УДК 004.6

ПРОЕКТИРОВАНИЕ, УПРАВЛЯЕМОЕ ПРЕДМЕТНОЙ ОБЛАСТЬЮ: ПОЛОЖИТЕЛЬНЫЕ И ОТРИЦАТЕЛЬНЫЕ АСПЕКТЫ

А.А. Файтельсон
Бакалавр второго года обучения по направлению подготовки «Математическое обеспечение и администрирование информационных систем»
Курский государственный университет
e-mail: z0tedd@gmail.com

Научный руководитель:
А.В. Кривонос
Кандидат технических наук, доцент кафедры программного обеспечения и администрирования информационных систем
Курский государственный университет
e-mail: krivonos_av@kursksu.ru

Статья посвящена исследованию подхода проектирования, управляемого предметной областью (Domain-Driven Design, DDD), который становится все более актуальным в разработке программного обеспечения. Основная цель DDD — устранение разрыва между бизнесом и разработкой, позволяя командам создавать более точные и поддерживаемые системы, глубоко интегрированные с бизнес-процессами.

Ключевые слова: *Domain-Driven Design, DDD, разработка программного обеспечения, архитектура.*

Введение. Подход Domain-Driven Design (DDD) был предложен Эриком Эвансом [1] в начале 2000-х годов как ответ на растущие сложности в разработке корпоративного ПО. Основная идея DDD заключается в том, чтобы максимально сосредоточиться на моделировании предметной области — совокупности бизнес-процессов и правил, характерных для конкретной компании или организации. DDD стремится устранить разрыв между бизнесом и разработкой, предлагая общий язык и методики, которые делают модели системы понятными для всех участников процесса, от разработчиков до бизнес-аналитиков и заказчиков.

DDD становится незаменимым при работе со сложными системами, где важна высокая точность в передаче и обработке данных. Использование DDD позволяет разработчикам лучше понимать суть бизнес-процессов, выделяя ключевые сущности и связи, которые играют важную роль для компании. Таким образом, разработка ПО идет не «в отрыве» от бизнес-логики, а в тесной интеграции с ней, что помогает создавать ПО, действительно решающее задачи бизнеса.

Преимущества DDD для разработки программного обеспечения. Одним из главных преимуществ DDD является возможность наладить взаимодействие между технической и бизнес-сторонами. Используя DDD, компании формируют «общее видение» — когда разработчики, бизнес-аналитики и

представители заказчика начинают работать как единая команда. Это общее понимание достигается благодаря созданию "общего языка" (Ubiquitous Language), который используется в повседневном общении и отражается в коде, моделях и документации. Общий язык позволяет снизить количество ошибок, вызванных недопониманием требований, и сделать требования к ПО более однозначными и структурированными.

DDD помогает значительно улучшить качество кода и архитектуры. [2] Код структурируется так, что его легче поддерживать, он становится более устойчивым к изменениям. Благодаря выделению основных сущностей и паттернов, DDD-система остается логичной и предсказуемой. К тому же этот подход снижает "долг по документации", так как сама структура кода и модели служат наглядной документацией системы.

Основные концепции и элементы DDD В этой части подробно раскрываются основные компоненты DDD, формирующие основу для построения четкой и устойчивой архитектуры. В основе DDD лежит доменная модель, которая описывает предметную область и отражает реальные бизнес-процессы. Доменные модели создаются в тесном сотрудничестве с бизнесом, чтобы максимально точно передавать смысл и цели, которые должны решаться в приложении. Такие модели становятся своего рода "картой" для разработки, обеспечивая разработчикам четкое представление о структуре и задачах системы.

DDD выделяет три основных элемента модели, каждый из которых выполняет свою роль [3]:

1. Сущности (Entities) - объекты с уникальной идентичностью, которые изменяются во времени, но остаются узнаваемыми (например, клиент в CRM-системе).
2. Объекты-значения (Value Objects) - объекты, не имеющие уникальной идентичности, но содержащие данные (например, адрес). Они неизменяемы и применяются для группировки значений.
3. Агрегаты (Aggregates) - совокупности сущностей и объектов-значений, объединенные вокруг главной сущности, называемой корнем агрегата (Aggregate Root). Агрегаты помогают контролировать целостность данных и обеспечивать атомарные операции.

Доменные события фиксируют важные моменты в жизненном цикле объекта или процесса. Это не просто данные, но и сигнал для системы о том, что произошло значимое изменение. Репозитории, в свою очередь, обеспечивают доступ к данным агрегатов и позволяют реализовать принципы DDD, сохраняя доменную модель независимой от деталей хранения данных.

При интеграции с внешними системами DDD предлагает использовать анти-коррупционные слои, чтобы избежать прямого воздействия внешней логики на внутреннюю модель предметной области. Это особенно важно для защиты предметной области от нежелательных влияний, позволяя системе сохранять устойчивость и единообразие.

Для корпоративных проектов, где одновременно работают большие команды и управляются обширные потоки данных, DDD может стать важным стратегическим решением. Один из основных плюсов DDD — это упрощение масштабирования системы: когда бизнес растет, модель может адаптироваться к новым требованиям без радикальной перестройки. В корпоративных условиях, где риски при внедрении нового ПО и изменения существующего кода особенно высоки, DDD позволяет минимизировать возможные ошибки и снизить затраты на поддержку.

DDD помогает адаптировать систему под уникальные бизнес-процессы и требования конкретной компании. В больших проектах часто наблюдается высокая сложность взаимодействий между различными отделами и сервисами, и доменная модель помогает организовать их более гибко и точно. Также, благодаря анти-коррупционным слоям, DDD позволяет легко интегрировать систему с устаревшими сервисами, что часто необходимо в корпоративных ИТ-инфраструктурах.

Несмотря на значительные преимущества, DDD не лишен сложностей. Одной из главных трудностей является высокая начальная стоимость внедрения. Потребуется время и ресурсы на обучение сотрудников, [4] так как для успешного применения DDD разработчики и аналитики должны обладать глубокими знаниями предметной области. При этом подход может показаться чрезмерно сложным для небольших или простых проектов, где его полное внедрение будет неоправданным.

Кроме того, DDD требует активного участия бизнеса на всех стадиях проекта. Компании, где коммуникация между бизнесом и разработкой затруднена, могут столкнуться с тем, что сложность доменной модели станет препятствием для эффективного внедрения. Интеграция с устаревшими системами также может быть вызовом, так как данные, не всегда соответствующие требованиям DDD, усложняют разработку анти-коррупционных слоев. [5]

DDD и Agile — подходы, которые могут дополнять друг друга, создавая гибкие и устойчивые системы. Agile ориентирован на итеративное развитие, постоянное улучшение и адаптацию системы к изменениям, и DDD хорошо вписывается в эти принципы, так как способствует адаптации системы к изменяющимся требованиям бизнеса. Важно, что обе методологии делают акцент на взаимодействие между командой разработки и бизнесом, что способствует взаимопониманию и снижению рисков при внесении изменений.

Однако, DDD требует тщательного анализа предметной области и создания модели, что может вступать в противоречие с Agile, когда итерации начинаются с минимального анализа и запускаются на раннем этапе. Совмещение этих подходов требует баланса: например, начинать проект с базовой модели, которую можно углублять и расширять в ходе итераций, одновременно с регулярными проверками согласованности доменной модели с актуальными требованиями бизнеса.

Несмотря на эффективность DDD, многие команды допускают ошибки при его внедрении. Одна из самых распространенных — излишне подробное моделирование, когда проект тратит ресурсы на проработку всех деталей до-

менной модели. Этот подход, известный как "Большой дизайн впереди" (Big Design Up Front), противоречит принципу адаптивного проектирования и делает систему слишком громоздкой и трудной для изменений.

Еще одной ошибкой является неправильное понимание общих принципов DDD, когда разработчики путают доменные модели с техническими компонентами системы. Важно понимать, что доменные модели должны полностью отражать логику предметной области, а не технические аспекты. Это требует плотного взаимодействия с бизнесом и правильного распределения задач между командой разработки и аналитиками.

Здесь приводятся реальные примеры применения DDD в крупных компаниях, где подход показал высокую эффективность. Кейс-стади демонстрируют, как доменные модели могут улучшать работу систем и повышать гибкость при изменениях требований. Также рассматриваются различные отрасли, где DDD применяется: от финансов и логистики до здравоохранения и ритейла. Успешные внедрения показывают, как DDD позволил компании избежать дорогостоящих ошибок и создать систему, соответствующую ее задачам.

С учетом современных технологий (например, микросервисов, облачных платформ и искусственного интеллекта) DDD продолжает развиваться, адаптируясь к новым требованиям. Доменные события и событийное взаимодействие (Event-Driven Design) становятся все более популярными, позволяя применять DDD в распределенных системах. Технологии продолжают влиять на подход DDD, и будущее показывает, что его роль будет расти в условиях гибридных архитектур и усложняющихся бизнес-процессов.

DDD подходит не всем проектам, но для систем, где важна точность в передаче бизнес-логики и долговременная поддержка, он может стать идеальным решением. Заключение подводит итоги, описывая сильные и слабые стороны DDD, и дает рекомендации для выбора этого подхода, а также совета по адаптации DDD для проектов различного масштаба.

Список используемой литературы:

1. Эванс, Эрик. Domain-Driven Design: Проектирование, управляемое предметной областью. Питер, 2010.
2. Вернон, Вон. Реализация Domain-Driven Design. ДМК Пресс, 2014.
3. Р. Дандан. Методические материалы по формированию карты контекстов. // International journal of humanities and natural sciences, vol. 5-2, 2022
4. Вахлова, Ольга. Программирование, управляемое предметной областью (DDD). БХВ-Петербург, 2010.
5. Бурдуков В.П. Domain-Driven Design: преимущества и недостатки методологии, управляемой предметной область. // "Научный аспект №7-2024" - Информ. технологии

GRNTI 20.53.24

DOMAIN-DRIVEN DESIGN: POSITIVE AND NEGATIVE ASPECTS

A.A. Faitelson
Bachelor of the second year of study in the direction of training "Software technology and administration of information systems"
Kursk State University
e-mail: z0tedd@gmail.com

Scientific supervisor:
A.V. Krivonos
PhD of Technical Sciences, Associate Professor of the Department of Software and Administration of Information Systems
Kursk State University
e-mail: krivonos_av@kursksu.ru

The article is devoted to the consideration of data-oriented programming, positive and negative aspects of this paradigm. This paradigm is intended to help in the design of information systems

***Keywords**: data-oriented programming, programming paradigms, design of information systems*