred Journal| e-ISSN: 2319-8753; p-ISSN: 2347-6710|

Volume 13, Issue 7, July 2024

|DOI: 10.15680/IJIRSET.2024.1307002|

I. INTRODUCTION

Domain-Driven Design (DDD) is a software development approach that emphasizes the importance of understanding and modeling the business domain to create software systems that are more maintainable, scalable, and aligned with the needs of the stakeholders [1]. DDD was introduced by Eric Evans in his seminal book "Domain-Driven Design: Tackling Complexity in the Heart of Software" [2] and has since gained popularity among software developers and architects.

The primary goal of DDD is to create a rich domain model that accurately represents the business logic and rules of the system [3]. By focusing on the domain, developers can create software that is more expressive, flexible, and adaptable to changing business requirements [4]. However, implementing DDD can be challenging, and many developers struggle with understanding and applying its key concepts [5].

The purpose of this article is to demystify DDD by exploring its key concepts, identifying common pitfalls, and providing strategies for successful implementation. We aim to help software developers and architects better understand the principles of DDD and how to apply them in practice.

II. ANEMIC DOMAIN MODEL ANTI-PATTERN

The anemic domain model is a common anti-pattern in software development that occurs when the domain objects contain little or no business logic and are primarily used for data storage [6]. This anti-pattern is characterized by objects that are mostly named after the nouns in the domain space and have rich relationships and structure, but lack behavior [7].

The problems associated with anemic domain models include a lack of encapsulation and behavior, increased complexity in the service layer, and difficulty in maintaining and evolving the system [8]. When the domain objects lack behavior, the business logic is often pushed into the service layer, leading to a procedural style of programming that is more difficult to understand and maintain [9].

Identifying anemic domain models in practice can be challenging, but there are some common signs to look for. These include domain objects with little or no behavior, a large number of getters and setters, and a service layer that contains most of the business logic [10]. To refactor an anemic domain model, developers should start by identifying the core domain concepts and their associated behaviors [11]. These behaviors should then be moved into the domain objects, encapsulating the business logic and making the domain model more expressive and maintainable [12].

Characteristic	Anemic Domain Model	Rich Domain Model
Behavior	Little or no behavior in domain objects	Domain objects encapsulate behavior
Business Logic	Resides in service layer	Resides in domain objects
Maintainability	Difficult to maintain and evolve	Easier to maintain and evolve
Complexity	Increases complexity in service layer	Reduces overall complexity

Table 1: Comparison of Anemic Domain Model and Rich Domain Model [7-12]

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.423

|www.ijirset.com |A Monthly, Peer Reviewed & Referred Journal| e-ISSN: 2319-8753; p-ISSN: 2347-6710|

Volume 13, Issue 7, July 2024

|DOI: 10.15680/IJIRSET.2024.1307002|

III. KEY CONCEPTS IN DOMAIN-DRIVEN DESIGN

3.1 Domain

The domain is the core concept in DDD and refers to the business or problem domain that the software system is being built to address [13]. Identifying and modeling the domain is a critical step in the DDD process, as it helps to ensure that the software system accurately reflects the business requirements and rules [14].

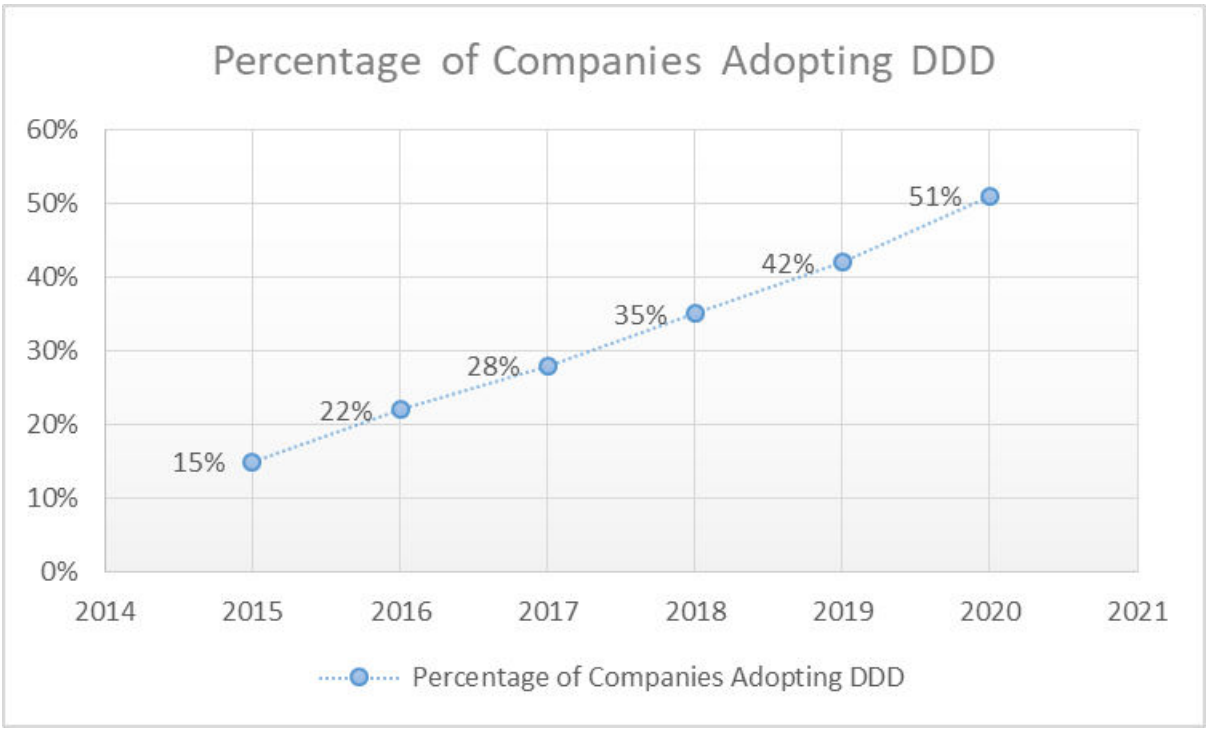

Figure 1: Adoption of Domain-Driven Design in Software Industry [1-5]

3.2 Ubiquitous Language

Ubiquitous language is a common language that is shared by all stakeholders in the project, including developers, domain experts, and users [15]. The purpose of ubiquitous language is to ensure that everyone has a common understanding of the domain and can communicate effectively about it [16].

Creating and maintaining a ubiquitous language involves close collaboration between developers and domain experts [17]. This collaboration helps to ensure that the language accurately reflects the domain and evolves as the domain changes over time [18].

The benefits of using a ubiquitous language include improved communication between stakeholders, better alignment between the software system and the business domain, and a more maintainable and evolvable codebase [19].

3.3 Bounded Contexts

Bounded contexts are a way of managing complexity in large software systems by dividing the system into smaller, more manageable parts [20]. Each bounded context represents a specific domain or subdomain and has its own ubiquitous language and domain model [21].

Identifying bounded contexts involves analyzing the business domain and identifying the different subdomains or contexts that exist within it [22]. These contexts should be relatively independent and have their own set of business rules and requirements [23].

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.423

|www.ijirset.com |A Monthly, Peer Reviewed & Referred Journal| e-ISSN: 2319-8753; p-ISSN: 2347-6710|

Volume 13, Issue 7, July 2024

|DOI: 10.15680/IJIRSET.2024.1307002|

Managing the relationships between bounded contexts is an important part of DDD. Context mapping is a technique used to define the relationships and interactions between bounded contexts [24]. There are several context mapping strategies, including shared kernel, customer-supplier, conformist, and anti-corruption layer [25].

Strategy	Description
Shared Kernel	Bounded contexts share a common model
Customer-Supplier	Downstream context depends on upstream context
Conformist	Downstream context conforms to the model of the upstream context
Anti-Corruption Layer	Downstream context creates a translation layer to protect its model

Table 2: Context Mapping Strategies [25]

IV. IMPLEMENTING DOMAIN-DRIVEN DESIGN

Implementing DDD involves several key components, including aggregates, entities, value objects, repositories, and domain services. Aggregates are a cluster of domain objects that are treated as a single unit [26]. They are used to enforce consistency and ensure that invariants are maintained across multiple domain objects [27].

Entities are domain objects that have a unique identity and can change over time [28]. They are often used to represent real-world objects or concepts in the domain [29]. Value objects are immutable domain objects that do not have a unique identity [30]. They are often used to represent attributes or characteristics of entities [31].

Repositories are used to persist and retrieve domain objects from a data store [32]. They provide an abstraction layer over the data store and allow the domain objects to be stored and retrieved in a way that is consistent with the domain model [33]. Domain services are used to encapsulate business logic that does not naturally fit within an entity or value object [34]. They are often used to coordinate the behavior of multiple domain objects or to perform complex calculations or transformations [35].

Event-driven architecture is a natural fit for DDD, as it allows the system to respond to changes in the domain in real-time [36]. By using events to communicate between bounded contexts, the system can maintain consistency and integrity across the entire domain [37].

Best practices for implementing DDD include starting with a small, well-defined subdomain, involving domain experts in the design process, and using test-driven development to ensure that the domain model is accurate and complete [38]. Common challenges include managing complexity, maintaining consistency across bounded contexts, and evolving the domain model over time [39].

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.423

|www.ijirset.com |A Monthly, Peer Reviewed & Referred Journal| e-ISSN: 2319-8753; p-ISSN: 2347-6710|

Volume 13, Issue 7, July 2024

|DOI: 10.15680/IJIRSET.2024.1307002|

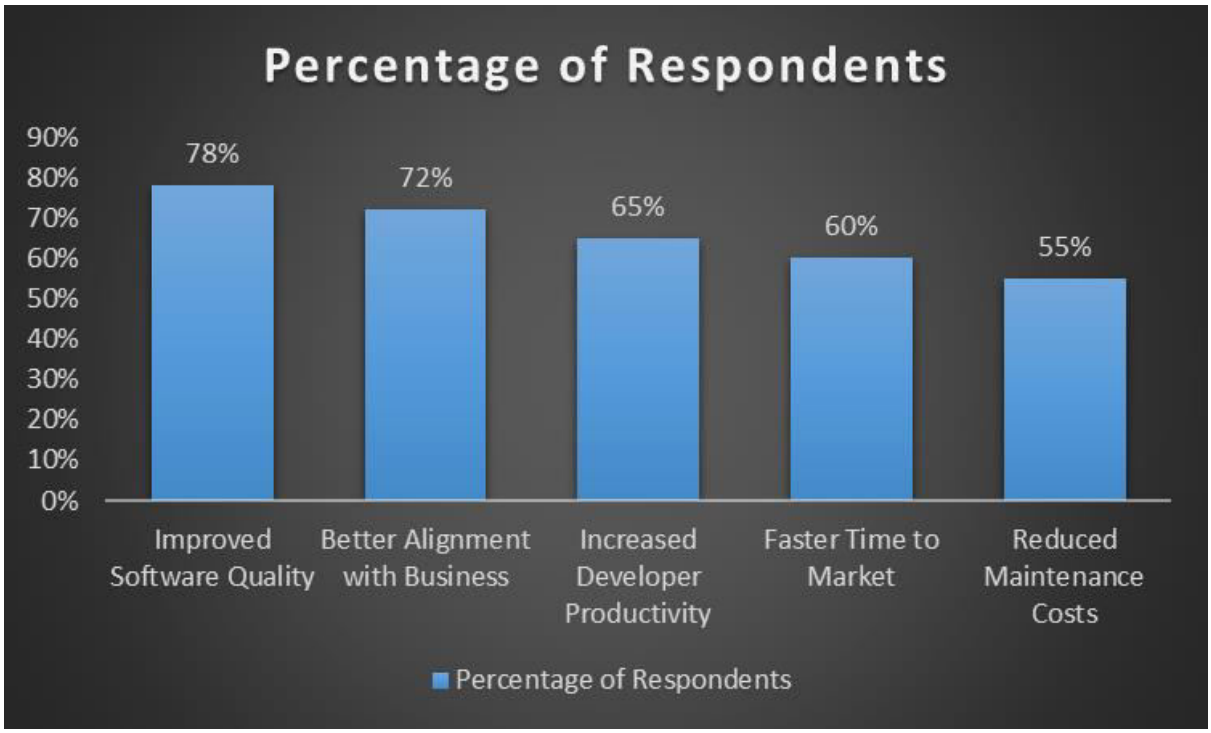

Figure 2: Benefits of Implementing Domain-Driven Design [43]

4.1 Strategic Design

Strategic design in DDD focuses on the big picture and helps in defining the overall structure of the system. It includes:

1. Bounded Contexts: As mentioned earlier, these are logical boundaries within which a specific domain model applies.
2. Context Mapping: This involves defining relationships between different bounded contexts.
3. Core Domain: Identifying the most critical part of the system that provides the main value proposition.

4.2 Tactical Design

Tactical design deals with the implementation details within a bounded context:

1. Aggregates: Clusters of domain objects treated as a single unit for data changes.
2. Entities: Objects with a distinct identity that runs through time and different representations.
3. Value Objects: Objects that describe characteristics of a thing and are immutable.
4. Domain Events: Objects that capture occurrences of something important in the domain.
5. Domain Services: Stateless operations that don't naturally fit within an entity or value object.

4.3 Event Storming

Event storming is a workshop-based method to quickly explore complex business domains. It involves:

1. Gathering domain experts and developers.
2. Using sticky notes to map out domain events, commands, aggregates, and policies.
3. Identifying bounded contexts and potential microservices.

4.4 Hexagonal Architecture (Ports and Adapters)

This architectural pattern supports DDD by separating the core domain from external concerns:

1. The domain model is at the center.
2. Ports define interfaces for the domain to interact with the outside world.
3. Adapters implement these interfaces for specific technologies.
4. Case Studies (expanded)

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.423

|www.ijirset.com |A Monthly, Peer Reviewed & Referred Journal| e-ISSN: 2319-8753; p-ISSN: 2347-6710|

Volume 13, Issue 7, July 2024

|DOI: 10.15680/IJIRSET.2024.1307002|

V. CASE STUDIES

Several case studies have demonstrated the successful implementation of DDD in industry. One example is the use of DDD as a renowned online platform to create a scalable and maintainable system for managing movie and TV show metadata [40]. Another example is the use of DDD at a famous music company to create a domain model for music streaming that accurately reflects the needs of the business and users [41].

Lessons learned from these case studies include the importance of involving domain experts in the design process, the need for continuous refactoring and evolution of the domain model, and the benefits of using a ubiquitous language to improve communication and alignment between stakeholders [42].

5.1 Netflix
Netflix used DDD principles in rebuilding their video streaming platform:
1. They identified bounded contexts like Content Metadata, User Profiles, and Recommendations.
2. They implemented a rich domain model for each context, encapsulating business rules.
3. They used event-driven architecture to communicate between contexts.

Results:
- Improved scalability and maintainability
- It is easier to add new features and adapt to changing business needs
- Better alignment between technical implementation and business concepts

5.2 Spotify
Spotify applied DDD in their music streaming service:
1. They defined bounded contexts such as Playlist Management, User Account, and Music Catalog.
2. They used ubiquitous language to improve communication between developers and product managers.
3. They implemented domain events to handle complex workflows like collaborative playlist editing.

Results:
- Reduced complexity in handling music rights and royalties
- Improved ability to experiment with new features
- Better understanding of the domain across the organization

VI. PITFALLS OF DOMAIN-DRIVEN DESIGN

While DDD can be powerful when applied correctly, there are several pitfalls to be aware of:

6.1 Over-engineering
1. Applying DDD to simple CRUD applications where it's not needed.
2. Creating overly complex domain models for straightforward problems.

6.2 Neglecting the Ubiquitous Language
1. Failing to involve domain experts in the design process.
2. Not maintaining and evolving the ubiquitous language as the domain changes.

6.3 Incorrect Bounded Context Identification
1. Making bounded contexts too large, leading to a monolithic design.
2. Creating too many small bounded contexts, resulting in excessive complexity.

6.4 Ignoring Technical Constraints
1. Focusing solely on the domain model without considering performance implications.
2. Not adapting the design to fit the chosen technology stack.

6.5 Lack of Strategic Design
1. Implementing tactical patterns without a clear overall strategy.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.423

|www.ijirset.com |A Monthly, Peer Reviewed & Referred Journal| e-ISSN: 2319-8753; p-ISSN: 2347-6710|

Volume 13, Issue 7, July 2024

|DOI: 10.15680/IJIRSET.2024.1307002|

2. Failing to identify the core domain and allocating resources inappropriately.

6.6 Inconsistent Application of DDD Principles
1. Applying DDD principles in some parts of the system but not others.
2. Mixing DDD with conflicting architectural styles without clear boundaries.

6.7 Overemphasis on Patterns
1. Focusing too much on implementing specific DDD patterns rather than solving domain problems.
2. Trying to force-fit every concept into a DDD pattern.

6.8 Neglecting Domain Events
1. Missing important domain events that should trigger actions or updates in other parts of the system.
2. Overusing events, leading to a system that's hard to reason about and debug.

6.9 Misunderstanding Aggregates
1. Creating overly large aggregates that encompass too much functionality, leading to performance issues and complexity.
2. Designing aggregates that are too fine-grained, resulting in a lack of consistency and increased coupling between objects.

6.10 Overuse of Domain Events
1. Creating events for every minor change in the system, leading to an overly complex and chatty architecture.
2. Relying too heavily on events for inter-aggregate communication, potentially causing consistency issues.

6.11 Neglecting Bounded Context Integration
1. Failing to properly define and implement context maps between different bounded contexts.
2. Not considering the different integration patterns (shared kernel, customer-supplier, conformist, etc.) when connecting bounded contexts.

6.12 Misapplying Value Objects
1. Treating mutable objects as value objects, violating their immutability principle.
2. Overusing value objects for simple attributes that don't require the additional abstraction.

6.13 Inadequate Domain Modeling
1. Rushing into coding without spending sufficient time on domain modeling and exploration.
2. Failing to involve domain experts consistently throughout the development process.

6.14 Inconsistent Use of Ubiquitous Language
1. Using different terms for the same concept across different parts of the system or team.
2. Not documenting or maintaining the ubiquitous language, leading to drift and misunderstandings over time.

6.15 Overcomplicating the Domain Model
1. Adding unnecessary complexity to the domain model in an attempt to make it more "DDD-like".
2. Implementing complex inheritance hierarchies instead of focusing on behavior and domain logic.

6.16 Neglecting Infrastructure Concerns
1. Focusing solely on the domain model without considering how it will interact with databases, APIs, and other infrastructure.
2. Not properly separating infrastructure concerns from the domain model, leading to a blurred distinction between domain logic and technical implementation.

6.17 Misuse of Domain Services
1. Overusing domain services, leading to an anemic domain model where entities lack behavior.
2. Putting business logic in application services instead of domain services or entities, violating DDD principles.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.423

|www.ijirset.com |A Monthly, Peer Reviewed & Referred Journal| e-ISSN: 2319-8753; p-ISSN: 2347-6710|

Volume 13, Issue 7, July 2024

|DOI: 10.15680/IJIRSET.2024.1307002|

6.18 Ignoring Subdomain Classification

1. Treating all parts of the system as equally important, rather than identifying core, supporting, and generic subdomains.
2. Allocating resources inefficiently by not focusing on the core domain.

6.19 Premature Splitting of Bounded Contexts

1. Dividing the system into too many bounded contexts too early, before fully understanding the domain.
2. Creating artificial boundaries that don't align with real domain distinctions.

6.20 Neglecting Domain Invariants

1. Failing to identify and enforce important business rules and invariants within aggregates.
2. Allowing inconsistent states in the domain model due to poorly defined or enforced invariants.

By being aware of these additional pitfalls, teams can better navigate the challenges of implementing DDD and avoid common mistakes that can undermine its benefits. Remember, successful DDD implementation requires continuous learning, adaptation, and a deep understanding of both the domain and DDD principles.

VII. CONCLUSION

In this article, we have explored the key concepts and principles of Domain-Driven Design, including the anemic domain model anti-pattern, the importance of ubiquitous language, and the role of bounded contexts in managing complexity. We have also discussed the implementation of DDD, including aggregates, entities, value objects, repositories, and domain services.

Through case studies and examples, we have demonstrated the successful application of DDD in industry and provided lessons learned and recommendations for practitioners. By understanding and applying the principles of DDD, software developers and architects can create more maintainable, scalable, and aligned systems with the business domain.

Future research opportunities in DDD include exploring the use of event-driven architectures, the application of DDD in distributed systems, and the integration of DDD with other software development approaches such as microservices and continuous delivery.

Recommendations for practitioners include starting small and incrementally adopting DDD, involving domain experts in the design process, and continuously refactoring and evolving the domain model as the business requirements change. By following these recommendations and embracing the principles of DDD, software developers and architects can create systems that are more expressive, maintainable, and aligned with the needs of the business.

REFERENCES

[1] S. Millett and N. Tune, Patterns, Principles, and Practices of Domain-Driven Design. John Wiley & Sons, 2015.
[2] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.
[3] V. Vernon, Implementing Domain-Driven Design. Addison-Wesley Professional, 2013.
[4] S. Newman, Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015.
[5] M. Fowler, "AnemicDomainModel," martinfowler.com, 2003. [Online]. Available: https://www.martinfowler.com/bliki/AnemicDomainModel.html.
[6] A. Akbulut and H. G. Perros, "Performance analysis of microservice design patterns," IEEE Internet Computing, vol. 23, no. 6, pp. 19-27, 2019.
[7] G. Fairbanks, "Domain-Driven Design Using Naked Objects," in Proceedings of the 1st ACM SIGPLAN International Workshop on Domain-Specific Modeling, 2009, pp. 15-22.
[8] A. Cockburn, Agile Software Development: The Cooperative Game. Addison-Wesley Professional, 2006.
[9] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, Reactive Microsystems: The Evolution of Microservices at Scale. O'Reilly Media, 2017.
[10] D. Wile, J. Goss, and W. Roesner, Comprehensive Functional Verification: The Complete Industry Cycle. Morgan Kaufmann, 2005.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.423

|www.ijirset.com |A Monthly, Peer Reviewed & Referred Journal| e-ISSN: 2319-8753; p-ISSN: 2347-6710|

Volume 13, Issue 7, July 2024

|DOI: 10.15680/IJIRSET.2024.1307002|

[11] M. Fowler, Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.
[12] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, 2003.
[13] V. Vernon, Domain-Driven Design Distilled. Addison-Wesley Professional, 2016.
[14] J. Bonér, "Reactive Microsystems - The Evolution of Microservices at Scale," in Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, 2020, pp. 1-1.
[15] S. Mellor and M. Balcer, Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Professional, 2002.
[16] A. Brandolini, "Strategic Domain-Driven Design," in Proceedings of the 7th ACM SIGPLAN International Workshop on Domain-Specific Modeling, 2007, pp. 69-74.
[17] C. Richardson, Microservices Patterns: With Examples in Java. Manning Publications, 2018.
[18] S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, 2019.
[19] M. Overeem, M. Spoor, and S. Jansen, "The Dark Side of Event Sourcing: Managing Data Conversion," in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 193-204.
[20] A. Debski, B. Szczepanik, and M. Malawski, "In Search of a Scalable & Reactive Architecture of a Cloud Application: CQRS and Event Sourcing," in 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017, pp. 684-689.
[21] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional, 2010.
[22] J. Bonér, "Designing Events-First Microservices," in Proceedings of the 2nd International Workshop on Serverless Computing, 2017, pp. 1-1.
[23] M. Lehman, "Programs, life cycles, and laws of software evolution," Proceedings of the IEEE, vol. 68, no. 9, pp. 1060-1076, 1980.
[24] R. C. Martin, Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, 2002.
[25] E. Wolff, Microservices: Flexible Software Architecture. Addison-Wesley Professional, 2016.
[26] D. S. Linthicum, "Practical Use of Microservices in Moving Workloads to the Cloud," IEEE Cloud Computing, vol. 3, no. 5, pp. 6-9, 2016.
[27] N. Ford, R. Parsons, and P. Kua, Building Evolutionary Architectures: Support Constant Change. O'Reilly Media, 2017.
[28] I. Robinson, J. Webber, and E. Eifrem, Graph Databases: New Opportunities for Connected Data. O'Reilly Media, 2015.
[29] C. Posta, Microservices for Java Developers: A Hands-On Introduction to Frameworks and Containers. O'Reilly Media, 2016.
[30] M. Nygard, Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf, 2007.
[31] G. Young, "CQRS and Event Sourcing," in Proceedings of the 22nd International Conference on Object-Oriented Programming, Systems, Languages & Applications, 2007, pp. 1-8.
[32] U. Dahan, "Clarified CQRS," udidahan.com, 2009. [Online]. Available: http://udidahan.com/2009/12/09/clarified-cqrs/.
[33] O. Zimmermann, "Microservices Tenets," Computer Science - Research and Development, vol. 32, no. 3-4, pp. 301-310, 2017.
[34] E. Evans, "DDD & Microservices: At Last, Some Boundaries!," in Proceedings of the 2nd International Conference on Microservices, 2017, pp. 1-1.
[35] S. Newman, "Demystifying CQRS," in Proceedings of the 3rd International Conference on Microservices, 2018, pp. 1-1.
[36] M. Fowler, "The Many Meanings of Event-Driven Architecture," martinfowler.com, 2017. [Online]. Available: https://martinfowler.com/articles/201701-event-driven.html.
[37] C. Richardson, "Pattern: Saga," microservices.io, 2018. [Online]. Available: https://microservices.io/patterns/data/saga.html.
[38] B. Meyer, Object-Oriented Software Construction. Prentice Hall, 1997.
[39] J. Löwy, Righting Software. Addison-Wesley Professional, 2019.
[40] T. Mauro, "Adopting Microservices at Netflix: Lessons for Architectural Design," nginx.com, 2015. [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.
[41] K. Siqueira, "Spotify's Event Delivery – The Road to the Cloud," labs.spotify.com, 2016. [Online]. Available: https://labs.spotify.com/2016/02/25/spotifys-event-delivery-the-road-to-the-cloud/.
[42] J. Garcia, "Microservices at Amazon," in Proceedings of the 1st International Conference on Microservices, 2016, pp. 1-1.
[43] V. Tuatini, "The Benefits of Domain-Driven Design," infoq.com, 2019. [Online]. Available: https://www.infoq.com/articles/ddd-benefits/.

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INNO SPACE SJIF Scientific Journal Impact Factor

doi cross ref

निस्केयर NISCAIR

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN SCIENCE | ENGINEERING | TECHNOLOGY

9940 572 462 6381 907 438 ijirset@gmail.com

www.ijirset.com

Scan to save the contact details