УДК 004.045 ***DOI: 10.46960/1816-210X_2023_4_20***

О МЕСТЕ МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ В СТРУКТУРЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРИЛОЖЕНИЯ

С.В. Логанов
ORCID: **0000-0002-7302-4586** e-mail: **loganovserg@yandex.ru**
Нижегородский государственный технический университет им. Р.Е. Алексеева
Нижний Новгород, Россия

Обосновано, что все зависимости слоев архитектуры разрабатываемой программной системы должны быть направлены в сторону модели предметной области, а интерфейсы необходимо вводить лишь по мере появления дополнительных требований к масштабированию и мультиплатформенности. Актуальность работы определяется тем, что традиционная трехслойная архитектура обладает существенным ограничением – транзитивной зависимостью от нижележащего слоя. Применение принципа инверсии зависимостей к нижележащему слою вынуждает преждевременно проектировать интерфейсы, которые не позволяют полноценно решить задачу заменяемости слоев в условиях недостаточно точно сформулированных требований. Преимущества объектно-ориентированного программирования могут быть полноценно реализованы только при проектировании архитектуры на основе модели предметной области.

Ключевые слова: объектно-ориентированное программирование, объектно-ориентированное проектирование, архитектура ПО, принцип DIP, принципы проектирования компонентов.

ДЛЯ ЦИТИРОВАНИЯ: Логанов, С.В. О месте модели предметной области в структуре объектно-ориентированного приложения // Труды НГТУ им. Р.Е. Алексеева. 2023. № 4. С. 20-26.
DOI: 10.46960/1816-210X_2023_4_20

POSITION OF THE DOMAIN MODEL IN OBJECT-ORIENTED APPLICATION ARCHITECTURE

S.V. Loganov
ORCID: **0000-0002-7302-4586** e-mail: **loganovserg@yandex.ru**
Nizhny Novgorod state technical university n.a. R.E. Alekseev
Nizhny Novgorod, Russia

Abstract. All dependencies of the architecture layers of the developed software system should be directed towards the domain model, while interfaces should be introduced only when an additional requirement for scaling and multiplatform appear. The relevance of the study lies in the fact that traditional three-layer architecture has a significant limitation, namely, a transitive dependence on the underlying layer. The principle of dependency inversion is applied to the underlying layer, which forces premature design of interfaces that do not allow to fully solve the problem of layer interchangeability in conditions of insufficiently precisely formulated requirements. The benefits of object-oriented programming can only be fully realized when designing a domain model-based architecture.

Key words: object-oriented programming, object-oriented design, software architecture, DIP principle, package principles.

FOR CITATION: Loganov S.V. Position of the domain model in object-oriented application architecture. Transactions of NNSTU n.a. R.E. Alekseev. 2023. № 4. Pp. 20-26. DOI: 10.46960/1816-210X_2023_4_20

Введение

Объектно-ориентированное программирование (ООП) разрабатывалось как возможность программировать не в терминах вычислительной системы, а в терминах естественной предметной области, на которую направлено разрабатываемое приложение. Следовательно,

достоинства ООП в полной мере могут проявиться лишь при проектировании архитектуры приложения на основе модели предметной области. Данная модель в высшей степени подходит для приложений большой сложности, поскольку максимально использует возможности объектной ориентации и облегчает получение подлинного соответствия решаемым задачам.

Архитектура построения объектно-ориентированных приложений существенно влияет как на качество функционирования программной системы, так и на эффективность ее поддержки и масштабирования. Построение гибкой архитектуры позволяет выполнять эффективную модернизацию программной системы, поддерживая ее постоянную актуальность. Традиционное расположение модели предметной области в трехслойной архитектуре в качестве слоя бизнес-логики приводит к ее зависимости от слоя управления данными [1], что препятствует как эффективной модернизации самой бизнес-логики, так и использованию дополнительной инфраструктуры.

Построение структуры объектно-ориентированных приложений

Типовая архитектура программной системы предполагает ее расслоение на три уровня [2] (рис. 1). Ее основной особенностью является использование слоя верхнего уровня услуг, предоставляемых нижележащим уровнем, никак не связанным с верхним. Таким образом, от вышележащих уровней скрыты дополнительные нижележащие слои, позволяющие поэтапно выполнить сложную задачу. При этом отдельный слой воспринимается как самодостаточное целое, которое может базироваться на различной реализации нижележащих слоев.

Рис. 1. Пример традиционного разбиения системы на слои

Fig. 1. An example of the traditional splitting the system into layers

Кардинальным преимуществом трехслойной архитектуры является возможность замены нижележащих слоев. Однако замена слоя бизнес-логики – основной ценности программной системы – обоснована только для решения конкретных задач. Иными словами, нет необходимости обеспечивать заменяемость бизнес-логики для каждой программной системы. Помимо этого, простое разбиение программной системы на слои приводит к их транзитивной зависимости друг от друга [1]. Существуют слои высокого уровня, устанавливающие определенные бизнес-правила, но на них влияют детали реализации слоев нижнего уровня. Если слои высокого уровня зависят от нижележащих, довольно сложно использовать их в различных контекстах, расширяя выпуск полезных приложений на новых платформах. Поэтому слои, содержащие описания и реализующие бизнес-правила высокого уровня, должны иметь определенную независимость и приоритет перед слоями, обеспечивающими выполнение функций обслуживания.

В качестве решения данной проблемы предлагается проведение границ между слоями, т.е. использование принципа инверсии зависимостей (DIP), в котором каждый слой верхнего уровня объявляет абстрактный интерфейс для необходимых служб, а каждый нижележащий

слой выполняет его реализацию (рис. 2) [3]. Это разрывает жесткую зависимость слоев верхнего уровня от слоев нижнего уровней: нижележащие слои и соответствующие им компоненты зависят от абстрактных служебных интерфейсов, объявленных в верхних слоях.

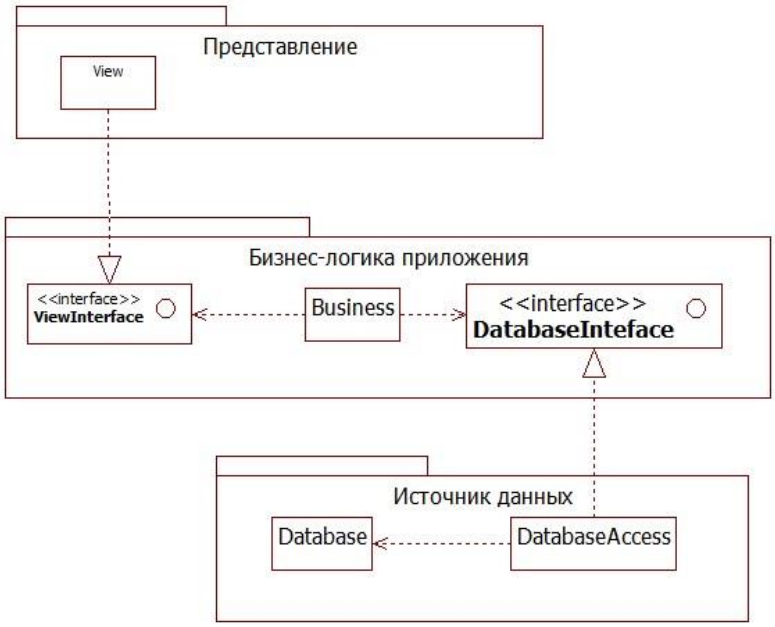

Рис 2. Разбиение системы на слои с использованием DIP

Fig. 2. Splitting the system into layers using DIP

Однако в данном случае появляется обратная зависимость, согласно которой невозможно разработать слой источника данных или представления без знания интерфейсов слоя бизнес-логики. Таким образом, графические библиотеки и библиотеки доступа к данным невозможно разработать без знания соответствующих интерфейсов и под каждый новый слой бизнес-логики придется разрабатывать новую библиотеку. Поэтому компоненты доступа к данным и представления должны реализовывать собственные интерфейсы, соответствующие тем задачам, которые они выполняют. При этом необходимо наличие связующих слоев, выполняющих реализацию задач представления и хранения с помощью готовых библиотек или фреймворков. Тем не менее, высокоуровневый слой бизнес-логики должен быть всегда изолирован (независим) от деталей его представления или сохранения с помощью различных промежуточных слоев; ввод интерфейсов также предполагает заменяемость слоев представления и источника данных. На ранних этапах разработки часто неизвестно, насколько популярным окажется программный продукт, соответственно, неизвестны и требования к его масштабируемости и многоплатформенности. Поэтому отсутствие таких требований не позволяет качественно спроектировать соответствующие интерфейсы, либо приводит к огромным затратам при построении решения для всех возможных случаев.

В [4] представлена более простая модель архитектуры программной системы, в которой все зависимости просто обращены к модели предметной области без обязательного использования интерфейсов (рис. 3). Здесь бизнес-слой разделен на две независимые части: слой приложения и слой модели предметной области. В модели предметной области определяются бизнес-объекты, которые обеспечивают возможность решения бизнес-задач, соблюдения бизнес-правил и структуру взаимосвязей этих бизнес-объектов. Таким образом, модель предметной области определяет пространство решений, а слой приложения определяет, как выполняется та или иная конкретная задача с помощью данной модели (пространство реализации). В слое приложения располагается набор классов, которые реализуют выполнение пользовательских сценариев использования, и которые предлагается называть контроллерами системных событий [5]. В [3] рассматривается аналог таких контроллеров (интеракторов), каждый из которых соответствует ровно одному пользовательскому сценарию. В [5]

приведена более развитая классификация котроллеров, разделяющая системные события на пять видов. Слоеная архитектура допускает обращение верхних слоев к нижним, минуя некоторые слои, однако нужно учитывать, что при этом минимизируется принципиальная возможность замены нижележащего слоя. На рис. 3 также отсутствуют связующие слои, изолирующие бизнес-логику от используемых фреймворков, без которых невозможна разработка ни одной современной системы.

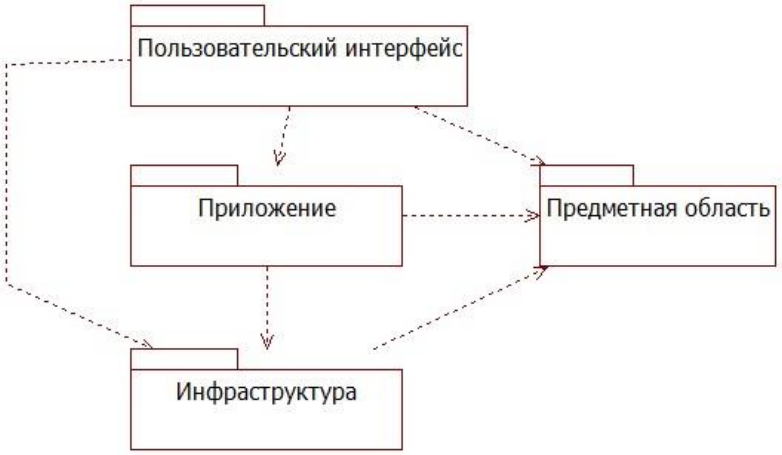

Рис. 3. Разбиение системы на слои с использованием зависимости от предметной области

Fig. 3. Splitting the system into layers using dependence on domain model

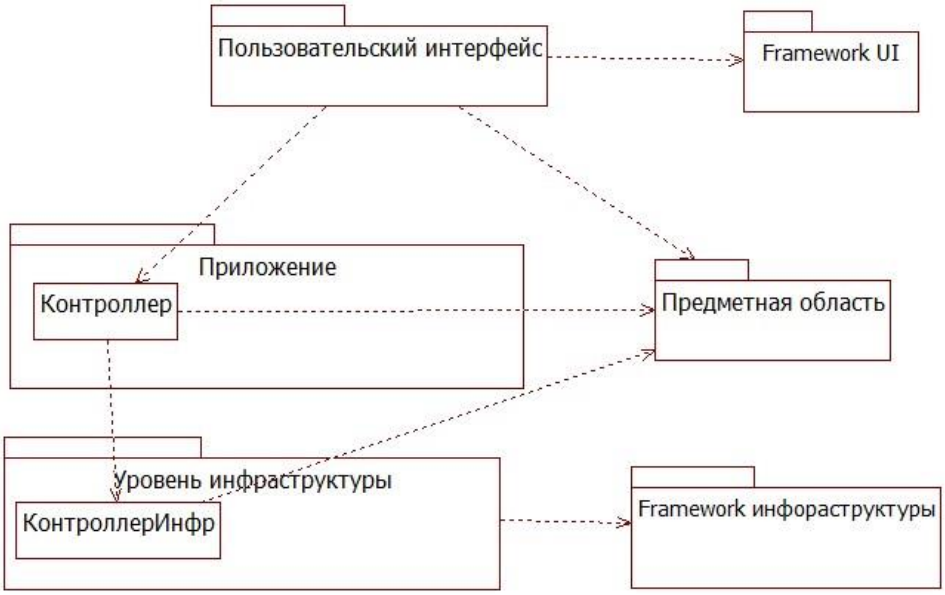

Рис. 4. Архитектура программной системы, не требующей замены обслуживающих слоев

Fig. 4. The architecture of a software that does not require the replacement of service layers

На рис. 4 приведена архитектура системы, в которой связующие слои пользовательского интерфейса и инфраструктуры непосредственно зависят от модели предметной области. Прямая зависимость от модели предметной области вполне оправдана в силу трех причин. Во-первых, программная система разрабатывается ради реализации соответствующих бизнес-правил и решения задач некоторого вида бизнеса и, следовательно, не нуждается в повторном использовании, поскольку для другого вида бизнеса следует разрабатывать другую программную систему. Во-вторых, соответствующий вид бизнеса является достаточно стабильным и, как правило, редко претерпевает кардинальные изменения. Поэтому, согласно

принципу стабилизации зависимостей, зависимость от модели предметной области является оправданной. В-третьих, элементы бизнес-модели, которые подвергаются частым изменениям, как правило, известны уже заранее, и для них могут быть построены соответствующие решения, допускающие их расширение.

Связующий слой пользовательского интерфейса зависит как от модели предметной области, так и от контроллеров системных событий, которые реализуют логику выполнения сценариев использования с помощью бизнес-объектов. Таким образом, пользовательский интерфейс занимается отображением и представлением бизнес-объектов, а за выполнением каких-либо действий обращается к слою приложения. Переход на другую модель представления или к пакетной обработке приведет к необходимости разработки нового слоя пользовательского интерфейса с использованием соответствующего фреймворка, а расширение способов использования – к разработке новых контроллеров, обеспечивающих выполнение новых сценариев. Контроллеры, расположенные в слое приложения для выполнения задач сохранения и восстановления бизнес-объектов, могут и должны обращаться к контроллеру инфраструктуры, который должен быть единственным и являться фасадом уровня инфраструктуры. Уровень инфраструктуры получает, сохраняет и возвращает бизнес-объекты модели предметной области. При этом слой модели предметной области должен содержать достаточное количество конструкторов бизнес-объектов в оперативной памяти, либо соответствующие фабрики таких объектов, имеющих сложную структуру для инициализации, а слой инфраструктуры обеспечит изменение их состояния, в соответствии с полученной или прочитанной информацией из какого-либо источника. Предполагается, что современные каркасы объектно-реляционного отображения могут оперировать простыми объектами *POJO* (*Plain Old Java Object*) или *POCO* (*Plain Old CLR Object)*, однако они все равно требуют указания дополнительных атрибутов, которые не позволяют сделать их полностью независимыми от инфраструктуры сохранения. Поэтому инфраструктурный слой должен восстанавливать и сохранять именно бизнес-объекты. Данная структура не делает ничего лишнего, но позволяет подготовится к дальнейшему масштабированию. Приложению обычно лучше подходит код, специфичный для проекта и его модели предметной области. Абстрактные универсальные решения не представляют практической ценности, и разработчикам проекта лучше иметь код, который распознает ожидаемое поведение приложения и концепции предметной области [6].

При появлении новых требований к использованию различных инфраструктурных слоев необходимо выполнить анализ публичного интерфейса контроллера инфраструктуры, на основе которого, в зависимости от появившихся требований, в слое приложения создаются один или несколько интерфейсов. Необходимость второго, а лучше третьего заменяемого инфраструктурного слоя позволяет более качественно спроектировать такие интерфейсы и существенно снизить возможные неявные зависимости. Кроме этого, наличие дополнительных требований позволяет эффективнее определить состав связующих слоев, формируя на их основе отдельные пакеты. При необходимости построения распределенного приложения и отдельного масштабирования его частей потребуется очередная модернизация архитектуры его построения за счет разделения ее на физические компоненты, выполняющиеся в отдельных процессах (рис. 5).

Поскольку между различными процессами, особенно удаленными, возможна и оптимальна передача только информации, необходимо проектирование *API* (*Application Program Interface*) бизнес-слоя, которое принимает и получает команды и данные. При этом *API* лучше сразу строить на основе шаблона разделения команд и запросов *CQRS* (*Command Query Responsibility Segregation*), который позволяет повысить масштабируемость получаемого решения. Для различных слоев пользовательского интерфейса необходим свой интерпретатор команд и данных, который позволяет запускать и получать результаты выполнения соответствующих сценариев использования приложения. Данный интерпретатор команд получив необходимую команду формирует требуемые бизнес-объекты и обращается к соответствую-

щему контроллеру приложения для ее выполнения и получения ее результатов. Далее бизнес-объекты преобразуются в информационные объекты *XML, JSON* или объекты (сообщения) какого-либо другого формата, которые и передаются в качестве результатов выполнения соответствующей команды.

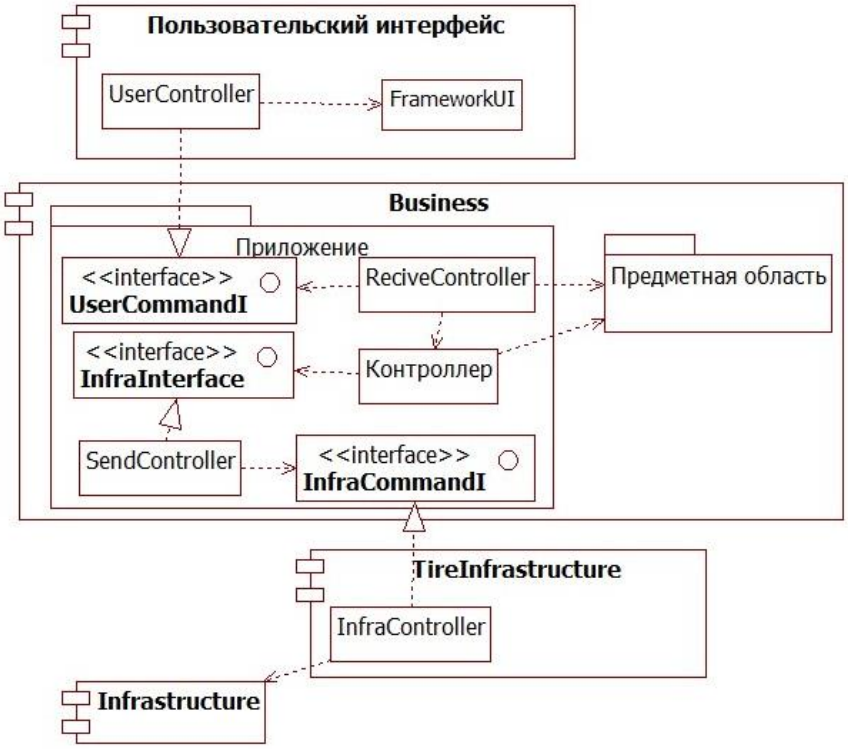

Рис. 5. Разбиение системы на слои функционирующих в различных процессах

Fig. 5. Splitting the system into layers functioning in various processes

Аналогичную процедуру в обратном направления выполняет интерпретатор команд приложения, который на основе требуемых действий и соответствующих бизнес-объектов формирует инфраструктурные команды, содержащие необходимую информацию. При этом соответствующий промежуточный слой инфраструктуры, возможно и не один, выполняет данные команды, возвращая необходимую информацию, которая затем преобразуется в соответствующие бизнес-объекты. При этом интерпретаторы команд пользовательского и инфраструктурных слоев или их соответствующее *API* не следует объединять в единое целое, поскольку они имеют принципиально разные причины для изменения и, согласно принципу *SRP*, должны быть разделены. При наличии слишком сложной предметной области и необходимости раздельного масштабирования отдельных частей бизнес-логики, т.е. разделения ее на отдельные независимые вычислительные процессы, необходимо переходить на т.н. гексагональную архитектуру [7]; в ней модель предметной области может быть разделена на отдельные относительно независимые подобласти, взаимодействие между которыми осуществляется с помощью событий предметной области. Соответственно, появляется необходимость построения дополнительных портов и адаптеров (т.е. интерфейсов и контроллеров команд) для обработки событий предметной области. Количество таких интерпретаторов команд или соответствующих *API* будет зависеть от количества подобластей и их организации в соответствующие иерархии. Термин *гексагональная архитектура* в большей мере свидетельствует о необходимости множества портов и адаптеров для объединения множества подмоделей предметной области и связующих слоев в единую распределенную систему, чем о конкретном их количестве. Следовательно, чем больше подобластей предполагает деление модели предметной области, тем больше промежуточных слоев, содержащих порты и адап-

теры, требуется разрабатывать, поскольку каждая подобласть выделена в отдельный компонент (сервис) и, соответственно, имеет свои причины для изменения.

Таким образом, по мере развития и распространения бизнес-приложения может возникнуть потребность универсализации и независимого масштабирования его частей, которые приводят к существенному увеличению затрат на его проектирование, реализацию, развертывание и поддержку. Труднее всего при этом изменять интерфейсы и *API* бизнес-логики, которые обеспечивают взаимодействие приложения с окружающим миром. Можно заключить, что преждевременное принятия решений в условиях неопределенности, делает успех маловероятным.

Выводы

Программная система оценивается с точки зрения ее поведения и структуры. Разработчики, как правило, сосредотачиваются на реализации необходимого поведения, не уделяя достаточного внимания архитектуре построения программной системы. Однако для обеспечения длительного жизненного цикла ПО важна возможность его легко изменить и масштабировать. Соответственно, проектирование архитектуры системы должно проводиться только на основе актуальных на данный момент бизнес-требований. Проработка архитектуры позволяет выбрать подходящее для актуальных задач решение, снизив финансовые затраты на разработку.

С другой стороны, проектирование объектно-ориентированной программы является процессом постепенных открытий, зависящих от оперативной обратной связи [8]. В противном случае, при проектировании очень гибкой архитектуры без достаточных оснований, неизбежны неоправданные финансовые затраты и снижение эффективности решения. Грамотные практики построения архитектуры программной системы позволят получить фундамент, на котором возможно дальнейшее масштабирование полученного решения.

Библиографический список

1. **Мартин, Р.** Принципы, паттерны и методики гибкой разработки на языке C# / Р. Мартин, М. Мартин. – СПб.: Символ-Плюс, 2011. – 768 с.
2. **Фаулер, М.** Архитектура корпоративных программных приложений / М. Фаулер. – М.: ИД «Вильямс», 2006. – 544 с.
3. **Мартин, Р.** Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин. – СПб.: Питер, 2018. – 352 с.
4. **Нильсон, Д.** Применение DDD и шаблонов проектирования. Проблемно-ориентированное проектирование приложений с примерами на C# и .NET / Д. Нильсон – М.: ООО И.Д. «Вильямс», 2008. – 560 с.
5. **Ларман, К.** Применение UML2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ и проектирование: учеб. пособие / К. Ларман. – М.: ИД «Вильямс», 2008. – 736 с.
6. **Нобак, М.** Принципы разработки программных пакетов: Проектирование повторно используемых компонентов / М. Нобак. – М.: ДМК Пресс, 2020. – 274 с.
7. **Вернон, В.** Реализация методов предметно-ориентированного проектирования / В. Вернон. – М.: ИД «Вильямс», 2016. – 688 с.
8. **Метц, С.** Ruby. Объектно-ориентированное проектирование / С. Метц. – СПб.: Питер, 2017. – 304 с.

Дата поступления
в редакцию: 06.09.2023

Дата принятия
к публикации: 01.11.2023