https://doi.org/10.31891/csit-2023-2-10
UDC 004.42

Volodymyr TYKHOKHOD, Anton PASICHNIUK
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"

MODELING AND IMPLEMENTATION OF DOMAIN EVENTS IN THE DOMAIN-DRIVEN ARCHITECTURE ON THE .NET CORE PLATFORM

A software architecture centered on a domain model can provide significant advantages over other types of architectures in the long-term development and maintenance of systems with complex domain logic. At the same time, domain-driven design and approaches to software implementation of systems are relatively new concepts and continue to develop in application to various platforms, technologies and programming languages, and are of considerable interest to designers and developers. The presented work examines existing approaches to modeling and implementing domain events on the .NET Core platform in domain-driven architecture, which is one of the newest patterns. There are two approaches to implementing domain event behavior: immediate and delayed event propagation. These two approaches are analyzed and their features are described in detail. The implementation of instant propagation of domain events within a command execution transaction in the CQRS architecture is described. This implementation allows you to get rid of external dependencies, achieve purity of domain entities, as it eliminates the need to inject services, repositories in entities, and also prevents memory leaks and is safe for multithreaded use. Schematically depicts the abstract process of an external command entering a domain model, which causes a change in the state of an aggregate and the propagation of side effects with domain events. This process takes into account the capabilities of the Entity Framework object-relational mapping framework to retrieve context objects that have been changed during process. The entire stack of objects involved in this activity is located in the shared process memory, and the interaction occurs in synchronous mode. For the conceptual detection of events and aggregates, the event storming technique is used, the features of which are discussed in the article.

Keywords: domain-driven design, DDD, design of complex domain areas, , events of the domain area.

Володимир ТИХОХОД, Антон ПАСІЧНЮК
НТУУ «Київський політехнічний інститут імені Ігоря Сікорського»

МОДЕЛЮВАННЯ ТА РЕАЛІЗАЦІЯ ПРЕДМЕТНИХ ПОДІЙ В ПРЕДМЕТНО-ОРІЄНТОВАНІЙ АРХІТЕКТУРІ НА ПЛАТФОРМІ .NET CORE

Архітектура програмного забезпечення, в центрі якої знаходиться модель предметної області, здатна принести значні переваги в довгостроковій розробці та підтримці систем зі складною логікою предметної області в порівнянні з іншими типами архітектур. В той же час предметно-орієнтоване проектування та підходи до програмної реалізації систем є відносно новими концепціями та продовжують розвиток в застосунку до різних платформ, технологій та мов програмування, та становлять значний інтерес у проектувальників та розробників. В представленій роботі розглянуто існуючі підходи до моделювання та реалізації предметних подій на платформі .NET Core в предметно-орієнтованій архітектурі, що є одним з найновіших шаблонів. Існує два підходи до реалізації поведінки предметних подій: миттєве та відкладене розповсюдження подій. Ці два підходи проаналізовано та детально описано їх особливості. Описана реалізація миттєвого розповсюдження подій предметної області в межах транзакції виконання команди в архітектурі CQRS. Дана реалізація дозволяє позбавитись зовнішніх залежностей, досягти чистоти сутностей домену, оскільки позбавляє необхідності у впровадженні сервісів, репозиторіїв у сутності, а також дозволяє запобігти витокам пам'яті та безпечна при багатопотоковому режимі використання. Схематично зображено абстрактний процес надходження зовнішньої команди в модель предметної області, яка спричиняє зміну стану агрегату та розповсюдження побічних ефектів з подіями домену. Цей процес враховує можливості каркасу об'єктно-реляційного відображення Entity Framework для отримання об'єктів контексту, що були змінені в процесі роботи. Весь стек об'єктів, що задіяні в цій діяльності, розташовані в пам'яті спільного процесу, а взаємодія відбувається в синхронному режимі. Для концептуального виявлення подій та агрегатів використовують техніку штурм подій, особливості якої розглянуті в роботі.

Ключові слова: предметно-орієнтоване проектування, DDD, проектування складних проблемних областей, події предметної області.

Introduction

Domain-driven design (DDD) refers to the field of software engineering used to build software systems that implement complex domain logic. DDD focuses on building a system architecture, the central link of which is a domain model (a pattern classified by Martin Fowler [1]). The term domain-driven design was proposed by Eric Evans [2], who described the methodological foundations of DDD and practical techniques for implementing these concepts in the Java programming language. Later, the theoretical and practical aspects of DDD were developed in the works of Vaughn Vernon [3], Martin Fowler [1], Scott Millet [4], Jimi Nielsen [5] and other authors.

The importance of the methodology is evidenced by the field of application of domain-driven design, in particular, in the development of land resource management systems [6], maritime navigation systems [7], delivery organization systems using unmanned aerial vehicles [8].

The methodology of domain-driven design involves the use of various templates of strategic and tactical levels. To form system components with Low Coupling и High Cohesion, domain area decomposition is used using the bounded context template at the strategic level of design, as well as the aggregate template at the tactical level of design.

Events are used to communicate side effects between aggregates and to communicate information about system state changes between bounded contexts. They allow to ensure interaction between system components, without direct interconnection.

Modeling using the behavior of the event system has its own characteristics and requires the use of new approaches. The DDD concepts underlying event modeling are evolving, and events have specific implementation features on different platforms and programming languages. This is especially true for the .NET Core platform, as it is a young technology for developing cross-platform applications that is actively developing.

Therefore, the purpose of this article is to analyze modern approaches to event modeling in domain-driven design on the .NET Core platform.

Features of the architectural style

Domain-driven design is an architectural style designed to create a software model that most accurately reflects a model of a subject area (domain), including business processes and the rules that operate in it. DDD includes strategic and tactical levels of design. At each of the levels, strategic and tactical design templates are used, respectively. The goal of strategic planning is to decompose the problem area into the most conceptually isolated areas in order to curb complexity and eliminate contradictions. These isolated areas are called bounded contexts.

The goal of tactical design is to build a domain model within individual contexts. Isolation of bounded contexts is preferably achieved by implementing a separate microservice for each bounded context. However, there is usually a need to communicate information about side effects between bounded contexts.

As a result of tactical design, a set of aggregates is obtained within limited contexts. Each aggregate has its own internal state and can perform certain actions with this state. Aggregates ensure separation of domain objects from the outside world and data consistency within their borders

When an aggregate changes its state, it can generate domain events that notify other components of the system about the change, other aggregates can respond to these events and change their own state according to the received changes. This behavior, when changes in one aggregate affect the behavior of another aggregate, is also called side effects.

Simulation of events using the technique "Event storming"

Events occupy an important place in modeling the behavior of domain objects. The activity of the subject area is modeled as a sequence of events. Using subject events allows you to build software that contains components that adhere to the principle of single responsibility as much as possible.

For the conceptual identification of events and aggregates, the Event Storming technique is used, which consists in the collective discussion of concepts in order to find the events of the subject area and the internal processes that occur at the same time, drawing clear boundaries between aggregates and determining dependencies between aggregates. The storm of events can be carried out using a board with stickers, or online tools, for example, fig. 1 displays a fragment with the results of the storming of events from a board created by the Miro online system [9].

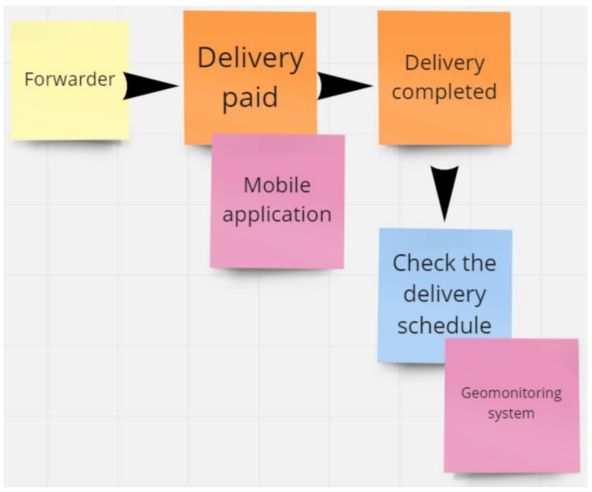


Fig. 1. A fragment of the board with the results of the Event Storming

The color of the cards on the board has a certain semantics, in figure 1 the color of the cards has the following purpose:

- The yellow color represents the object that causes the event to be generated.
- Orange color — a card with this color represents an event.
- Pink represents the system component generating the event or the external system to which the side effect is propagated.

- The blue color of the card reflects the action performed as a result of the use of side effects.

In DDD, the formation and observance of a common language of the domain area in the program code plays an important role. All conceptual and programmatic elements must be consistent and use terms and words from the dictionary of a common language. Therefore, the "Delivery Paid" and "Delivery Completed" events shown in Figure 2 can be named, for example, "DeliveryPaid" and "DeliveryCompleted" in the domain model. In nouns, verbs are used in the past tense, because the event represents a certain fact that happened.

Implementation of domain model events

Events in DDD are divided into two conceptual types: domain events and integration events, their distribution is carried out in synchronous and asynchronous modes, respectively.

Domain events are modeled in the form of simple program objects included in the domain model, which is a simple C# class that contains state but no behavior. An event represents some fact that happened in the past, so it is advisable to prohibit the change of the event object after its creation. Therefore, the following requirements are possible for the event class

1. Read-only properties with event information.
2. A public constructor with the arguments required to initialize the event instance.
3. No behavior, i.e. no methods in the event class.

Their main purpose is to spread side effects between aggregates — when certain changes in the state of one aggregate (that is, when a certain fact has occurred) affect another aggregate. At the same time, aggregates can be located in a common bounded context or divided between different contexts

The event template of the domain area is conceptually separated into a separate template, but its implementation is based on another design template Publish-subscribe [10], which separates two sides of the process — the sender (publisher), which sends messages about important facts of its state change, and subscribers (subscriber), which subscribes to messages from the sender and responds to them.

There are two approaches to the implementation of the behavior of distribution and processing of domain events:

1. Immediate distribution of events.
2. Delayed event propagation.

The first approach is to propagate an event immediately after the domain state changes, resulting in the new state being immediately committed to persistent storage, then an integration event can be published to propagate the state and achieve consistency between different microservices, bounded contexts, or external systems.

The second approach is to store events in memory objects and propagate them during persistent storage.

Implementation of immediate distribution of events

Currently, the following approaches to the implementation of domain events are used:

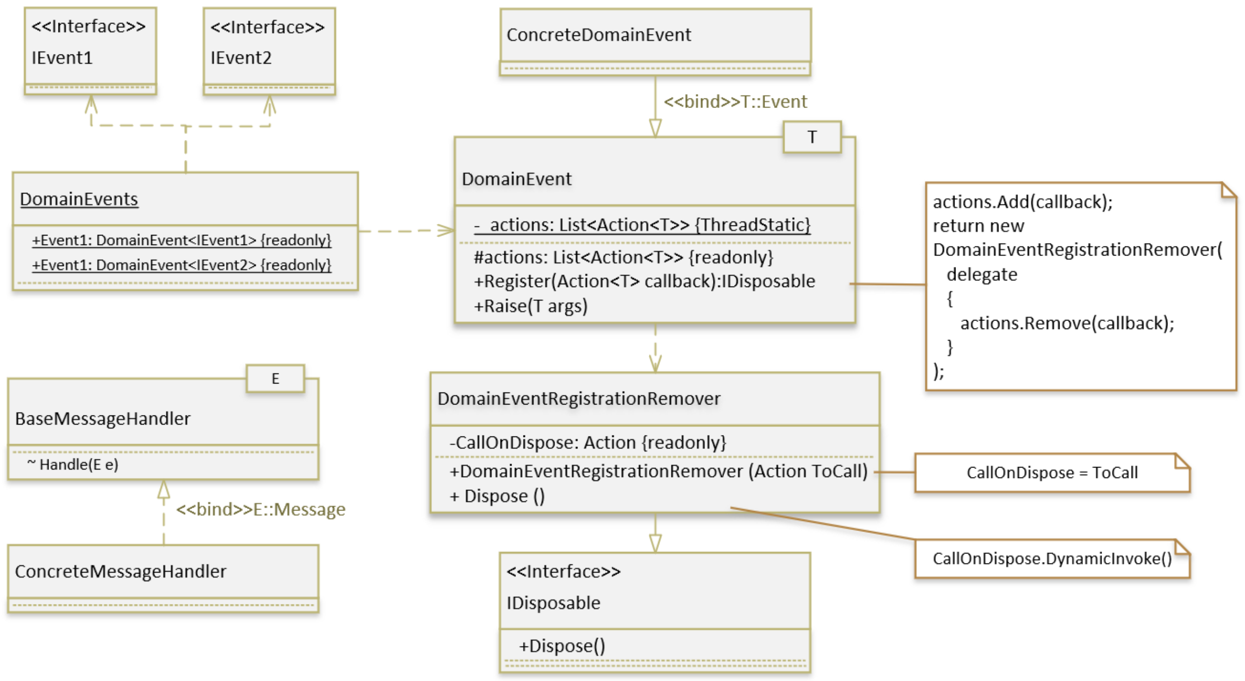
- A classic implementation with static methods of class [11].
- Using the MediatR Nuget package [12], which implements the MediatR pattern [10] and can be used to build an infrastructure for event distribution and management.

Implementation based on a class with static methods [11] allows you to get rid of external dependencies, to achieve purity of domain entities, as it eliminates the need to implement services and repositories in the entity. This implementation prevents memory leaks and is safe for multi-threaded use. Figure 2 shows a UML class diagram that represents the abstract infrastructure of domain events.

The DomainEvent class is responsible for maintaining the domain event infrastructure, it is templated and closed by the class that models the specific event. The class interface provides Register and Raise methods. The Register method allows you to register an event handler as a delegate, with a reference to the delegate added to the private actions collection. The Raise method performs event pushing — in a loop, the handler delegates contained in the actions collection are enumerated and called.

The Register method returns an object that implements the IDisposable interface, so a client of the DomainEvent class can safely implement a mechanism for freeing unmanaged resources. In this scheme, the DomainEvent class creates an instance of the DomainEventRegistrationRemover class, passing a delegate to the constructor that removes the event handler from the actions collection. This delegate is called in the Dispose method of the DomainEventRegistrationRemover class. This mechanism avoids a memory leak where a reference to an unnecessary event handler would block the release of memory by the garbage collector.

The ThreadStatic property next to the private variable actions is thread-safe, indicating to the .NET runtime that each executing thread will have access to a separate instance of the collection. An element with a ThreadStatic attribute must be static - this is a limitation of the .NET framework.



<<Interface>> IEvent1
<<Interface>> IEvent2
ConcreteDomainEvent
<<bind>>T::Event
T
DomainEvent
- actions: List<Action<T>> {ThreadStatic}
#actions: List<Action<T>> {readonly}
+Register(Action<T> callback):IDisposable
+Raise(T args)
DomainEvents
+Event1: DomainEvent<IEvent1> {readonly}
+Event1: DomainEvent<IEvent2> {readonly}
actions.Add(callback); return new DomainEventRegistrationRemover(delegate { actions.Remove(callback); });
E
BaseMessageHandler
~ Handle(E e)
<<bind>>E::Message
ConcreteMessageHandler
DomainEventRegistrationRemover
-CallOnDispose: Action {readonly}
+DomainEventRegistrationRemover (Action ToCall)
+ Dispose ()
CallOnDispose = ToCall
CallOnDispose.DynamicInvoke()
<<Interface>> IDisposable
+Dispose()

Fig. 2. Abstract infrastructure of domain events

In fig. 3 presents a sequence diagram of some abstract process of propagating domain events during the execution of a web request in the CQRS architecture.

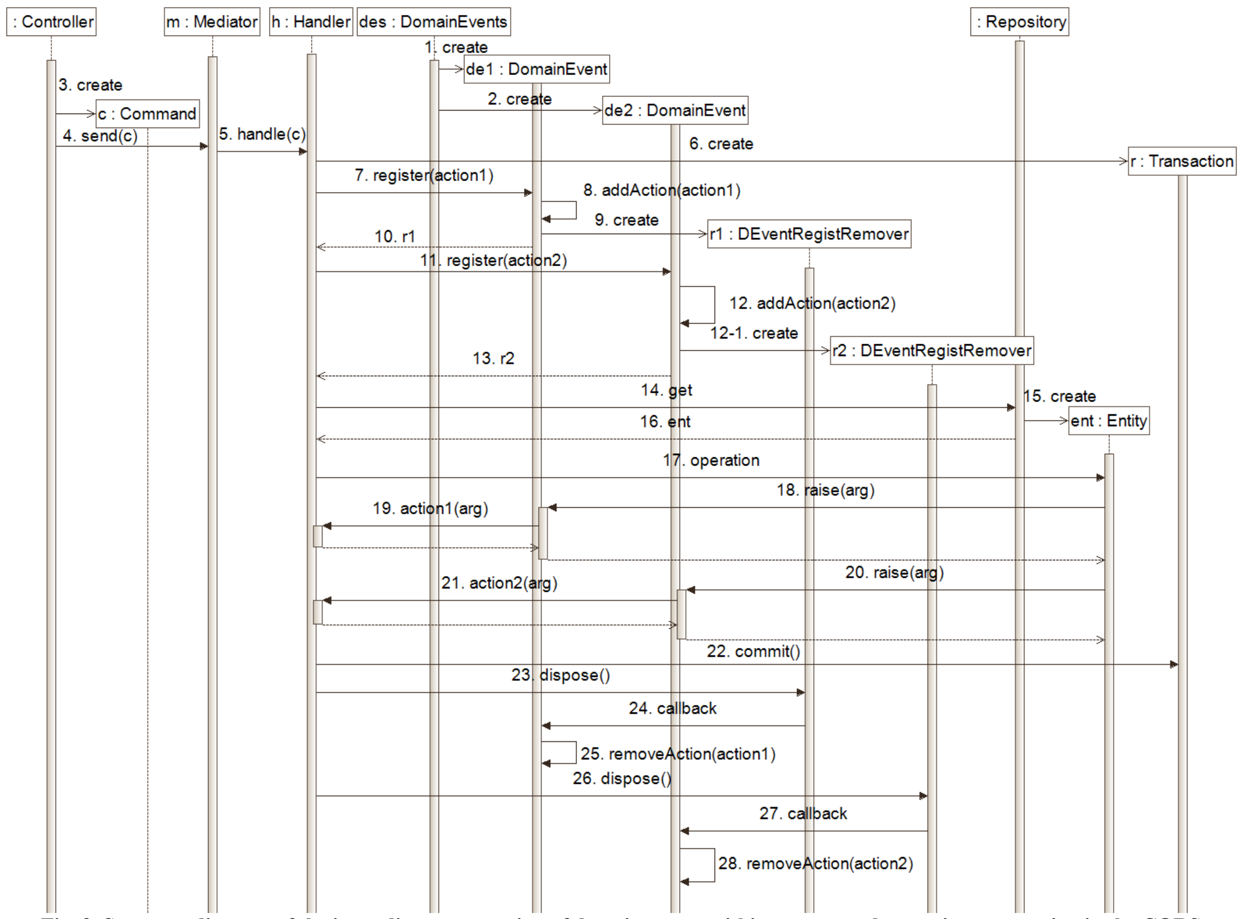


Fig. 3. Sequence diagram of the immediate propagation of domain events within a command execution transaction in the CQRS architecture

After receiving a request, a command object is created in the controller (3), which is passed to the mediator (4), the mediator finds the executor of the command and passes it to them for processing (5). In the event handler, a transaction is created (6), the object event handlers are registered (7, 11), during which handlers are added to the internal collection of DomainEvent objects (8, 12) and objects responsible for cleaning resources are created (9 , 12-1). In step 14, the domain entity is requested from the repository based on the command data, and control is passed to the operation method of this object. Next, the entity, after performing certain actions, generates two events, calling the Raise methods (18, 20) of the event objects created in steps 1, 2. They then call delegates (19, 21) responsible for processing subject events, these delegates are contained in the Handler object. Event handlers can change the states of other aggregates or pass integration events to external systems. After the work of all handlers, the transaction is completed (22) and resources are cleaned (23, 26), during which events are unsubscribed, that is, references to delegates (25, 28) are removed from the internal collections of objects of type DomainEvent

Delayed launch and dispatch of domain events

In [14], the implementation of delayed generation and sending of events is proposed. The approach of delayed launch and dispatch of domain events [16] can use the features of the Entity Framework object-relational mapping framework to obtain context objects that have been changed during operation. Figure 3 schematically depicts the abstract process of the arrival of an external command into the domain model, which causes a change in the state of the aggregate and the propagation of side effects by domain events

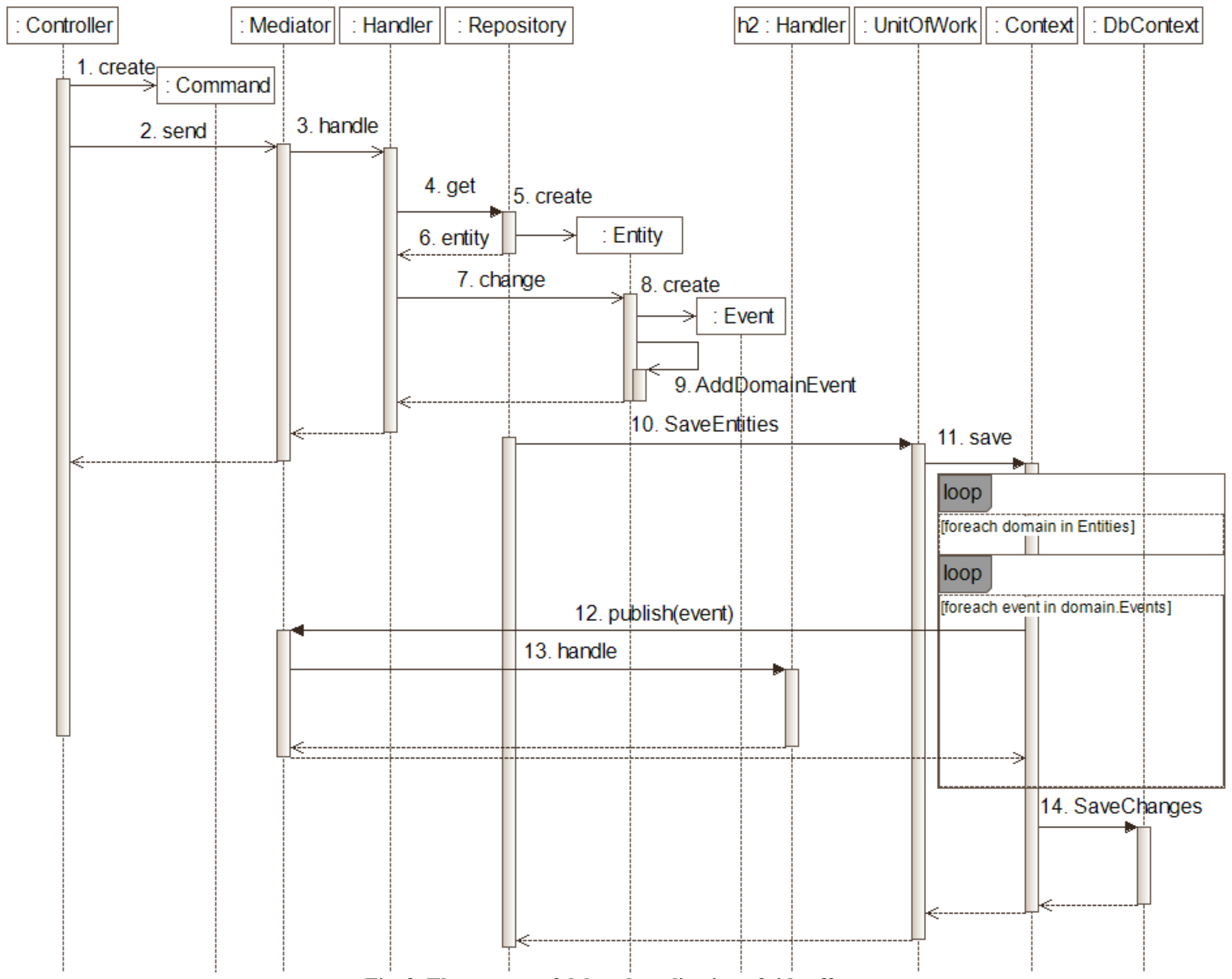


Fig. 3. The process of delayed application of side effects

The key points for understanding this process are the following: after changing the entity (Entity) in step 7, an event is generated in step 8, which is added to the collection of events of the aggregate (step 9), then when saving the aggregate in step 11, the objects are iterated of the context that were changed, the event objects published by the mediator (step 12) and processed by handlers (step 13) are selected from them. Handlers perform updates on context objects, thus propagating side effects. In the final step 14 of the process, the entire entity graph is stored in the repository (DbContext represents the base repository class and is provided by the Entity Framework .NET Core).

The entire stack of objects involved in this activity is located in the shared process memory, and the interaction occurs in synchronous mode.

Conclusions

The paper examines the main concepts of domain area events in domain-driven architecture. Models of immediate and delayed event distribution and features of their software implementation on the .NET Core platform were analyzed.

References

1. Fowler M. Patterns of Enterprise Application Architecture. — Addison-Wesley Professional. — 2003. — 560 p.
2. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. — Addison-Wesley Professional.— 2004. — 560 p.
3. Vernon V. Implementing Domain-Driven Design. — Addison-Wesley Professional. — 2013. — 656 p.
4. Millett, S., & Lippert, J. (2009). Patterns, Principles and Practices of Domain-Driven Design
5. Jimmy Nilsson. Applying Domain-Driven Design And Patterns: With Examples in C# and .net 1st Edition. (2006) Addison-Wesley Professional
6. Domain-Driven Design applied to land administration system development: Lessons from the Netherlands P Oukes, M Van Andel, E Folm
7. Jinsong Zhang, Yan Chen, Shengjun Qin. The Application of Domain-Driven Design in NMS. Proceedings of SPIE - The International Society for Optical Engineering (2011). DOI:10.1117/12.920133
8. Design microservices for drones [Electronic resource]. — Access Mode: https://learn.microsoft.com/en-us/azure/architecture/microservices/design — (Viewed 22.05.2023). — Title from the screen.
9. The Visual Collaboration Platform for Every Team | Miro [Electronic resource]. — Access Mode: https://miro.com/ — (Viewed 22.05.2023). — Title from the screen.
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
11. Domain Events – Take 2 [Electronic resource]. — Access Mode: https://udidahan.com/2008/08/25/domain-events-take-2/ — (Viewed 22.05.2023). — Title from the screen.
12. Immediate Domain Event Salvation with MediatR [Electronic resource]. — Access Mode: https://ardalis.com/immediate-domain-event-salvation-with-mediatr/ (Viewed 22.05.2023). — Title from the screen.
13. jbogard/MediatR: Simple, unambitious mediator implementation in .NET [Electronic resource]. — Access Mode: https://github.com/jbogard/MediatR — (Viewed 22.05.2023). — Title from the screen.
14. A better domain events pattern [Electronic resource]. — Access Mode: https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/ — (Viewed 22.05.2023). — Title from the screen.

Volodymyr Tykhokhod **Володимир Тихоход**	Ph.D, Senior Lecturer of Department of Automation of Design of Energy Processes and Systems, National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute” e-mail: tykhokhod@i.ua https://orcid.org/0000-0002-1525-4770	кандидат технічних наук, старший викладач кафедри автоматизації проектування енергетичних процесів і систем, Національний технічний університет України «Київський політехнічний інститут»
Anton Pasichniuk **Антон Пасічнюк**	Student, National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute” e-mail: pasichniuk@ukr.net	студент, Національний технічний університет України «Київський політехнічний інститут»