UDC 614.2+574/578+004.38
DOI https://doi.org/10.32782/IT/2024-1-7

Oleksandr LYTVYNOV
Candidate of Technical Sciences, Associate Professor, Faculty of Physics, Electronics and Computer Systems, Oles Honchar Dnipro National University, 72, Haharina ave., Dnipro, Ukraine, 49000, lytvynov@ffeks.dnu.edu.ua
***ORCID:** 0000-0001-7660-1353*

Dmytro HRUZIN
Master, Postgraduate Student, Faculty of Physics, Electronics and Computer Systems, Oles Honchar Dnipro National University, 72, Haharina ave., Dnipro, Ukraine, 49000, hruzin_dl@ffeks.dnu.edu.ua.
***ORCID:** 0009-0004-8534-2559*

Maksym FROLOV
Master, Faculty of Physics, Electronics and Computer Systems, Oles Honchar Dnipro National University, 72, Haharina ave., Dnipro, Ukraine, 49000, frolov_mo@ffeks.dnu.edu.ua.
***ORCID:** 0009-0000-6624-6028*

To cite this article: Lytvynov. O., Hruzin, D., Frolov, M. (2024). Pro mihratsiiu proektu z Domain Driven Design arkhitekturoiu na CQRS ta Event Sourcing [On the migration of Domain Driven Design to CQRS with Event Sourcing software architecture]. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 1, 50–60, doi: https://doi.org/10.32782/IT/2024-1-7

ON THE MIGRATION OF DOMAIN DRIVEN DESIGN TO CQRS WITH EVENT SOURCING SOFTWARE ARCHITECTURE

The article addresses the issue of migrating applications, particularly those following the Domain-Driven Design (DDD) architecture, to the Command Query Responsibility Segregation (CQRS) paradigm with Event Sourcing. Long-standing systems often need help with problems related to inflexible, outdated architecture, and dependencies, leading to increased maintenance costs. The paper examines the advantages of DDD and proposes CQRS as a viable alternative, focusing on improving productivity and scalability.

*The **main objective** of the work is to assess a secure path for migrating a project from DDD architecture to the CQRS and Event Sourcing architecture and to determine the migration roadmap. The article conducts an experiment in which migration of a test project is performed, evaluating the time, effort, and results of the migration. The research methodology includes evaluating complexity using McCabe's Cyclomatic Complexity metric and assessing performance through the execution time of system methods.*

The experiment is conducted on a typical project – a task-tracking system. The results of implementing CQRS show a fourfold increase in the number of classes and a 50% increase in the number of lines of code. However, this increase is justified as it improves modularity, transparency, and manageability during development, ultimately facilitating system maintenance and significantly enhancing overall system productivity. It is worth noting that the overall cyclomatic complexity of the system remains almost unchanged.

***In summary,** the article examines the assessment of migrating a project from DDD architecture to CQRS and Event Sourcing, combining theoretical findings with practical experimentation. It provides valuable insights into the advantages, disadvantages, and challenges of implementing CQRS architecture in complex information systems.*

***Key words:** Domain-Driven Design, CQRS, Event Sourcing, Architecture migration.*

Олександр ЛИТВИНОВ
кандидат технічних наук, доцент, факультет фізики, електроніки та комп'ютерних систем, Дніпровський національний університет імені Олеся Гончара, просп. Гагаріна, 72, м. Дніпро, Україна, 49000
***ORCID:** 0000-0001-7660-1353*

Дмитро ГРУЗІН
магістр, аспірант, факультет фізики, електроніки та комп'ютерних систем, Дніпровський національний університет імені Олеся Гончара, просп. Гагаріна, 72, м. Дніпро, Україна, 49000
***ORCID:** 0009-0004-8534-2559*

Максим ФРОЛОВ
магістр, факультет фізики, електроніки та комп'ютерних систем, Дніпровський національний університет імені Олеся Гончара, просп. Гагаріна, 72, м. Дніпро, Україна, 49000
***ORCID:** 0009-0000-6624-6028*

Бібліографічний опис статті: Литвинов, О., Грузін, Д., Фролов, М. (2024). Про міграцію проекту з Domain Driven Design архітектурою на CQRS та Event Sourcing. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 1, 50–60, doi: https://doi.org/10.32782/IT/2024-1-7

ПРО МІГРАЦІЮ ПРОЕКТУ З DOMAIN DRIVEN DESIGN АРХІТЕКТУРОЮ НА CQRS ТА EVENT SOURCING

Стаття розглядає проблему міграції додатків, зокрема тих, що використовують архітектурний підхід Domain-Driven Design (DDD), до парадигми Command Query Responsibility Segregation (CQRS) з Event Sourcing. Довго існуючі системи часто стикаються з проблемами, пов'язаними з негнучкою, застарілою архітектурою та залежностями, що призводять до збільшення витрат на обслуговування. У роботі розглядаються переваги DDD та пропонується CQRS як життєздатна альтернатива, з акцентом на покращенні продуктивності та масштабованості.

*Основною **метою роботи** є оцінка безпечного шляху міграції проекту з DDD архітектурою на архітектуру CQRS та Event Sourcing, а також визначення дорожньої карти міграції. У статті проводиться експеримент, в якому здійснюється міграція тестового проекту, оцінюються час, зусилля та результати міграції. Методологія дослідження включає оцінку складності за допомогою метрики цикломатичної складності МакКейба та оцінку продуктивності через час виконання методів системи.*

Експеримент проводиться на типовому проекті – системі відстеження завдань. Результати реалізації CQRS показують збільшення кількості класів у чотири рази та кількості рядків коду на 50 відсотків. Однак це збільшення є обґрунтованим, оскільки воно покращує модульність, прозорість та керованість під час розробки, що в кінцевому підсумку полегшує обслуговування та, в цілому, значно підвищує продуктивність системи. Варто зазначити, що загальна цикломатична складність системи майже не змінилася.

***Підсумовуючи**, стаття розглядає оцінку міграції проекту з DDD архітектурою на CQRS та Event Sourcing, поєднуючи теоретичні висновки з практичним експериментом. Вона надає цінну інформацію щодо переваг, недоліків та викликів при впровадженні архітектури CQRS в складні інформаційні системи.*

***Ключові слова:** доменно-орієнтований дизайн, CQRS, Event Sourcing, міграція архітектури.*

Background and Literature Review. In the market, there is a vast number of applications. These are complex information systems that still perform their functions but over time have lost flexibility, possess outdated architecture, or dependencies. This leads to an increase in the complexity and cost of maintaining such systems.

Many modern applications adopt the Domain-Driven Design (DDD) architecture (Evans, 2004) (Vernon, 2013), a methodology that emphasizes aligning software design with the domain model. DDD revolves around the concept of bounded contexts, which define clear boundaries within a system where specific domain models, terms, and rules apply coherently. The concept of bounded contexts (Fowler, 2014) in DDD promotes modular design and clear separation of concerns, allowing different parts of the system to operate independently within their designated contexts. Each bounded context delineates its language, rules, and semantics, fostering better understanding and communication among team members working within that context. Within these bounded contexts, system entities are identified and organized into aggregates or aggregate roots, which represent clusters of associated objects treated as a single unit for data manipulation and consistency.

An aggregate is the primary entity at the business logic level. When there is a request to modify or read data from outside, this request passes through the controller level and reaches the business logic level, which calls the repository to retrieve the necessary aggregate from the database. Once the aggregate is loaded into memory, the necessary methods are invoked to perform the requested operations, thereby modifying its state according to the business logic encapsulated within the domain model. Subsequently, the updated aggregate is saved back to the database, by the repository (Fig. 1).

One alternative, when it comes to the architecture of complex information systems, is **CQRS (Command Query Responsibility Segregation) with Event Sourcing** (Young, 2010; Betts, 2013). This approach can be regarded as a derivative of DDD.

Event Sourcing implies the absence of a classical database, and data is stored in the form of events that represent changes to the system's state. Events are stored in an Event Store, which

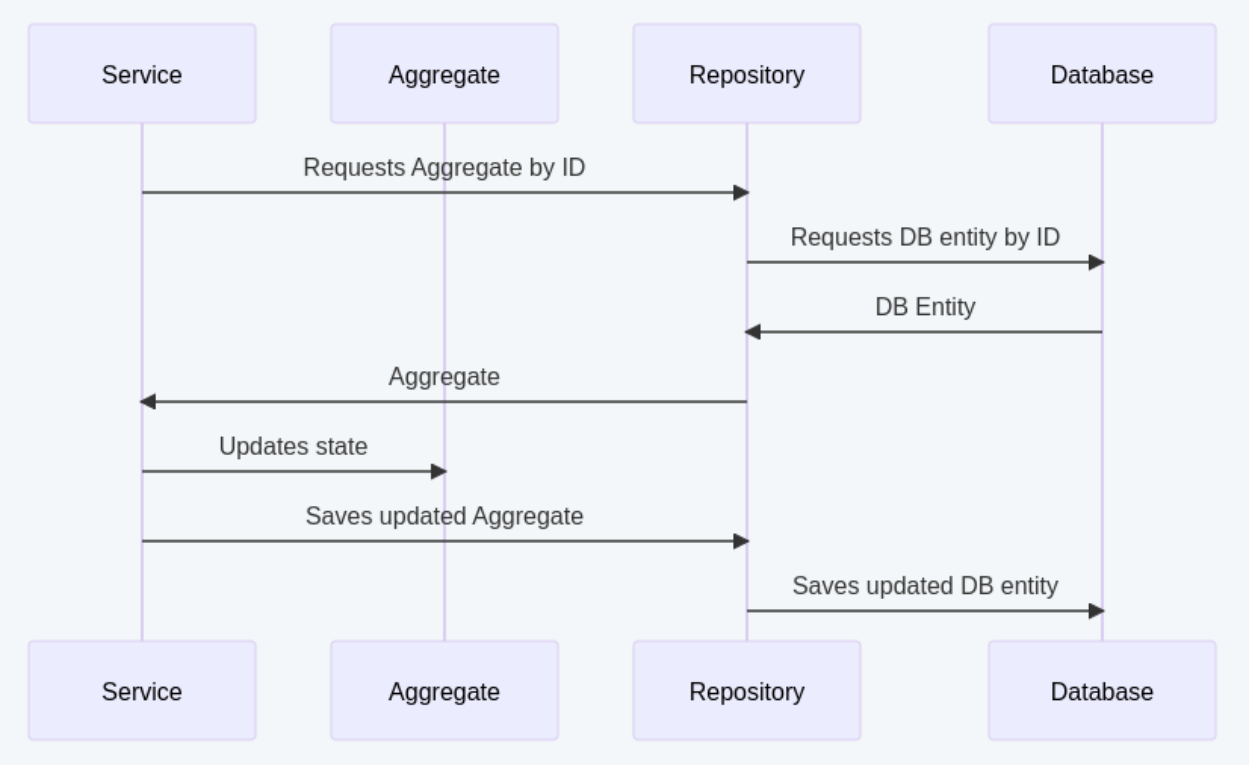

Fig. 1. DDD update operation flow

serves as the source of truth for the system. Classically, when querying an aggregate, the repository does not simply fetch the required record from the database but retrieves all events associated with the relevant instance from the event store. Subsequently, these events are replayed onto a new entity, resulting in the retrieval of the aggregate object in its current state.

CQRS, in turn, proposes the separation of read and write operations at the logic level and often even at the data storage level. In conjunction with Event Sourcing, the data processing process looks as follows (Fig. 2). After the aggregate's state is modified, events about its changes are saved in the Event Store and also sent to the Event Bus to notify other parts of the system about the occurred changes. Individual handlers are subscribed to specific events and update projections (6) – denormalized data views prepared directly for read operations. When a read Query is requested, the query handler selects pre-prepared data from the database without resorting to complex aggregation queries or additional data mappings. This approach adds development complexity but positively impacts the system's performance and flexibility.

The advantages of the CQRS with Event Sourcing architecture compared to DDD (Kenneth, 2013) include improved performance for read and write requests, as well as better flexibility and scalability due to asynchronous event processing and reduced risk of conflicts when making changes. This is because commands that modify data and queries that read data operate independently of each other. Another significant advantage is the instant storage of all events, enabling the system's state to be restored to any point in time from its creation to the present.

Task definition. For some systems using DDD architecture, there arises a need to store events for monitoring or taking some business solutions, or enhance flexibility, making the CQRS with Event Sourcing architecture more suitable for the system than DDD. For modern information systems, issues of performance, scalability, and ease of software maintenance are crucial. The CQRS architecture provides an innovative methodological approach to optimize command and query processing in applications, contributing to increased productivity, scalability, and modularity of systems. However, alongside the advantages, there are practical challenges in its implementation, such as an increase in the number of classes and configuration complexity. Therefore, researching the practical aspects of applying the CQRS architecture for developing a modern information system, analyzing and evaluating the pros and cons of this approach, remains relevant.

The objective of this article is to assess a secure path for migrating a complex information

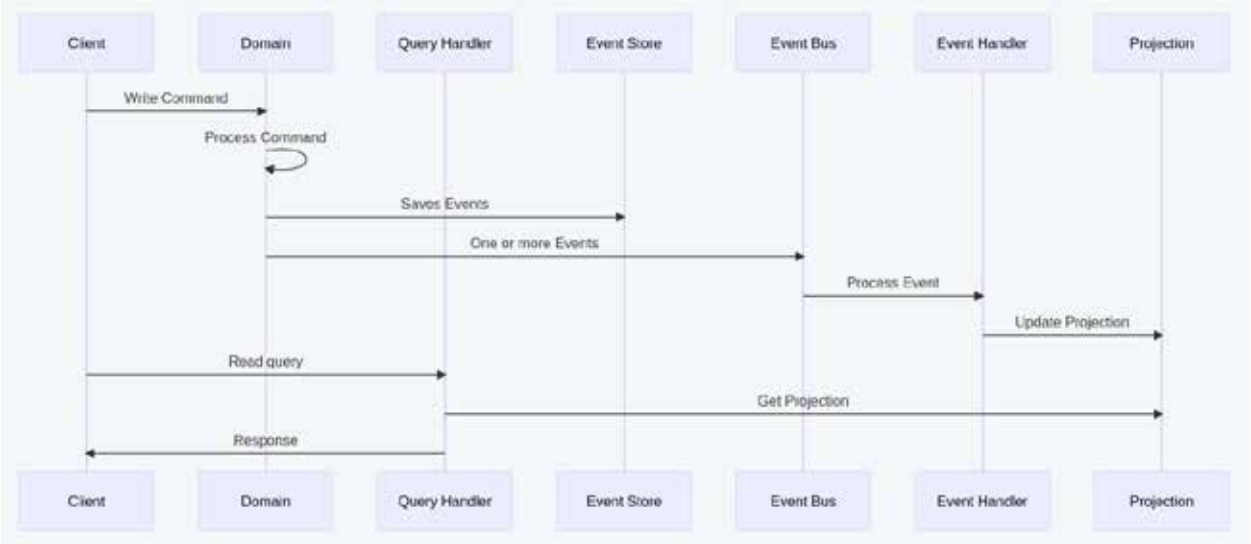

Fig. 2. CQRS read/write operations flow

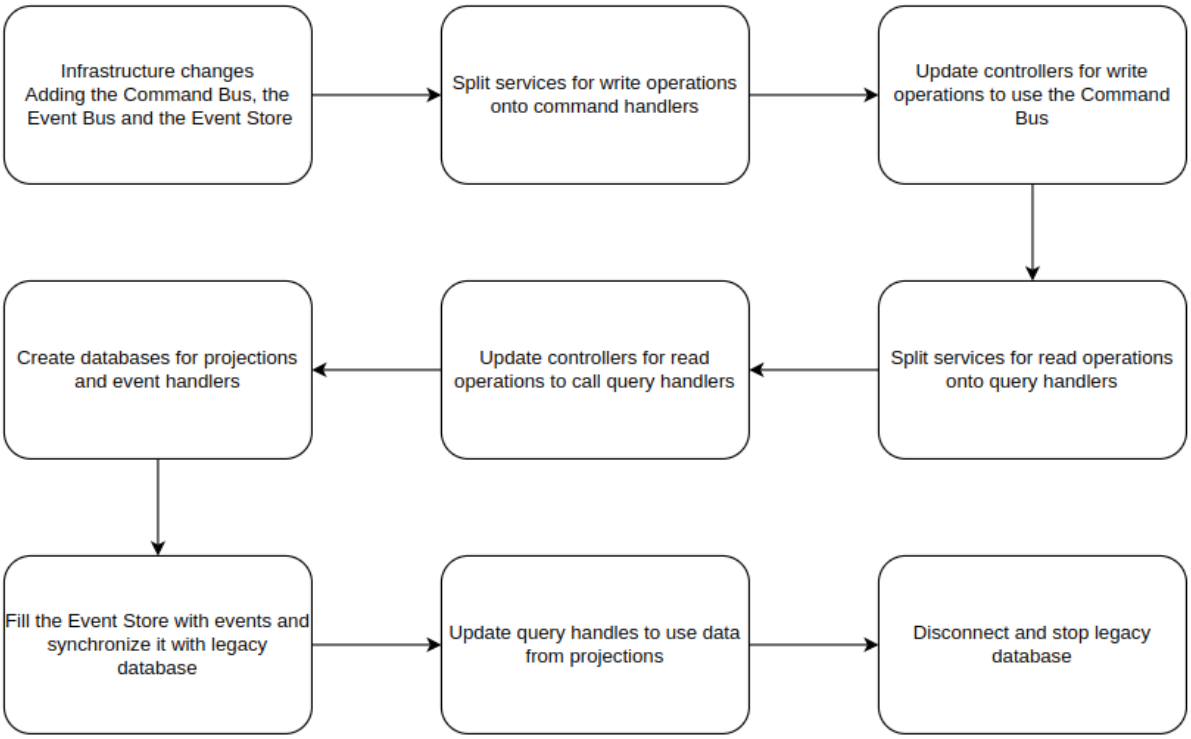

Fig. 3. DDD to CQRS with Event Sourcing architecture migration flow

system from DDD to the CQRS with Event Sourcing architecture. This article describes the migration stages. Additionally, it involves conducting an experiment to migrate the architecture of a test project and provides an assessment of the time and results of the migration for a test typical project.

Main part. The migration of an existing complex information system from DDD to CQRS with Event Sourcing architecture (Salvatierra, 2013) is not a simple task and involves several stages.

Fig. 3 describes the flow of migration of the DDD system to a CQRS with Event Sourcing application. The first step is to make infrastructural changes, specifically adding the Command Bus, the Event Store, and the Event Bus. After implementing these modules, the controller level and service level (domain) need to be adapted to work with the new modules. At the service level, in accordance with the CQRS architecture, it is assumed that the controller receives a request from an external client, creates a command, and

sends it to the Command Bus. At this stage, it's essential to decide whether to modify the public API of the system. If the public API is going to change, it's better to do it as early as possible to give developers of client applications time to update the communication contract with the system while the main migration is taking place. Existing controllers need to be updated in the following way. Instead of directly calling domain-level services, controllers responsible for write operations should create corresponding commands and pass them to the Command Bus and it means that the Client application cannot receive the result of the operation in response to the operation request directly. The domain services then should be divided and transformed into command handlers and query handlers. Command handlers should subscribe to the Command Bus and process the corresponding commands. The logic for read operations is moved into query handlers. The controllers responsible for performing queries continue to call these methods to retrieve and pass data to the client, but now from query handlers. In result, each application service will be divided into several handlers depending on its logical load.

In the next stage, databases for denormalized data views, which query handlers will use for quick retrieval of necessary responses (projections), are described and created. Having ready projections allows creating and testing event handlers, which, upon receiving corresponding events in the event bus, will update the projections.

All the previous steps only involved changes in the code and did not affect the system's data. The next step of the Cold Turkey migration (Brodie, 1995) is migrating data from the normalized database to the Event Store and shifting the focus of source-of-truth to the Event Store. Performing such an operation on a constantly updating system is a challenging process. Based on Marius Breitmayer's work on deriving Event Logs from Legacy Software Systems (Breitmayer, 2023), data migration occurs as follows: during a maintenance break, a database dump is made, and the system is updated to a version in which, in parallel with the operation of the legacy database, events are recorded to the Event Store. After launching the system, based on the available data in the dump, events of creating existing entities with a timestamp equal to the creation of a record in the database are migrated to the event store. Thus, the event store is synchronized with the legacy database. After synchronization, another maintenance break will be required, during which projection databases will be filled using the event replay algorithm from the current event store, and the system will be updated to a version in which event handlers are enabled and update projections.

In the subsequent stages of development, the logic of query handlers is updated one by one to work with projections instead of the legacy database. After fully updating the codebase of query handlers, the last stage of system update is disconnecting and stopping the legacy database.

Methods

McCabe Cyclomatic Complexity is a crucial metric in software engineering that assesses the complexity of a program by measuring the number of linearly independent paths through its source code. Introduced by Thomas J. McCabe (McCabe, 1976), this algorithmic approach has become an integral part of software quality assessment and maintenance.

The Cyclomatic Complexity of a program is calculated using the formula 1.

$$V = E - N + 2 * P, \quad (1)$$

where: E – is the number of edges in the flow graph,

N – is the number of nodes in the flow graph, and

P – is the number of connected components (regions) in the flow graph.

The resulting value provides insights into the program's complexity. Generally, a higher Cyclomatic Complexity indicates a higher likelihood of bugs and maintenance challenges. Various thresholds and guidelines exist to help interpret the complexity score, aiding developers in optimizing and refactoring their code.

To assess **performance**, measurements of the execution time of the system's basic methods are conducted. The measurements are performed automatically. The method is called in a loop 1000 times, after which the arithmetic mean is calculated to determine the average execution time of the method.

Experiment.

The goal of the experiment is to explore the possibilities of implementing CQRS with Event Sourcing approach in real projects and to assess the time and effort required to migrate a project with a specific volume of code from DDD architecture to CQRS with Event Sourcing. Additionally, the complexity and performance of the original and migrated systems will be evaluated and compared.

To achieve the goal, the following tasks need to be addressed:

- Select a typical project and research methodology.
- Implement a typical project using a conventional approach (without applying CQRS and Event Sourcing).

● Conduct experiments to adapt the developed project to the CQRS with Event Sourcing architecture, measuring the time and effort spent on adaptation.

● Conduct an analysis of the results. Evaluate the advantages and disadvantages of CQRS with Event Sourcing architecture, and potential challenges in its implementation.

As already defined, it is necessary to select a project with logic that is not overly complex but includes typical tasks of a real project, to be developed using the chosen approach. Additionally, to identify the advantages and disadvantages of this approach, a comparison of the time spent on implementation and modification of functions with the conventional approach (without using CQRS and Event Sourcing) is required. The chosen project is a Task Tracking System, which is a classic sample project for studying the design and implementation aspects of systems with a complex application domain. For example, a similar project is used as a sample in Vernon's work (Vernon, 2011).

The application domain of the project is ideally suited for research because developers do not need to spend time learning unfamiliar terminology and business logic. Additionally, this application domain is not simple, and such a system can contain many functions and capabilities.

The source code for the conducted experiments is hosted in an open-source repository on GitHub and is accessible via the link (Frolov, 2023).

TaskTrackingSystem description

To grasp the magnitude of the experimental undertaking, refer to Figures 4 and 5, which provide component diagrams and the database structure of the Domain-Driven Design system. The system's domain comprises three aggregate roots: User, Assignment, and Project.

In Figure 4, the component diagram delineates the relationships between the aggregate roots and their subordinate entities. The Project aggregate oversees projects and their statuses, while the Assignment aggregate encompasses assignments and assignment statuses. The User is more intricate, aggregating users, user projects, and positions.

Within the Data Access Layer, additional entities exist, but the overarching structure remains consistent. The primary entities are User, Assignment, and Project models. Each of these entities maintains relationships with other tables, such as Project Statuses, Tokens, Positions, etc. Users exhibit a one-to-many relationship with Assignments and

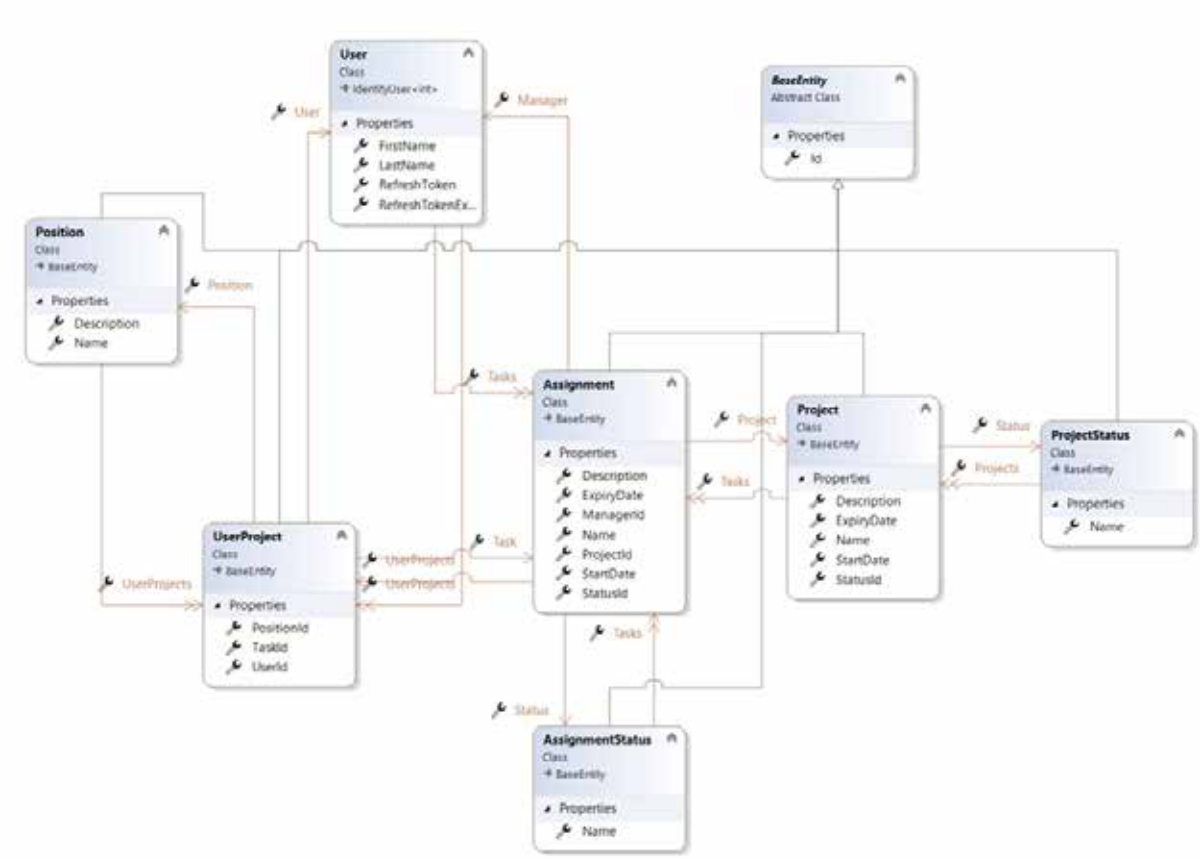

Fig. 4. Component diagram of TaskTrackingSystem

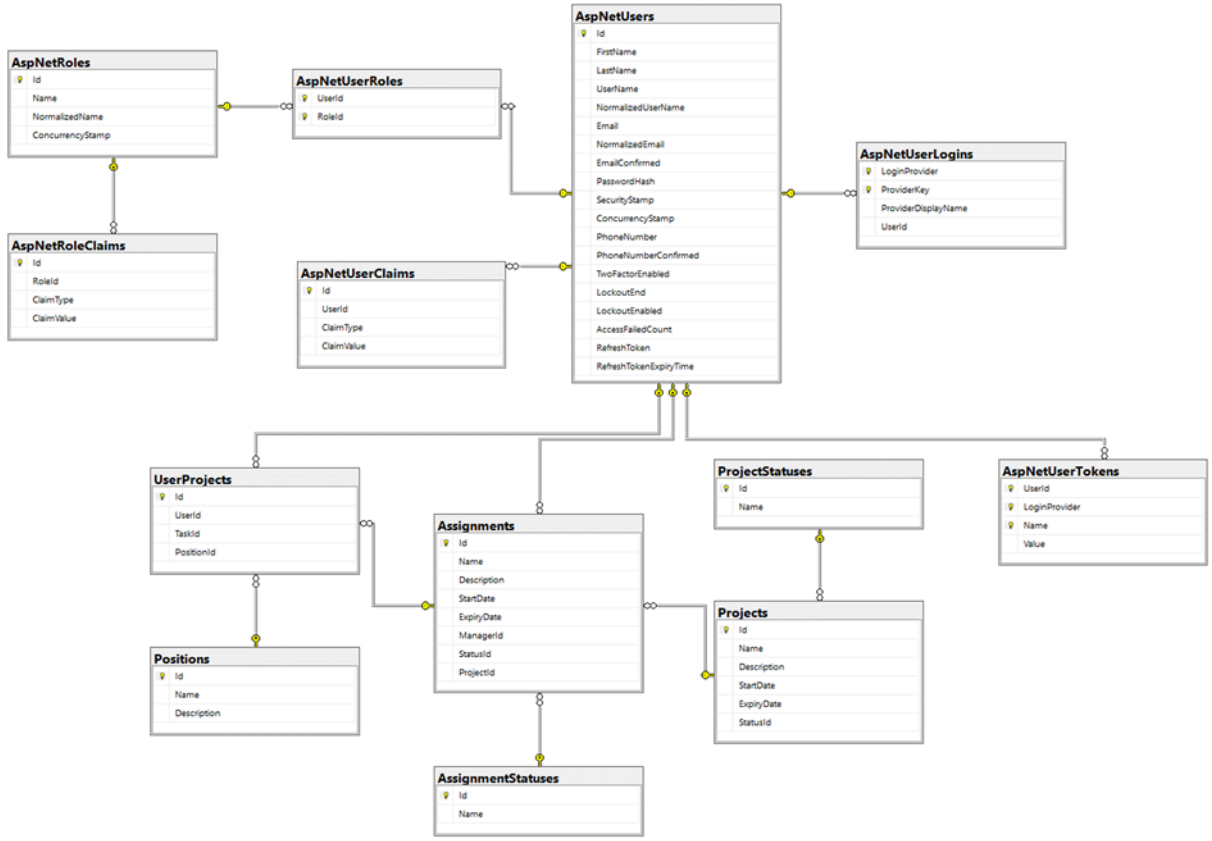

Fig. 5. Database structure of TaskTrackingSystem

a many-to-many relationship with Projects. Additionally, each project can encompass multiple assignments.

The TaskTrackingSystem functionality encompasses various methods. For the User aggregate, these include adding, reading, editing, and deleting (CRUD operations) users, reading users with comprehensive information, CRUD operations for user roles, and authorization methods like sign-up, sign-in, and password update. The Assignment functionality involves assigning users to tasks with selected positions, CRUD operations for assignments, and assignment statuses. As for the Project aggregate, it involves simultaneous CRUD operations for projects and project statuses, along with operations for adding and removing tasks to/from the project.

Results of CQRS Implementation

Complexity evaluation

To write the BLL without using the CQRS pattern, it took 12 days, and 47 classes were written to ensure the full functioning of the business logic layer.

As seen in Fig. 6, there was initially a growth in project development in the form of creating various classes that met the requirements. However, towards the end of the development, there were fewer new classes created under new requirements, and the focus shifted to modifying existing ones. By the 12th day, the majority of classes contained at least 300 lines of code. In Fig. 7, it can be observed that when a new requirement emerged or a bug requiring modification of a part of the business logic was detected, it was necessary to check whether the modification affected other parts of the business logic since everything was interconnected. In the end, modifying and maintaining the project required significant effort.

The implementation of the CQRS (Command Query Responsibility Segregation) pattern resulted in an increase in the number of classes in the business logic from 47 to 213 and required a significant portion of the expended time (Fig. 8).

At first glance, it may seem that this increase complicates the system. However, the reason for this increase is an improved distribution of responsibilities among system components, which enhances the modularity of the system, its adaptability to changes, testability, and ensures its ongoing support and development (Fig. 9).

The main advantage of this increase in the number of classes lies in the separation of responsibilities between commands and queries. Commands, responsible for changing the system's state, and queries providing data access are placed in distinct classes. This allows for better manageability

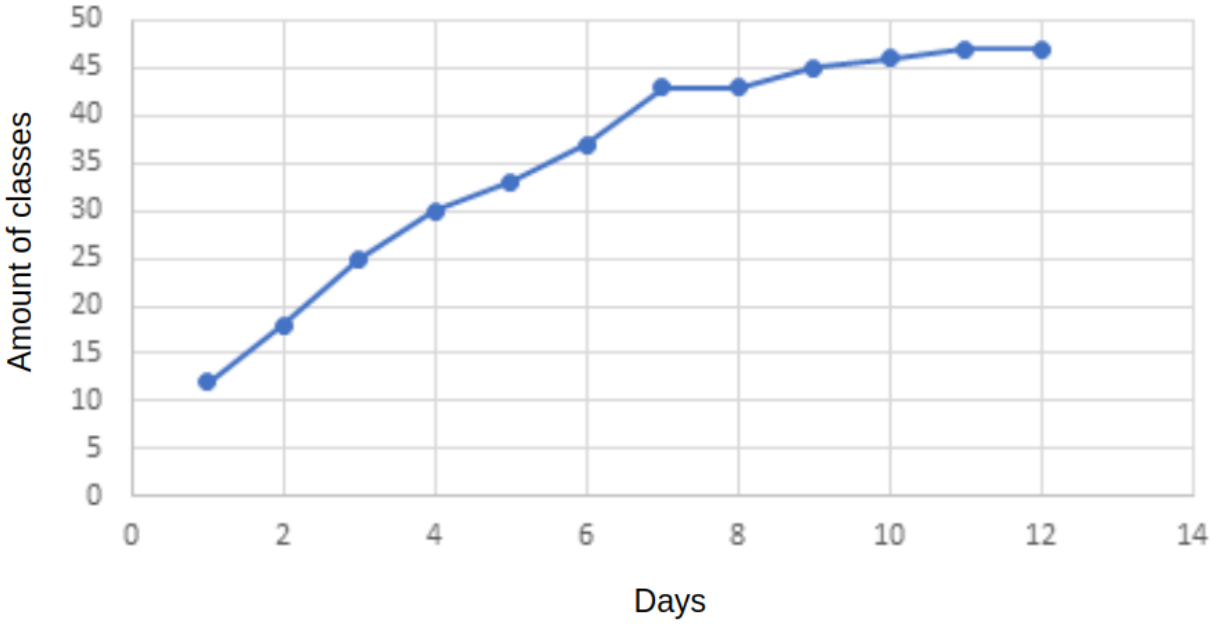

Fig. 6. The dependence of the number of classes on the number of development days for DDD approach implementation

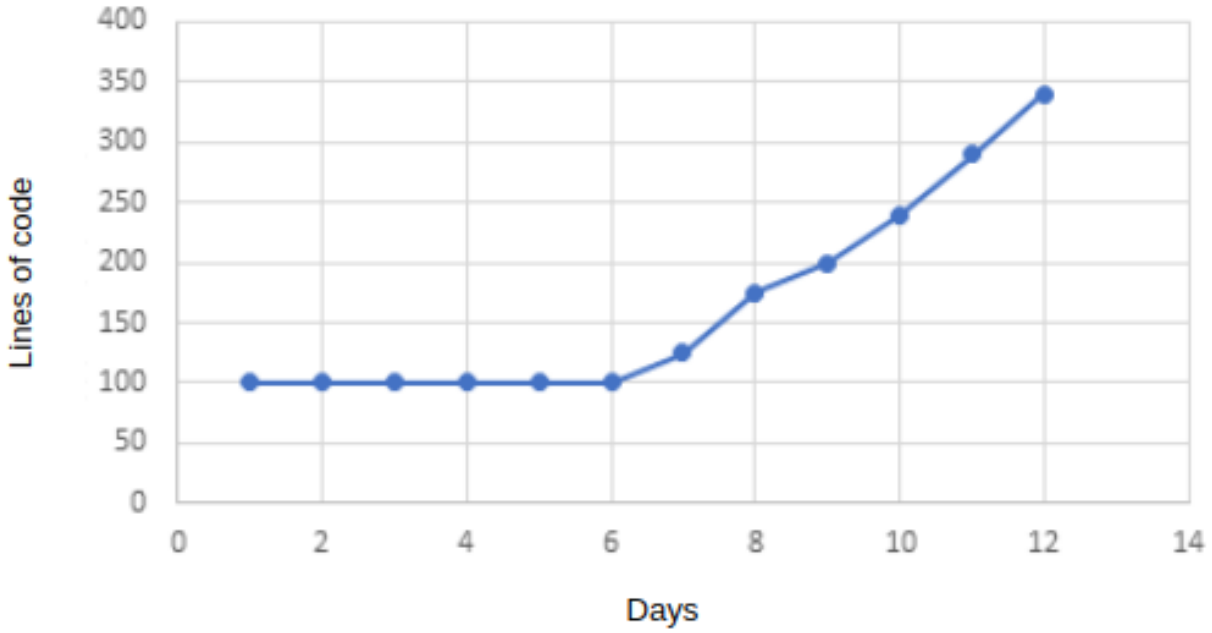

Fig. 7. The dependence of the development and maintenance cost coefficient on the number of development days for DDD approach implementation

during development, as it becomes more transparent and modular.

Although it may seem that this leads to an increased workload during implementation, it ultimately facilitates system maintenance. By separating commands and queries, each class is responsible for specific functionality. This simplifies debugging, extension, and adding new functionality without the need to edit large code blocks, as the largest classes now contain no more than 50 lines of code.

This is also confirmed by cyclomatic complexity metrics, calculated for the whole project using SonarCloud (14). Table 1 shows the metrics for the system with DDD architecture and CQRS with Event Sourcing architecture. Despite the increase in the number of classes and the addition of modules such as Command Bus, Event Bus, and Event

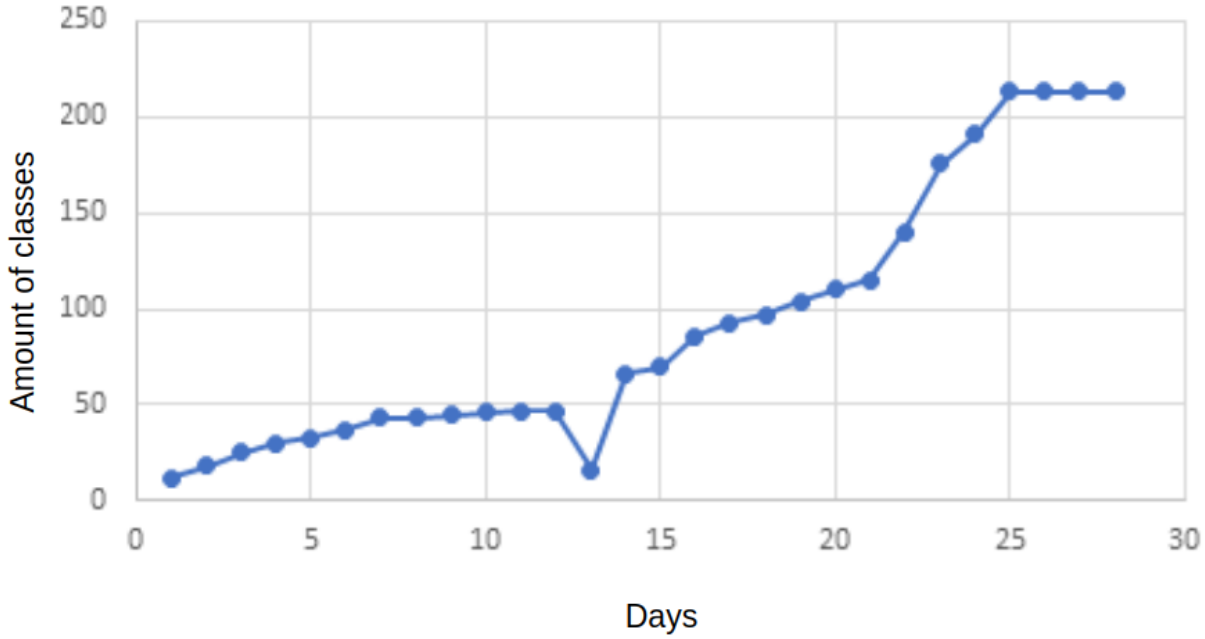

Fig. 8. The dependency between the number of classes and the development days including migration to CQRS and Event Sourcing approach

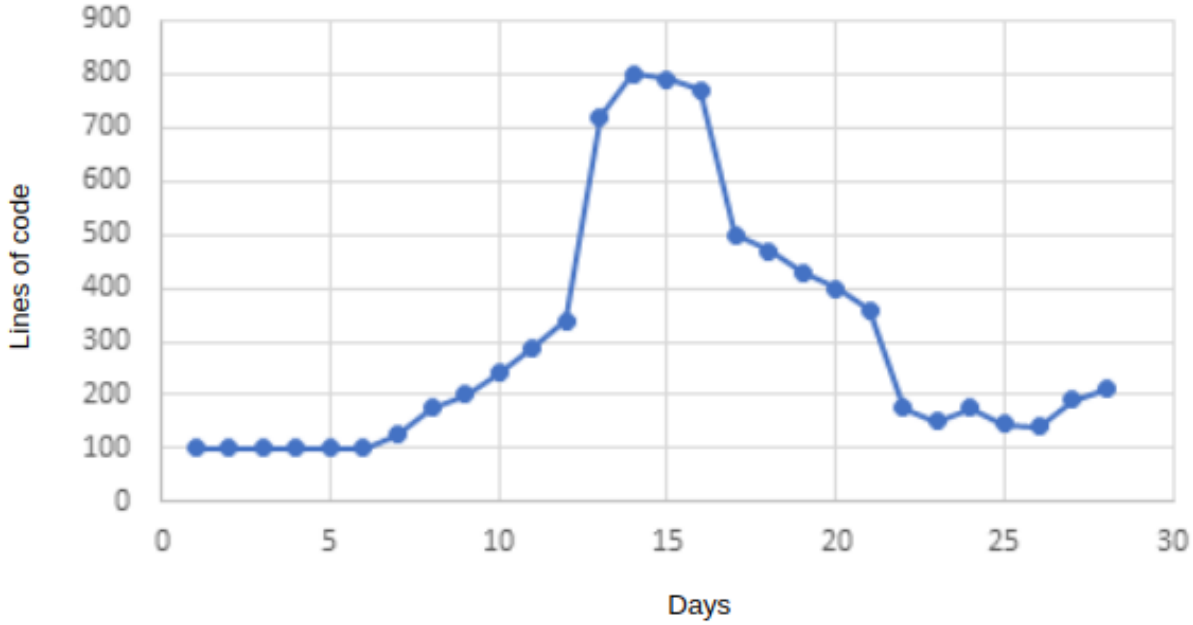

Fig. 9. The relationship between the lines of code and the number of development days including migration to CQRS and Event Sourcing approach

Store, the overall complexity of the system after migration to CQRS with Event Sourcing architecture became even lower. Thus, the total number of lines of code for the DDD architecture option amounted to slightly over three thousand with a total cyclomatic complexity of 534, while for CQRS with Event Sourcing, it was 4,620 with a complexity of 522.

However, if the project lacks complex business logic and there are no prospects for its complication, the time and effort spent on developing complex infrastructure and system functions may not be justified. That is, the time and effort in developing such a project with a conventional architecture will be significantly less than in the case of applying CQRS, and the benefits of supporting a

Table 1

Complexity metrics

Metric	DDD	CQRS with Event Sourcing
Lines of code	3 101	4 620
Overall Cyclomatic complexity	534	522
Business Logic Layer Cyclomatic complexity	312	285
Data Access Layer Cyclomatic complexity	133	157
Web API Cyclomatic complexity	89	80

Table 2

Performance evaluation. Function response time

Method	DDD (time ms.)	CQRS with Event Sourcing (time ms.)
getUsers	281	43
addUser	28	48
updateUser	119	46
deleteUser	71	52
getUser	37	37

complex system provided by the CQRS approach may be nullified.

Performance evaluation

Table 2 provides a comparison of the average execution speed across 1000 requests for basic methods for the case of applying the CQRS with Event Sourcing architecture and DDD architecture. To reduce experimental error, the database was replaced with a mock object containing static data.

Reading the list of users (getUsers): The use of CQRS significantly accelerated the operation of retrieving the list of users, reducing the execution time from 281 to 43. This demonstrates the effectiveness of separating read and write operations.

Adding a new user (addUser): Although the addition operation became slightly slower with the use of CQRS, this may be compensated by other advantages of the architecture in the future.

Updating a user (updateUser): Applying CQRS resulted in a significant reduction in the execution time of the updateUser operation, reducing it from 119 to 46.

Deleting a user (deleteUser): With the use of CQRS, the execution time of deleting a user became slightly faster, decreasing from 71 to 52.

Getting a specific user (getUser): The execution time of the operation to retrieve a specific user remains stable with both approaches.

Summary. The work investigated the implementation of the CQRS with Event Sourcing architecture in the already developed TaskTrackingSystem. A performance comparison was made between systems with and without the architecture, and the time costs of migrating the system to the new architecture were evaluated.

The analysis of the research results indicates a significant impact of implementing CQRS with Event Sourcing architecture on the system's efficiency. In particular, the getUsers method demonstrated a noticeable reduction in execution time from 281 ms to 43 ms after the architecture's introduction. On the other hand, the addUser method showed an increase in execution time from 28 ms to 48 ms. The data analysis also shows that the implementation of CQRS significantly affects the cost coefficient. Without considering the days of CQRS implementation, the coefficient gradually increases depending on the development duration due to the system's complexity. With the introduction of CQRS, there is a sharp increase in the coefficient on the 13th day, after which it stabilizes and becomes lower than when CQRS was not used. The overall dynamics of the cost coefficient show a systematic increase without the implementation of CQRS, depending on the development duration. After the introduction of CQRS, project maintenance costs are stable.

BIBLIOGRAPHY:

1. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software, 2004. ISBN: 978-0321125217.

2. Vernon V. Implementing Domain-Driven Design, 2013. ISBN: 978-0321834577.

3. Fowler M. Bounded Context, 2014. URL: https://martinfowler.com/bliki/BoundedContext.html.

4. Young G. CQRS Documents by Greg Young, 2010. Pages: 50–52. URL: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.

5. Betts D. Dominguez J. Melnik G. Simonazzi F. Subramanian M. Young G. Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure, 2013. ISBN: 978-1621140160.

6. Practical and focused guide for survival in post-CQRS world. Projections. URL: http://cqrs.wikidot.com/doc:projection.

7. Kenneth T. Introduction to Domain Driven Design, CQRS and Event Sourcing, 2013. URL: https://www.kenneth-truyers.net/2013/12/05/introduction-to-domain-driven-design-cqrs-and-event-sourcing/.

8. Salvatierra G. Mateos C. Crasso M. Legacy System Migration Approaches, 2013. DOI: 10.1109/TLA.2013.6533975.

9. Brodie M. L. Stonebraker M. Ai S. DARWIN: On the Incremental Migration of Legacy Information Systems, 1995.

10. Breitmayer M. Arnold L. La Rocca S. Reichert M. Deriving Event Logs from Legacy Software Systems, 2023. DOI: 10.1007/978-3-031-27815-0_30.

11. McCabe T. J. A Complexity Measure, 1976. DOI: 10.1109/TSE.1976.233837.

12. Vernon V. Effective Aggregate Design, 2011. Parts I – III.

13. Frolov M. TaskTrackingSystem repository on GitHub, 2023.

14. SonarCloud Online Code Review as a Service Tool. URL: https://sonarcloud.io/.

REFERENCES:

1. Evans, E. (2004). Domain-Driven Design: Tackling Complexity in the Heart of Software. ISBN: 978-0321125217.

2. Vernon, V. (2013). Implementing Domain-Driven Design. ISBN: 978-0321834577.

3. Fowler, M. (2014). Bounded Context. Retrieved from: https://martinfowler.com/bliki/BoundedContext.html.

4. Young, G. (2010). CQRS Documents by Greg Young. Pages: 50–52. Retrieved from: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.

5. Betts, D., Dominguez, J., Melnik, G., Simonazzi, F., Subramanian, M. & Young, G. (2013). Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure. ISBN: 978-1621140160.

6. Practical and focused guide for survival in post-CQRS world. Projections. Retrieved from: http://cqrs.wikidot.com/doc:projection.

7. Kenneth, T. (2013). Introduction to Domain Driven Design, CQRS and Event Sourcing. Retrieved from: https://www.kenneth-truyers.net/2013/12/05/introduction-to-domain-driven-design-cqrs-and-event-sourcing/.

8. Salvatierra, G., Mateos, C. & Crasso, M. (2013). Legacy System Migration Approaches. DOI: 10.1109/TLA.2013.6533975.

9. Brodie, M., L. & Stonebraker, M., Ai, S. (1995). DARWIN: On the Incremental Migration of Legacy Information Systems.

10. Breitmayer, M., Arnold, L., La Rocca, S. & Reichert, M. (2023). Deriving Event Logs from Legacy Software Systems. DOI: 10.1007/978-3-031-27815-0_30.

11. McCabe, T., J. (1976). A Complexity Measure. DOI: 10.1109/TSE.1976.233837.

12. Vernon, V. (2011). Effective Aggregate Design. Parts I – III.

13. Frolov, M. (2023). TaskTrackingSystem repository on GitHub.

14. SonarCloud Online Code Review as a Service Tool. Retrieved from: https://sonarcloud.io/.