DOI 10.34185/1562-9945-1-138-2022-16
UDC 004.38

O.A. Lytvynov

ON TRANSACTIONAL BUSINESS LOGIC DESIGN

Abstract. The given work is devoted to the task of transactional business logic design. Additional layer responsible for transaction processing is introduced to make the architecture of the system more flexible and robust, testable and maintainable.
Key words: Domain Driven Design, business logic layer, Clean Architecture, transaction processing.

Actuality. Information system developers often face the task of implementing transactional business rules. And while the specific of transaction processing at the DBMS level is well-known (Read Committed, Repeatable Read etc.), the question of implementing transactions at the level of software components seems open and the decision on how to design and implement transactional business logic remained the prerogative of developers.

Today there are several possible practical implementations of transactional business logic: using **Unit Of Work** pattern (UoW) [1, 2]; using **SystemTransactions** library [3] provided by the .NET platform offered a ready-made solution (namely TransactionScope class); direct implementation of transactional logic using **stored procedures** stored within the database.

At first glance, the stored procedures variant seems to be the best solution of the problem, because stored procedures provide higher performance and security levels in comparison with others. But there is one problem that outweighs the benefits: *according to the method tasks for which business logic is responsible got smeared among several layers: database level, DAL (data access layer) and business logic layer*. Hiding business rules within stored procedures significantly reduces the system resistance to changes (e.g., in case of changing database provider developers face the necessity to rewrite all stored procedures, which, firstly, will take a lot of time, and secondly, it will not always be implemented effectively), and secondly, to understand the implementation of business logic rules, a developer cannot rely only on the appropriate layer (i. e. partly the rules will be implemented in the layer of business log-

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

ic, and partly in the stored procedure), which violates one of the most important robust software development principles - separation of concerns.

Unit of Work is a pattern provides a mechanism to escape the problems mentioned above by using logical transaction, i.e. atomicity of transaction is the responsibility of UoW object. Practically it means that we have a class responsible for transactions realization: committing or approving, rejecting or rolling back DB-oriented operations. Thus, the methods of the repositories responsible for entity changes, interact with that UoW object, which, in its turn, connects the commands to be executed to a given transaction (e.g. IDbTransaction realization). In this case, to commit the block of DB-oriented operations done by the methods of repositories we need to call commit method of UoW, and without that call the command block will not be executed. The advantage of the approach is that the transaction logic is concentrated in the business logic layer (application layer), meeting the separation of concerns principle. However, *the execution of each transaction requires creating a separate UoW with a set of dependencies (at least a bunch of repositories), which can lead to significant costs in a multi-user system: the number of objects associated with the processing of the transaction increases in direct proportion to the number of transactions, facing the risk of decreasing performance.* The use of ORM systems (NHibernate, Entity Framework etc.) seemed not always efficient: the logic of optimizing the execution of operations is not always transparent, and it is impossible to correct or modify the specifics of ORM transactions processing.

The TransactionScope class from the System.Transactions library provides an easy way to mark a block of code involved in a transaction hiding details of realization and making the definition of transaction as easy as it is possible. Due to its ease of use and efficiency, it is very convenient to use this mechanism to implement transactional business rules. TransactionScope has a number of advantages, namely: it makes the implementation of ACID transactions simple (i.e. no need to write an explicit "rollback" code or cleanup); it can coordinate resources such as databases, message queues and transaction file systems within one transaction; it automatically detects an external transaction and obtains its support. As a disadvantage, *it is "heavy" and absolutely hidden, so it cannot be an optimal variant for the systems focused on high performance.*

Task definition. Advantages and disadvantages of the above approaches led us to the idea of the solution which would be rather flexible and adoptable to the needs of software developers.

Main part. The solution is based on introduction of the additional Transaction Manager component responsible for transaction processing. This layer is used by the business logic layer and interacts with the data access layer (repositories) to perform data operations. It can be thought as an analogue of Entity services facade in terms of service-oriented architecture, or as a component superstructure above DAL layer responsible for transactions processing. The architecture of the system with the additional layer responsible for transactions processing is shown in Fig.1.

As we can see in the fig.1. Transaction Processing Layer can be connected not only to DAL but also to the Services and Utilities units to perform, for example, distributed transactions as necessary.

The question of its location is debatable: on the one hand, this component is a part of DAL, on the other it is a service of BLL. It is more correct to talk about some intermediate layer between DAL and BLL, which is responsible for implementing transactions. Each possible transaction is implemented as a method of this component. To perform a transaction that component uses a bunch of necessary repositories. Of course, these methods can only serve as access points (i.e. gateways) to the objects of the corresponding type, which are, in those turn, directly responsible for the specifics of transactions hiding implementation details and related resources (e.g., repositories).

The creation or, more specifically, assembly of the transaction manager seemed to be the task of a special class-factory, which is responsible for resolving the dependencies and providing all necessary resources to the manager (e.g. objects that implement specific transactions, repositories, etc.). In our opinion, providing of Transaction layer does not mean that all DB-oriented operations should be passed through that layer, simple operations can be performed by the repositories without any transaction mechanisms engagement and it differs provided approach from the approach of SOA Entity services [4]. From the DDD (domain-driven design) point of view, it can be considered as a service responsible for transaction processing. As we can see, the transaction manager is connected only with those repositories, which participate in the transaction processing.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

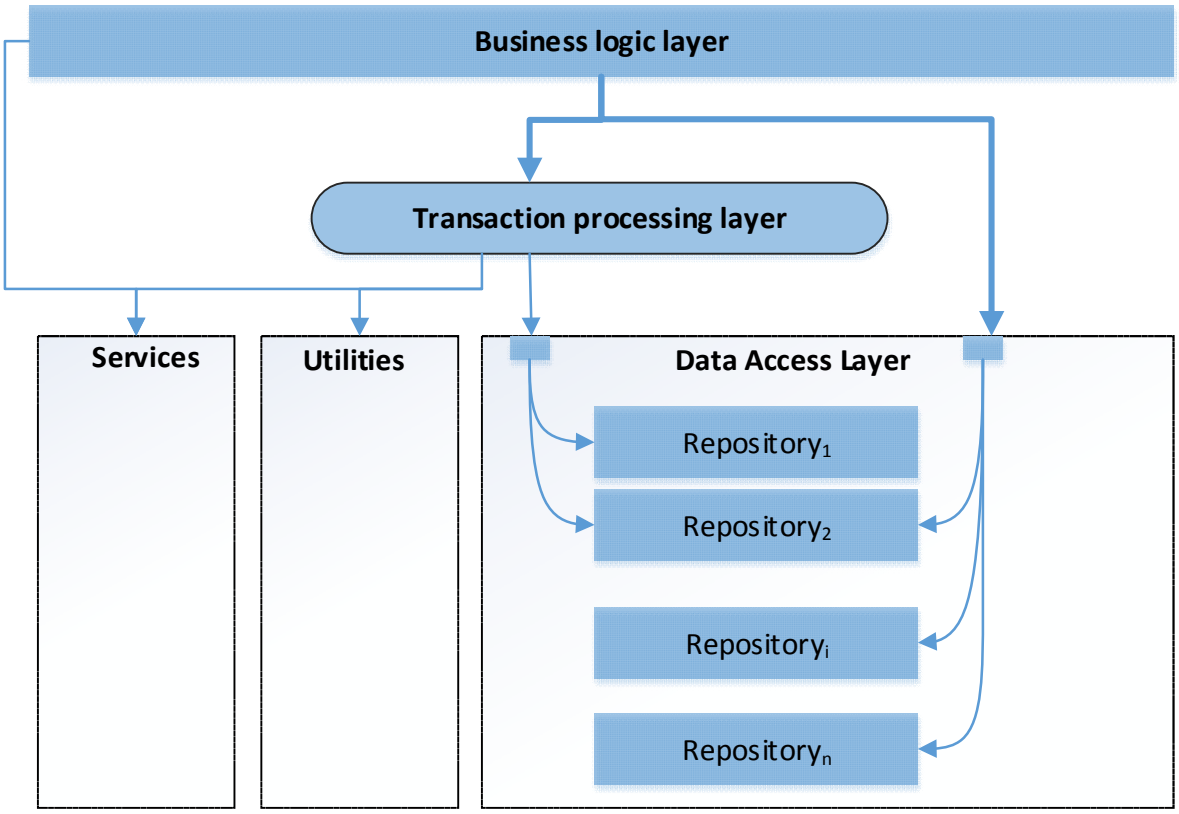

Figure 1 – The architecture of the system with the Transaction Processing Layer

To compare the classical business logic layer organization according to Clean Architecture approach [5] with the proposed one let us see fig.2 and Fig.3. It is suggested that RegisterBookUseCase depends on transaction which involves two repositories (IDeliveryRepository and IBookRepository) and a component responsible for logging transactional operations (ILogger). In this situation, according to [4], transactional logic processing will be the responsibility of the use case component. To change the transactional logic mechanism means to change each component it uses (e.g. to use TransactionScope instead of IDbTransaction). To test the transaction logic means to test use case component. The second solution (Fig.2) is seemed to be more flexible: the part responsible for transaction processing is separated from business logic and its modification and testing does not depend on the use case.

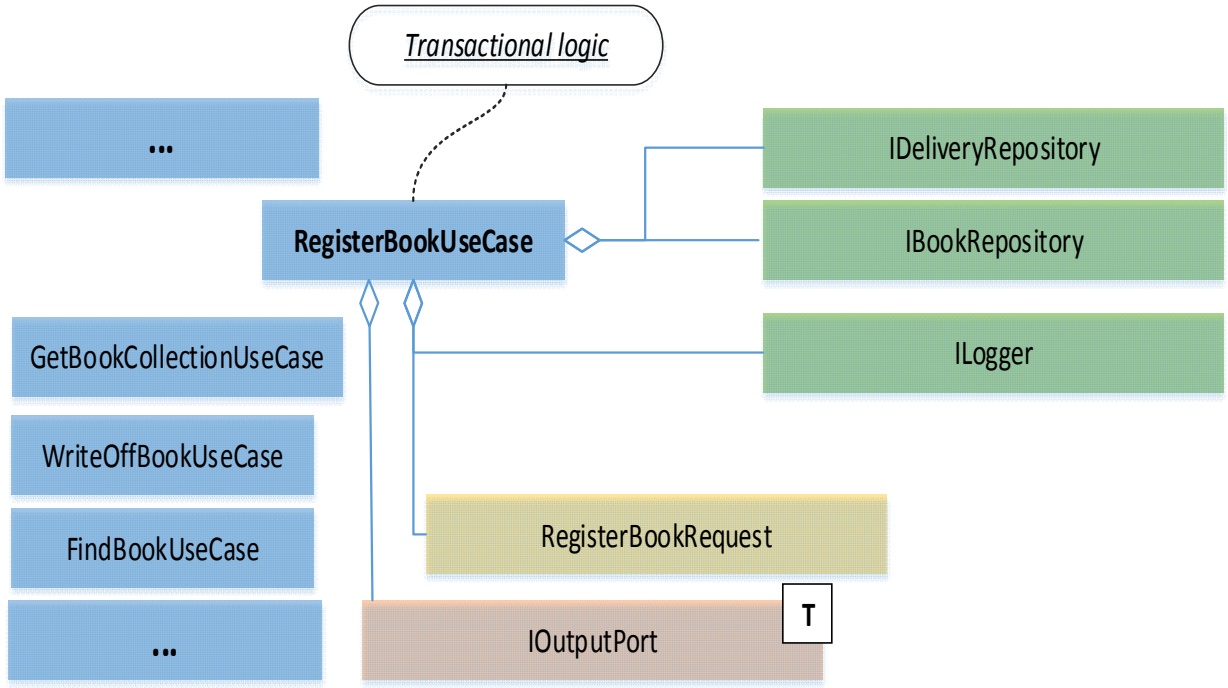

Figure 2 – Classical use case organization according to [5]

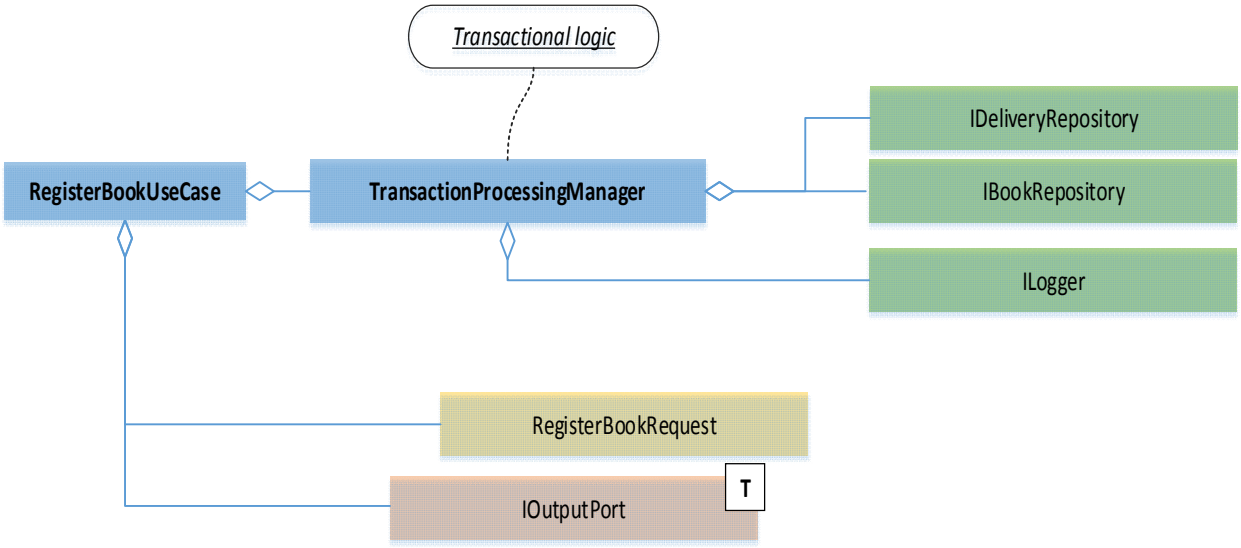

Figure 3 – Use case organization using Transaction processing layer

Since all requests for transaction implementation will pass only through one Transaction Manager object, there is an obvious need for asynchronous methods implementation. In the generalized variant, we can consider a variant of **transaction managers pool** that can be used to distribute the load, but even that variant seems more effective than UoW, where a full set of objects for each connection is created.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

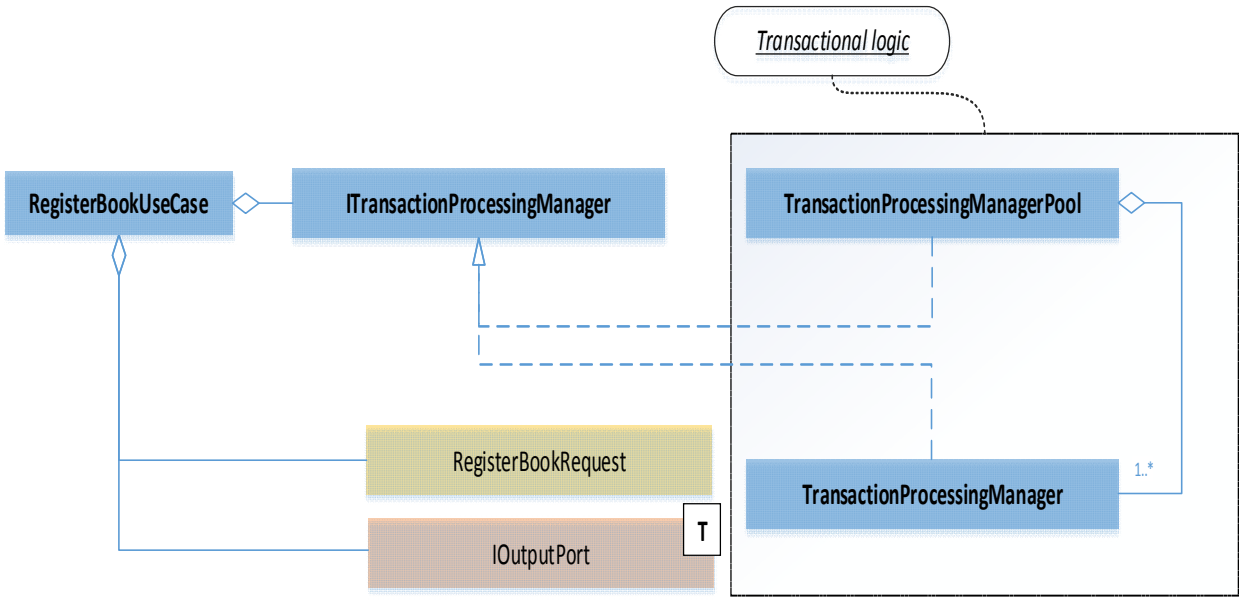

Figure 4 – Use case organization using Transaction processing managers pool

Another controversial issue is the possibility of having several transactional Managers (i.e., horizontal decomposition of the manager). The fact is that real projects have hundreds of tables in their databases and can reach up to a thousand objects and most of them can participate in transactions. However, it is rare that a use case or even a group of use cases of a particular type deals with hundreds of entities. Hence, a variant of distribution of transaction managers by contexts related to different types of users seems to be the best solution (Fig. 5).

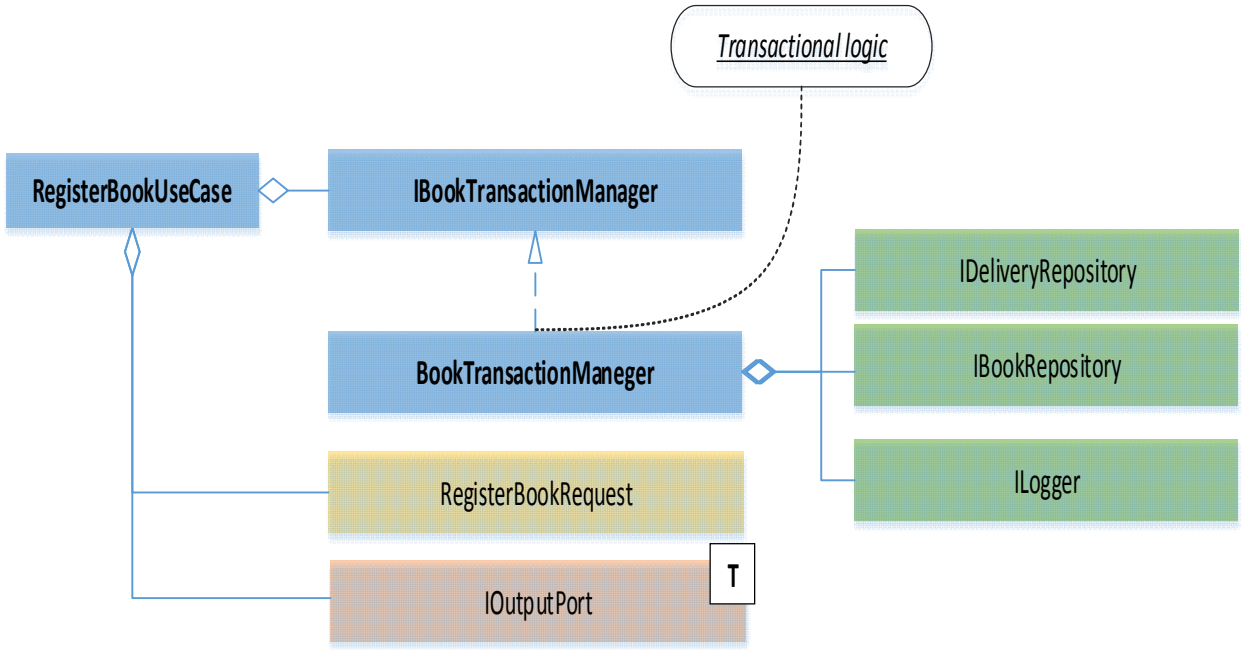

Figure 5 – A variant of transaction managers distribution

Conclusions. At present, transactions are a very powerful mechanism for implementing business rules and the success of their use depends on the approaches chosen by the developer. Each of the above approaches has its advantages and disadvantages. The choice depends on the subject area, business requirements and complexity of the project being developed. In this work, we consider the approach with introduction of additional layer of abstraction responsible for transaction processing that can be effectively applied if standard approaches are not suitable for some reason.

ЛІТЕРАТУРА

1. Eric Evans. “Domain-Driven Design: Tackling Complexity in the Heart of Software”. 2003.
2. Martin Fowler. P of EAA Catalogue. Unit of Work pattern. https://martinfowler.com/eaaCatalog/unitOfWork.html
3. О.А. Литвинов, В.С. Хандецький. Розподілена обробка інформації. ТОВ «Баланс-Клуб, с. 314.
4. Thomas Erl. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall; 2016.
5. Robert C. Martin. Clean Architecture. A Craftsman’s Guide to Software Structure and Design. 2018.

REFERENCES

1. Eric Evans. “Domain-Driven Design: Tackling Complexity in the Heart of Software”. 2003.
2. Martin Fowler. P of EAA Catalogue. Unit of Work pattern. https://martinfowler.com/eaaCatalog/unitOfWork.html
3. O.A. Lytvynov, V.S. Khandetskyi. Rospodilena obrobka informasii. TOV «Balans-Club, p. 314.
4. Thomas Erl. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall; 2016.
5. Robert C. Martin. Clean Architecture. A Craftsman’s Guide to Software Structure and Design. 2018.

Received 20.01.2022.
Accepted 25.01.2022.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Аспекти розробки транзакційної бізнес логіки

Актуальність. Перед розробниками інформаційних систем часто постає завдання впровадження транзакційної бізнес логіки. І хоча специфіка обробки транзакцій на рівні СУБД добре відома, питання реалізації транзакцій на рівні програмних компонентів зда ється відкритим, а рішення про те, як розробити та реалізувати транзакційну бізнес логіку. залишалася прерогативою розробників.

Сьогодні існує кілька можливих практичних реалізацій транзакційної бізнес логіки: використання шаблону Unit Of Work (UoW) [1, 2]; бібліотеки SystemTransactions [3]; без посередня реалізація транзакційної логіки за допомогою збережених процедур, що збері гаються в базі даних. Кожен з існуючих варіантів має свої переваги та недоліки. Так, при ховування бізнес правил у збережених процедурах значно знижує стійкість системи до змін, а по друге, щоб зрозуміти реалізацію правил бізнес логіки, розробник не може пок ладатися лише на відповідний рівень бізнес логіки (частина логіки буде знаходитися в шарі бізнес логіки, а частина у збережених процедурах). Використання шаблону UoW спрощує підтримку системи, збільшує її гнучкість та змогу адаптації до змін, але вико нання кожної транзакції вимагає створення окремого UoW з набором залежностей, що може призвести до значних витрат у багатокористувацькій системі. Клас TransactionScope з бібліотеки System.Transactions забезпечує простий спосіб позначити блок коду, який бере участь у транзакції, приховуючи деталі реалізації та максимально полегшуючи визначення транзакції, але є ресурсомістким компонетом зы скритою внут рішньою логікою, тому не може бути оптимальним варіантом для систем, орієнтованих на високу продуктивність..

Проблема. Переваги та недоліки вищезазначених підходів привели нас до ідеї рішен ня, яке було б досить гнучким і пристосованим до потреб розробників програмного за безпечення.

Основна частина. Рішення засноване на введенні додаткового компонента Transaction Manager, що відповідає за обробку транзакцій. Цей рівень використовується рівнем бізнес логіки і взаємодіє з рівнем доступу до даних (сховищами) для виконання операцій з даними. Його можна розглядати як компонентну надбудову над рівнем DAL, що відповідає за обробку транзакцій. Рівень обробки транзакцій можна підключити не тіль ки до DAL, а й до шарів служб та утиліт для виконання, наприклад, розподілених транза кцій, якщо це необхідно. Кожна можлива транзакція реалізована як метод цього компо нента. Для виконання транзакції цей компонент використовує купу необхідних репози торіїв. Ці методи можуть служити лише точками доступу (тобто шлюзами) до об'єктів відповідного типу, які, у свою чергу, безпосередньо відповідають за специфіку транзакцій, що приховують деталі реалізації та пов'язані ресурси (наприклад, репози торії).

Оскільки всі запити на реалізацію транзакцій будуть проходити тільки через один об'єкт Transaction Manager, існує очевидна потреба в реалізації асинхронних методів. В узагальненому варіанті ми можемо розглянути варіант пулу менеджерів транзакцій, який можна використовувати для розподілу навантаження.

Іншим питанням є можливість наявності декількох транзакційних менеджерів (тобто горизонтальна декомпозиція менеджера). Справа в тому, що реальні проекти мають сотні таблиць у своїх базах даних і можуть досягати тисячі об'єктів, і більшість з них можуть брати участь в транзакціях. Але ситуація, у якій use case (прецедент) або навіть група прецедентів використовує для реалізації сотні сутностей трапляється рідко.

Висновки. В даний час транзакції є дуже потужним механізмом впровадження бізнес правил і успіх їх використання залежить від підходів, обраних розробником. У цій роботі ми пропануємо підхід із введенням додаткового рівня абстракції, що відповідає за обробку транзакцій, який можна ефективно застосувати, якщо стандартні підходи з якихось причин не підходять.

Литвинов Олександр Анатолійович - кандидат технічних наук, доцент кафедри електронних обчислювальних машин Дніпропетровського національного університету ім. О. Гончара.

Lytvynov Olekcandr Anatoliyovich - candidate of technical sciences, associate professor, associate professor of the department of Electronic Computing Machinery, Oles Honchar Dnipro National University.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)