ANALYSIS OF THE ARCHITECTURE DESIGN APPROACH BASED ON REQUIREMENTS

Adzinets D., Sychev Y., Alooeff E.
Belarusian State University of Informatics and Radioelectronics, Kaspi Bank JSC, ATEK
Minsk, Belarus, Almaty, Kazakhstan, Wroclaw, Poland
E-mail: adzinets@bsuir.by, sigmadel10@gmail.com, alooeff@atek.dev

We present a requirement-driven architecture for .NET production applications: three layers (API, Core, Infrastructure) with dependency inversion and low coupling. Over one year, unit-test coverage rose from 7% to 78%, data-source switch time fell from 4.2 days to 1 day, .NET migration time fell from 1 week to 1 day, and null/mapping incidents decreased from 23 per quarter to 3 per quarter. This separation simplifies testing, speeds migrations, and lets teams replace data stores without changing business logic.

Introduction

During the software lifecycle there is a design phase. It is common to use reusable, proven solution to a recurring problem at the system level, addressing concerns related to the overall structure, component interactions, and quality attributes of the system. This is also known as software architecture patterns and they typically affect system-level concerns. Modern software applications must not only meet business requirements but also be easy to adapt to changes and remain reliable.

In real production systems, applications often fall into two extremes: the lack of any architecture, which leads to "spaghetti code" and huge methods, or the other extreme — excessive use of patterns and layers without real need. This article considers the principle of selecting and designing architecture based on key requirements.

Therefore, practical principles for modern production applications are prioritized over theoretical ideas applied without understanding.

I. Methods

Based on the team's experience maintaining different projects, the following requirements for new application architecture were defined:

- Simplicity of component testing;
- Easy migration to new platform versions;
- Simplicity of maintenance and code understanding;
- Ability to replace data sources without changing business logic;
- Avoiding libraries that heavily affect architecture.

II. Justification of Architecture Choice

Testability was one of the main priorities during design. As Microsoft recommendations show, for easy unit testing, a central layer that does not depend on the infrastructure or other layers is needed [1]. Therefore, it was decided to use a Core layer for business logic and interfaces for external communication; and an Infrastructure layer for implementations.

Since the Core layer is isolated from external dependencies, business rules can be tested independently, without running a database, web server, or other infrastructure elements. For integration tests, this also brings advantages: instead of real infrastructure components, you can connect alternative implementations (fake services, in-memory databases, etc.), which makes testing complex scenarios much easier [2] [3].

This approach also solves another requirement – the ability to replace data sources without changing business logic (fig. 1).

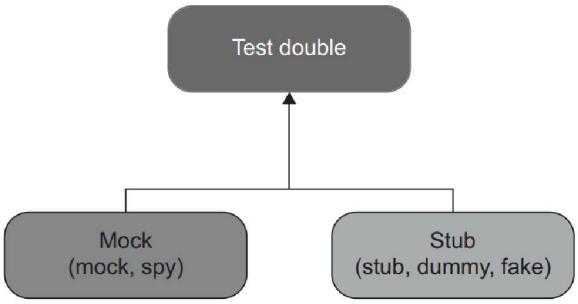

Figure 1 – All variations of test doubles can be categorized into two types: mocks and stubs.

III. Project Structure

The architecture follows the Dependency Inversion principle: the infrastructure depends on abstractions defined in the application core [1].

The project is divided into three main layers:

- API – controllers, input validation, serialization;
- Core – domain models, use cases, repository contracts;
- Infrastructure – data access implementations, mapping.

Table 1 – Application Layers, Responsibilities, and Dependencies

Layer	Responsibility	Dependencies
API	controllers, input validation, serialization	Core, Infrastructure (for DI only)
Core	domain models, use cases, repository contracts	—
Infrastructure	implementation of data access, mapping	Core

Note: the API project references Infrastructure only for dependency injection configuration; business logic (Core) knows nothing about Infrastructure types.

As the table shows, business logic is fully concentrated in the Core layer, which does not depend on external details. The Core layer contains the domain model (entities, use cases) and abstractions for operations that need implementation on the infrastructure level – for example, repository interfaces for data access. These interfaces define interaction contracts, and concrete implementations are located in the Infrastructure layer. This achieves the Dependency Inversion Principle: instead of business logic depending on the database or data framework, infrastructure details depend on abstractions defined in Core [4] [5].

IV. Results

During one year of using the proposed architecture in production projects, quantitative and qualitative metrics confirmed its effectiveness. The comparison before and after implementation is shown below.

Table 2 – Before/After Metrics (average over 1 year)

Metric	Before	After
Business logic unit test coverage	7%	78%
Average time to change data source	4.2 days	1 day
Time for .NET version migration	1 week	1 day
Number of null/mapping incidents	23 per quarter	3 per quarter

V. Interpretation of Results

The increase of test coverage by more than 11 times became possible due to the isolation of business logic in Core and the removal of tight integration with infrastructure components. This improved confidence in changes and simplified refactoring.

Reducing the time required to switch data sources confirms the independence from specific storage implementations: to migrate to another storage, it is enough to implement a new repository interface without touching business logic.

Faster platform migration is explained by minimal dependency on platform APIs and high modularity. Infrastructure updates do not require rewriting Core logic.

The decrease in mapping and null-related errors is related to clear separation of responsibilities and centralized data transformation handling in the Infrastructure layer.

VI. Conclusion

Applying a requirement-based architecture helps to create flexible, testable, and reliable systems. Practical results show a significant increase in quality and a reduction in operational costs.

1. Common web application architectures. Microsoft Learn *[Electronic resource]*. Available from: `https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures` (accessed 24 Oct 2025).
2. Martin, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson, 2017. 432 p.
3. Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. 560 p.
4. Khorikov, V. *Unit Testing Principles, Practices, and Patterns*. Manning Publications, 2020. 304 p.
5. Lock, A. *ASP.NET Core in Action*. Third Edition. Manning Publications, 2023. 984 p.