УДК 004

Архитектура легко тестируемых и надежных приложений

Senior Software Engineer АО "Kaspi Bank" Сычев Е. А.

Резюме

В статье обобщен накопленный за три года промышленной эксплуатации опыт применения шаблона Clean Architecture при разработке распределенных веб-сервисов на платформе .NET. Показано, как строгая инверсия зависимостей, разделение ядра (Core) и инфраструктуры, а также минимизация внешних фреймворков повышают модульность, ускоряют написание автоматических тестов и упрощают миграцию на новые версии платформы и СУБД. Приведены количественные метрики, подтверждающие рост покрытия тестами и снижение времени устранения инцидентов. Также показано влияние архитектурного подхода на скорость обучения новых разработчиков.

Summary

This paper summarizes three years of production experience applying the Clean Architecture pattern to distributed web services built with .NET. It explains how strict dependency inversion, a clear separation between Core and Infrastructure layers, and intentional avoidance of intrusive frameworks increase modularity, accelerate unit testing, and simplify upgrades to new platform versions and data stores. Quantitative metrics show an eleven-fold rise in test coverage, a seven-times faster mean time to recovery when changing storage, and a significant drop in mapping-related incidents. Organisational effects on onboarding speed are also discussed.

Түйіндеме

Мақалада .NET платформасында тара-тылған веб-қызметтерді құру кезінде Clean Architecture үлгісін қолданудың үш жылдық өндірістік тәжірибесі жинақталып берілген. Тәуелділіктерді инверсиялау, Core мен Infrastructure қабаттарын айқын бөлу және фреймворктерге шамадан тыс тәуелділіктен бас тарту жүйенің модульдігін арттырып, модульдік тесттерді жылдамдатады, сондай-ақ платформаның жаңа нұсқаларына және дерекқорларға көшу үрдісін жеңілдететіні көрсетілді. Метрикалық деректер тесттерді қамту он бір есе өскенін, сақтау көзін ауыстырғандағы кідіріс жеті есе қысқарғанын және mapping қателерінің айтарлықтай азайғанын дәлелдейді. Соңында жаңа әзірлеушілерді енгізу жылдамдығына әсері талданады.

Ключевые слова: Clean Architecture, Dependency Inversion, Unit Testing, .NET, Repository Pattern, Testability, Software Architecture, Data Abstraction, тестирование.

Введение

Современные приложения должны не только соответствовать бизнес-требованиям, но и легко адаптироваться к изменениям. В статье рассматриваются практические приемы проектирования архитектуры на основе реальных требований и трехлетнего опыта эксплуатации.

Формулировка требований

При проектировании архитектуры были сформулированы следующие основные требования:

1. Простота тестирования компонентов.
2. Легкость перехода на новые версии платформ.
3. Простота поддержки и понимания кода.
4. Возможность замены источников данных без изменения бизнес-логики.
5. Отказ от библиотек, существенно влияющих на архитектуру (например, Revo, Marten, MediatR в .Net).

Несоответствие указанным требованиям усложняет поддержку кода и снижает гибкость разработки. Целью не является строгая приверженность DDD, CQRS или Event Sourcing, но фокусируется на практическом удобстве и надежности.

Методы (архитектура и реализация)

Выбор архитектуры

На основании изложенных требований был выбран шаблон Clean Architecture («луковая» архитектура). В отличие от традиционных трехуровневых систем (UI – BLL – DAL), Clean Architecture предполагает инверсию зависимостей: инфраструктура зависит от абстракций, заданных в ядре приложения [1].

Проект разделяется на три основных слоя:

1. API – контроллеры, валидация входных данных, сериализация.
2. Core – доменные модели, сценарии использования (use cases), контракты репозиториев.
3. Infrastructure – реализация контрактов доступа к данным, маппинг.

Таблица 1. Слои приложения, обязанности и зависимости

Слой	Ответственность	Зависимости
API	Контроллеры, валидация входа, сериализация	Core, Infrastructure (только для DI)
Core	Доменные модели, сценарии использования, контракты репозиториев	—
Infrastructure	Реализация контрактов доступа к данным, маппинг	Core

Примечание: проект API имеет ссылку на Infrastructure исключительно для настройки контейнера зависимостей, при этом код бизнес-логики (Core) напрямую ничего не знает о типах из слоя Infrastructure.

Как видно из таблицы, бизнес-логика полностью сосредоточена в слое Core, который не зависит от внешних деталей. Слой Core включает в себя модель предметной области (сущности, сценарии использования) и абстракции для операций, требующих реализации на уровне инфраструктуры – например, интерфейсы репозиториев для доступа к данным. Эти интерфейсы определяют контракты взаимодействия, а конкретные реализации располагаются во внешнем слое Infrastructure. Тем самым достигается принцип инверсии зависимостей (Dependency Inversion): вместо того чтобы бизнес-логика зависела от базы данных или фреймворка доступа к данным, сами детали инфраструктуры

зависят от абстракций, заданных в Core [2, с. 63] [3, с. 494]. Это ключевое свойство чистой архитектуры, которое позволяет изолировать высокоуровневую логику от технических деталей.

Реализация

Пример контроллера:

```
[ApiController]
[Route("api/[controller]")]
public class ProductController : ControllerBase
{
    private readonly IGetProductUseCase _getProductUseCase;

    public ProductController(IGetProductUseCase getProductUseCase)
    {
        _getProductUseCase = getProductUseCase;
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<ProductResponse>> GetProduct(int id)
    {
        var productCoreResult = await _getProductUseCase.Execute(id);
        var productResponse = productCoreResult.ToResponse();

        return Ok(productResponse);
    }
}
```

Изображение 1. Пример контроллера, принимающего запрос на получение продукта и вызывающего соответствующий сценарий использования из Core

Слой Core содержит бизнес-логику, изолированную от инфраструктурных зависимостей:

```
public class GetProductUseCase : IGetProductUseCase
{
    private readonly IProductRepository _productRepository;

    public GetProductUseCase(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }

    public async Task<ProductCoreResult> Execute(int id)
    {
        var productDataResult = await _productRepository.GetById(id);
        var productCoreResult = productDataResult.ToResult();

        return productCoreResult;
    }
}
```

Изображение 2. Пример реализации сценария использования, который извлекает данные через интерфейс репозитория.

Инфраструктурный слой реализует доступ к данным и маппинг:

```
public class ProductRepository : IProductRepository
{
    public async Task<ProductDataResult> GetById(int id)
    {
        const string query = "SELECT Id, Name, Price FROM Products WHERE Id = @Id";

        using var connection = new SqlConnection("DbConnection");
        await connection.OpenAsync();

        var productSourceModel = await connection.QuerySingleOrDefaultAsync<ProductSourceModel>(query, new { Id = id });
        var productDataResult = productSourceModel.ToDataResult();

        return productDataResult;
    }
}
```

Изображение 3. Пример реализации репозитория, выполняющего SQL-запрос к базе данных и возвращающего данные в формате, ожидаемом слоем Core.

```
Api.csproj

├── Controllers
|   └── ProductController.cs

Core.csproj

├── RepositoriesContracts
|   └── IProductRepository.cs
├── UseCases
|   └── GetProduct
|     ├── GetProductUseCase.cs
|     └── IGetProductUseCase.cs

Infrastructure.csproj

├── Repositories
|   └── ProductRepository.cs
```

Изображение 4. Структура проекта.

Обсуждение

Минимизация влияния библиотек

Одной из ключевых практик, реализованных в предложенной архитектуре, является минимизация внешних зависимостей и отказ от фреймворков, навязывающих свой подход к организации кода.

В чистой архитектуре подобной проблемы удается избежать за счет строгого разделения: ORM вообще не используется на уровне Core. Если проект написан на платформе .NET, в инфраструктурном слое может применяться Entity Framework Core, Dapper или другой инструмент работы с БД – однако вся логика ORM-инструмента инкапсулируется внутри реализации репозиториев слоя Infrastructure. Слой Core ничего не знает о том, какая именно БД или ORM используется: вместо этого он оперирует методами интерфейса репозитория [4, с. 492]. Такой подход соответствует рекомендациям официальной документации Microsoft, где отмечается, что реализация доступа к данным (включая DbContext EF Core и миграции) должна располагаться во внешнем слое, а наиболее распространенный способ абстрагировать код доступа к данным – использовать шаблон репозиторий. Аналогично поступаем и с другими возможными инфраструктурными деталями: например, интеграция с внешними API, отправка сообщений в очередь, кэширование – все это реализуется в Infrastructure, предоставляя Core простой интерфейс.

Практический плюс такого решения – снижение технологической связности. Команда разработки получает свободу использовать любые инструменты в инфраструктуре, не опасаясь сломать бизнес-логику. Если на этапе старта проекта выбрана одна СУБД или ORM, а через некоторое время возникает необходимость заменить их, то благодаря слою абстракций сделать это значительно проще. Достаточно реализовать тот же интерфейс репозитория для новой системы хранения – и остальной код продолжит работать без изменений. Таким образом достигается требование независимости бизнес-логики от источников данных, заявленное в постановке задачи.

Тестируемость и простота сопровождения

Тестируемость была одним из главных приоритетов при проектировании, и Clean Architecture естественным образом обеспечивает это свойство. Поскольку слой Core изолирован от внешних зависимостей, бизнес-правила можно тестировать автономно, без поднятия базы данных, веб-сервера и прочих инфраструктурных элементов. В юнит-тестах вместо реальных репозиториев подставляются заглушки или mock-объекты, реализующие нужный интерфейс – благодаря этому тесты выполняются быстро и концентрируются на проверке логики, а не окружения. Microsoft подчеркивает, что если ядро приложения не зависит от инфраструктуры, писать автоматические модульные тесты для бизнес-логики очень просто [1]. Для интеграционных тестов также достигается выигрыш: вместо реальных компонентов инфраструктуры на время тестирования можно подключить альтернативные реализации (фейковые сервисы, in-memory базы и т.п.), что существенно упрощает тестирование сложных сценариев [5, с. 217] [6, с. 1015]. В совокупности архитектура с четким разделением ответственности устраняет одну из главных болей традиционных многослойных систем, где для тестирования бизнес-логики приходилось поднимать полноценное окружение или использовать громоздкие моки из-за тесной связи с БД.

Результаты

В течение трех лет использования предложенной архитектуры в промышленных проектах были собраны количественные и качественные метрики, подтверждающие ее эффективность. Сравнение до и после внедрения представлено в таблице ниже.

Показатель	**До внедрения**	**После внедрения (среднее за 3 года)**
Покрытие бизнес-логики модульными тестами	7 %	78 %
Среднее время адаптации к смене источника данных	4,2 дня	1 день
Время миграции между версиями .NET	1 неделя	1 день
Количество инцидентов, связанных с null/mapping	23 / квартал	3 / квартал

Интерпретация результатов:

1. Рост покрытия тестами более чем в 11 раз стал возможен благодаря изоляции бизнес-логики в Core и отказу от тесной интеграции с инфраструктурными компонентами. Это существенно повысило уверенность в изменениях и упростило рефакторинг.

2. Сокращение времени при смене источников данных подтверждает достижение независимости от конкретных реализаций хранения: для перехода на альтернативные хранилища достаточно реализовать новый интерфейс репозитория, не трогая бизнес-логику.

3. Ускорение миграции платформы объясняется минимизацией зависимости от платформенных API и высокой модульностью системы. Обновления инфраструктуры не требует переписывания логики Core.

4. Снижение количества ошибок маппинга и null-значений связано с явным разделением ответственности и возможностью централизованной обработки преобразований данных в слое Infrastructure.

Заключение

Применение Clean Architecture с четким разделением слоев помогает создавать гибкие, тестируемые и надежные системы. Практические результаты показывают значительный рост качества и сокращение операционных затрат.

Список литературы:

1. Общие архитектуры веб-приложений, Microsoft Learn. [Электронный ресурс]. URL: https://learn.microsoft.com/ru-ru/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures
2. Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2018. — 352 с.
3. Шаблоны корпоративных приложений.: Пер. с англ. — М. : ООО «И.Д. Вильямс», 2016. — 544 с.
4. Реализация методов предметно-ориентированного проектирования, Пер. с англ. - М. : ООО "И.Д. Вильяме': 2016. - 688 с.
5. Хориков Владимир. Принципы юнит-тестирования. Спб.: Питер, 2022 – 320 с.
6. ASP.NET Core в действии. Третье изд. ДМК Пресс, 2024 – 1046 с.

ӘОЖ 005.95

ҚАЗАҚСТАННЫҢ ДҮНИЕЖҮЗІЛІК САУДА ҰЙЫМЫНА КІРУІНІҢ ҚАЗІРГІ ЖАҒДАЙЫ МЕН МӘСЕЛЕЛЕРІ

Махашева Ж.А. – сеньор-лектор, экономика ғылымдарының магистрі
Маханова А.Ж. – сеньор-лектор, экономика ғылымдарының магистрі
Умирзакова Н.Т. – сеньор-лектор, экономика ғылымдарының магистрі
«Экономика және бизнес» факультеті,
«Экономика және басқару» кафедрасы
Шерхан Мұртаза атындағы Халықаралық Тараз университеті

Аннотация: Сегодня Казахстан может соответствовать требованиям Всемирной торговой организации, или насколько эффективны эти требования для нашей экономики? вопросы рассматриваются подробно. Казахстану следует учитывать действия других государств в этом отношении. Приведены текущие темпы экономического развития