Domain-Centric Architecture with C#

Building Maintainable Applications Using Clean Architecture, SOLID Principles, and Modern Design Patterns

TailoredRead AI

Domain-Centric Architecture with C#

Building Maintainable Applications Using Clean Architecture, SOLID Principles, and Modern Design Patterns

TailoredRead AI

Title:	Domain-Centric Architecture with C#
Subtitle:	Building Maintainable Applications Using Clean Architecture, SOLID Principles, and Modern Design Patterns
Edition:	3nd Edition
Date Created:	Jan 8, 2026

Language:	English (US)
License:	Commercial License
Written By:	TailoredRead AI (TailoredRead.com) Co-editor: Kaj Bromose
Technology:	Artificial Intelligence and manual corrections
AI Model Used:	anthropic:claude-sonnet-4-5-20250929
Engine Version:	21

0 TABLE OF CONTENS

1 INTRODUCTION

Software architecture shapes how applications evolve over time. The decisions you make early in a project determine whether your codebase becomes easier to maintain or gradually transforms into an unmaintainable mess. Many developers have experienced the frustration of working with applications where simple changes require modifications across dozens of files, where business logic hides in controllers or database procedures, and where adding new features feels like navigating a minefield. These problems don't emerge from lack of effort or skill—they result from architectural choices that couple business logic tightly to technical infrastructure.

Domain-centric architecture offers a different path. By placing your business logic at the center of your application and organizing everything else around it, you create systems that reflect how your business actually works. This approach isolates your core domain from external concerns like databases, web frameworks, and third-party services. When your domain logic stands independent of these technical details, your application becomes easier to understand, test, and modify. Changes to your database technology or UI framework no longer require rewriting your business rules.

This book guides you through the principles and practices of building domain-centric applications in C#. You'll learn how Clean Architecture and Onion Architecture organize code into layers with clear dependencies flowing toward the domain. You'll discover how SOLID principles create flexible, maintainable code that adapts to changing requirements. Through practical examples and real-world scenarios, you'll see how to design aggregates that enforce business rules, implement repositories that abstract data access, and leverage dependency injection to achieve loose coupling. Whether you're starting a new project or improving an existing application, these patterns will help you build software that serves your business needs today and adapts to tomorrow's challenges.

1.1 Why Architecture Matters

Architecture determines the long-term health of your software. In the early stages of development, architectural differences might seem invisible—features get delivered regardless of how you organize your code. But as applications grow, poor architectural decisions compound. What started as a small application with a few database tables and controllers becomes a tangled web where every change risks breaking existing functionality. Teams slow down, bugs multiply, and developers spend more time fighting the codebase than delivering value.

Consider a typical scenario: your application stores customer data in SQL Server, and your business logic lives in ASP.NET controllers. This works fine initially. Then your company decides to add a mobile app that needs the same business logic. You discover that your logic is tightly coupled to HTTP concerns—it reads directly from `Request` objects and returns `ActionResult` instances. You can't reuse it. You face a choice: duplicate the logic in a new API layer, or refactor everything. Both options are expensive and risky.

Good architecture prevents these problems by establishing clear boundaries between different concerns. When your business logic exists independently of delivery mechanisms and data storage, you can:

- **Test business rules without databases or web servers** running, making tests fast and reliable
- **Change your database technology** from SQL Server to PostgreSQL without touching business logic

- **Add new interfaces** like REST APIs, GraphQL endpoints, or message queues without duplicating code
- **Understand the system** by reading domain code that expresses business concepts clearly
- **Onboard new developers** faster because the architecture guides them toward correct patterns

The cost of poor architecture grows exponentially over time. A study by the Consortium for IT Software Quality found that technical debt costs U.S. companies approximately $2.41 trillion annually[1]. Much of this debt stems from architectural decisions that seemed expedient initially but created maintenance nightmares later. Every shortcut taken, every business rule embedded in a stored procedure, every domain concept scattered across multiple layers adds to this debt.

Domain-centric architecture addresses these challenges by inverting traditional dependencies. Instead of building your business logic on top of frameworks and databases, you build frameworks and databases around your business logic. Your domain becomes the stable center, and everything else becomes a replaceable detail. This inversion requires discipline and upfront thinking, but it pays dividends throughout your application's lifetime. When your business rules change—and they will—you modify domain code without touching infrastructure. When new technologies emerge, you swap out infrastructure without touching domain code.

The benefits extend beyond technical concerns. When your code reflects business concepts clearly, domain experts can read and validate it. Your application becomes a living model of your business, not just a collection of technical artifacts. Conversations with stakeholders improve because

[1]Consortium for IT Software Quality (2022). The Cost of Poor Software Quality in the US: A 2022 Report.

you share a common language expressed in code. This alignment between business and technology reduces misunderstandings and helps ensure you're building the right thing, not just building things right.

Architecture also affects team dynamics and productivity. In poorly architected systems, developers fear making changes because they can't predict the consequences. They work around problems instead of fixing them, adding layers of complexity. In well-architected systems, developers understand where different concerns belong. They make changes confidently because the architecture guides them. Code reviews focus on business logic and design rather than untangling dependencies. Teams move faster and deliver more value.

The investment in good architecture pays off most when you need it least obviously—during maintenance and evolution. The majority of software costs occur after initial development[2]. Applications that survive long enough to provide value spend years in maintenance mode, receiving bug fixes, feature additions, and technology updates. Architecture determines whether this maintenance is straightforward or agonizing. Domain-centric architecture makes maintenance manageable by isolating changes and making dependencies explicit.

1.2 THE SHIFT TO DOMAIN-CENTRIC THINKING

Traditional application development often starts with technical concerns. You choose a framework, design a database schema, create tables, and then add business logic wherever it fits. This database-centric or framework-centric approach feels natural because it follows the tools' conventions. Entity Framework encourages you to think about entities as database rows. ASP.NET MVC encourages you to put logic in controllers.

[2]Sommerville, Ian (2015). Software Engineering, 10th Edition. Pearson.

These frameworks provide structure, but they structure your application around technical concerns rather than business concepts.

Domain-centric thinking flips this approach. You start by modeling your business domain—the concepts, rules, and processes that define what your application does. You identify entities like `Customer`, `Order`, and `Product` not as database tables but as business concepts with behaviors and rules. You discover that an `Order` isn't just a collection of data fields; it's an entity that can be placed, confirmed, shipped, and cancelled, with rules governing each transition. This understanding shapes your code before you consider databases or web frameworks.

The shift requires changing how you think about dependencies. In traditional layered architecture, your business logic depends on data access layers, which depend on databases. Your domain code references Entity Framework classes, calls repository methods that return database entities, and couples itself to persistence concerns. In domain-centric architecture, dependencies flow inward. Your domain defines interfaces for what it needs—perhaps an `IOrderRepository`—without knowing or caring about implementation details. Infrastructure layers implement these interfaces, depending on the domain rather than the domain depending on infrastructure.

This inversion of dependencies is the Dependency Rule, a core principle of domain-centric architecture. The rule states: *source code dependencies must point only inward, toward higher-level policies*. Your domain layer sits at the center, containing your business rules and policies. Application services surround the domain, orchestrating use cases. Infrastructure layers sit at the edges, implementing technical details. Dependencies flow from outer layers toward the center, never outward. This means:

- Domain code never references infrastructure code or frameworks

- Application services depend on domain interfaces but not infrastructure implementations
- Infrastructure implementations depend on domain interfaces they implement
- Presentation layers depend on application services but not directly on domain internals

Consider a concrete example. You're building an e-commerce system, and you need to process orders. In a traditional approach, your `OrderController` might directly instantiate an `OrderService`, which directly instantiates an `OrderRepository`, which directly uses Entity Framework's `DbContext`. Each layer knows about and depends on the layer below it. If you want to test the order processing logic, you need a database. If you want to change from Entity Framework to Dapper, you modify the repository, which might affect the service, which might affect the controller.

In a domain-centric approach, you start with the domain. You create an `Order` entity that understands order states and transitions. You define domain services that implement business rules like pricing calculations or inventory checks. You declare an `IOrderRepository` interface in the domain layer that specifies what operations the domain needs—perhaps `GetById`, `Save`, and `FindByCustomer`. Your domain code uses this interface without knowing about databases.

Your application layer contains use cases like `PlaceOrderUseCase` that orchestrate domain objects and call repository interfaces. Your infrastructure layer provides an `OrderRepository` class that implements `IOrderRepository` using Entity Framework. Your presentation layer calls use cases through application service interfaces. Dependencies flow inward: infrastructure depends on domain interfaces, application services depend on domain entities, and presentation depends on application services. You can test domain logic without databases, swap Entity Framework for another ORM without touching domain code, and add new presentation layers without modifying business rules.

This shift requires discipline. Frameworks and tools encourage you to take shortcuts—to put business logic in controllers because it's convenient, to let Entity Framework entities serve as domain models because it reduces mapping code, to reference infrastructure concerns from domain code because it's faster initially. Domain-centric architecture asks you to resist these shortcuts. The payoff comes later, when your application needs to evolve and you discover that changes are localized and manageable rather than cascading through every layer.

The shift also changes how you communicate about your application. Instead of discussing tables, controllers, and frameworks, you discuss domains, aggregates, and use cases. Your code uses the language of your business—`PlaceOrder`, `CalculateShippingCost`, `ApplyDiscount`—rather than generic CRUD operations. This ubiquitous language, a concept from Domain-Driven Design, bridges the gap between technical and business stakeholders. When everyone uses the same terms with the same meanings, misunderstandings decrease and collaboration improves.

Domain-centric thinking doesn't mean ignoring technical concerns. Databases, frameworks, and infrastructure matter. But they matter as implementation details that serve your domain, not as foundations that constrain it. You still use Entity Framework, ASP.NET, and other tools—you just organize them so they depend on your domain rather than your domain depending on them. This organization gives you flexibility and control, allowing you to make technical decisions based on current needs while preserving the ability to change those decisions later.

1.3 How to Use This Book

This book builds progressively from foundational concepts to practical implementation. Each chapter introduces ideas that build on previous chapters, so reading sequentially provides the clearest path. However, if you're already familiar with certain concepts, you can skip ahead to chapters that address your specific interests. The book is organized into three main sections: foundations, core patterns, and practical implementation.

The foundations section (Chapters 1-4) establishes the principles underlying domain-centric architecture. Chapter 1 examines problems with traditional layered architecture and introduces the domain-first approach and the Dependency Rule. Chapter 2 covers SOLID principles in depth, showing how each principle contributes to maintainable code. Chapters 3 and 4 explore Clean Architecture and Onion Architecture respectively, explaining their layers, comparing their approaches, and demonstrating implementation in C#. These chapters provide the conceptual framework for everything that follows.

The core patterns section (Chapters 5-9) dives into specific architectural patterns and layers. Chapter 5 focuses on designing the domain layer, covering entities, value objects, domain services, and domain events. Chapter 6 explores aggregate design patterns, teaching you how to define consistency boundaries and size aggregates correctly. Chapter 7 examines the application layer, showing how to implement use cases and separate commands from queries. Chapter 8 details the Repository Pattern, including when to use generic versus specific repositories. Chapter 9 covers infrastructure layer implementation and integration with external services.

The practical implementation section (Chapters 10-14) applies these patterns to real-world scenarios with specific technologies. Chapter 10 shows how to use Entity Framework Core within clean architecture, including DbContext design and aggregate mapping. Chapter 11 covers dependency injection and IoC containers in depth. Chapter 12 demonstrates building presentation layers with Blazor while maintaining separation of concerns. Chapter 13 addresses testing strategies for domain-centric applications. Chapter 14 provides practical guidance on organizing solution structures and migrating existing applications.

Each chapter follows a consistent structure designed to maximize learning:

- **Concept introduction** explains what the pattern or principle is and why it matters
- **Problem scenarios** illustrate the issues the pattern solves with concrete examples
- **Implementation guidance** shows how to apply the pattern in C# with code examples
- **Best practices** highlight common pitfalls and how to avoid them
- **Practical examples** demonstrate the pattern in realistic scenarios

Code examples throughout the book use modern C# features including nullable reference types, records, and pattern matching where appropriate. Examples start simple to illustrate core concepts, then grow more sophisticated to show real-world application. You'll see both what to do and what to avoid, with explanations of why certain approaches work better than others. All code examples are complete enough to understand in context but focused enough to highlight specific concepts without overwhelming detail.

The book assumes you have intermediate C# knowledge. You should be comfortable with classes, interfaces, generics, and LINQ. You don't need prior experience with Entity Framework, Blazor, or architectural patterns—the book explains these as needed. If you encounter unfamiliar C# features, the book provides brief explanations, but you may want to consult C# documentation for deeper understanding of language features.

To get the most from this book, consider setting up a practice project where you apply concepts as you learn them. Start with a simple domain—perhaps a library management system, a task tracker, or a simple e-commerce store. As you read each chapter, implement the patterns in your practice project. This hands-on approach reinforces learning and helps you encounter and solve real implementation challenges. The book's examples provide guidance, but applying concepts to your own domain deepens understanding.

You'll notice the book emphasizes principles over prescriptive rules. Domain-centric architecture provides guidelines and patterns, not rigid formulas. Your specific context—team size, project complexity, business domain, and technical constraints—influences how you apply these patterns. The book teaches you to think architecturally so you can make informed decisions rather than blindly following templates. When the book presents multiple approaches, it explains trade-offs so you can choose appropriately for your situation.

Throughout the book, you'll find references to additional resources for deeper exploration. Domain-centric architecture draws from Domain-Driven Design, SOLID principles, and various architectural patterns. While this book provides comprehensive coverage of applying these concepts in C#, the referenced works offer additional perspectives and deeper theoretical foundations. Consider these references as opportunities for continued learning rather than prerequisites.

The book's conclusion synthesizes key takeaways and provides guidance for continuing your architectural journey. Software architecture is a discipline that deepens with experience. This book gives you the knowledge and patterns to start building better applications immediately, but mastery comes from applying these concepts repeatedly, learning from mistakes, and adapting patterns to new contexts. View this book as the beginning of your architectural education, not the end.

As you progress through the book, you'll develop a new perspective on application design. You'll start seeing applications as collections of layers with clear dependencies rather than monolithic codebases. You'll recognize when business logic is leaking into infrastructure and know how to fix it. You'll design classes that have single responsibilities and depend on abstractions rather than concrete implementations. These skills will serve you throughout your career, regardless of which specific frameworks or technologies you use. The principles of domain-centric architecture transcend any particular tool or platform.

2 Chapter 1: Foundations of Domain-Centric Architecture

Software architecture shapes how applications evolve over time. The decisions you make early in a project determine whether your codebase becomes easier to maintain or gradually transforms into an unmaintainable mess. Traditional approaches to organizing code often lead to systems where business logic becomes scattered across multiple layers, tightly coupled to databases, frameworks, and external dependencies. This coupling makes testing difficult, changes risky, and understanding the system's behavior nearly impossible without tracing through infrastructure code.

Domain-centric architecture offers a fundamentally different approach. Instead of organizing your application around technical concerns like databases or web frameworks, you structure it around your business domain. The core business logic sits at the center of your application, completely isolated from external concerns. This isolation creates systems that are easier to understand, test, and modify. When requirements change—and they always do—you can adapt your business logic without worrying about breaking database queries or UI components.

This chapter establishes the foundational concepts that underpin domain-centric architecture. You'll discover why traditional layered architectures create problems that compound over time. You'll learn how the domain-first approach inverts these problems by placing business logic at the core. Most importantly, you'll understand the Dependency Rule, which governs how different parts of your application relate to each other. By the end of this chapter, you'll have a clear mental model of what makes architecture "domain-centric" and why this approach leads to more maintainable applications.

2.1 Problems with Traditional Layered Architecture

Traditional layered architecture organizes applications into horizontal layers: presentation, business logic, and data access. Each layer depends on the one below it, creating a top-down dependency flow. The presentation layer calls the business layer, which calls the data access layer, which interacts with the database. This pattern appears logical at first glance—after all, data flows from the database through your application to the user interface. However, this structure creates several critical problems that emerge as applications grow.

The most significant problem is **database-centric design**. In traditional layered architectures, the database schema often drives the entire application structure. Your business logic layer works with data transfer objects that mirror database tables. Your domain entities become simple containers for data, with little or no behavior. This approach, known as an *anemic domain model*, pushes business logic into service classes that manipulate data structures. The result is code where business rules are scattered across multiple service classes, making it difficult to understand what the system actually does.

Consider a typical e-commerce application built with traditional layering. You might have an `Order` class that's essentially a data container:

```
public class Order
{
    public int OrderId { get; set; }
    public DateTime OrderDate { get; set; }
    public decimal TotalAmount { get; set; }
    public string Status { get; set; }
    public List<OrderItem> Items { get; set; }
}
```

`OrderServiceOrder`

Tight coupling to infrastructure represents another major problem. In traditional architectures, your business logic layer directly references data access technologies like Entity Framework. Your business services might call `DbContext` methods directly, or work with repository implementations that expose infrastructure concerns. This coupling makes testing difficult—you need a database connection to test business logic. It also makes switching technologies painful. If you decide to move from SQL Server to MongoDB, or from Entity Framework to Dapper, you'll need to modify your business logic layer.

The dependency direction in traditional layered architecture creates a fundamental issue: **your business logic depends on infrastructure details**. Your domain layer references your data access layer, which means changes to database technology or ORM configuration can force changes to business logic. This violates a core principle of good design: high-level policy should not depend on low-level details. Your business rules—the core value of your application—should not care whether data is stored in SQL Server, PostgreSQL, or a file system.

Testing becomes increasingly difficult as applications grow. Unit testing business logic requires either a database connection or complex mocking of data access layers. Integration tests become slow and brittle because they depend on database state. The tight coupling between layers means you can't test business rules in isolation. You end up writing fewer tests, or tests that are so complex they become maintenance burdens themselves. This testing difficulty leads to lower code quality and more bugs in production.

Traditional layered architectures also suffer from **poor separation of concerns**. Business logic leaks into presentation layers through validation logic in controllers or view models. Data access concerns leak into business logic through lazy loading behavior or query optimization requirements. Over time, the boundaries between layers blur. What started as a clean three-layer architecture becomes a tangled mess where every layer knows too much about every other layer. Making changes requires understanding the entire stack, and modifications in one layer ripple through the entire application.

The cumulative effect of these problems is **reduced agility**. As your application grows, making changes becomes slower and riskier. Adding new features requires touching multiple layers. Refactoring becomes dangerous because dependencies flow in all directions. Technical debt accumulates because the architecture fights against change rather than enabling it. Teams spend more time managing complexity than delivering business value. This is why many applications become "legacy" systems long before they should—not because the technology is old, but because the architecture makes evolution prohibitively expensive.

2.2 The Domain-First Approach

Domain-centric architecture inverts the traditional approach by placing your business domain at the center of your application. Instead of building your application around database tables or framework capabilities, you start with your business logic. The domain layer contains entities, value objects, and domain services that model your business concepts and enforce business rules. This layer has no dependencies on external concerns—it doesn't know about databases, web frameworks, or external APIs. Everything else in your application exists to support the domain.

This inversion creates a fundamentally different structure. Your domain layer defines interfaces for the services it needs, such as repositories for data access or external service integrations. The infrastructure layer implements these interfaces, but the domain layer never references the infrastructure layer directly. Dependencies point inward toward the domain, never outward. This means your business logic can evolve independently of technical decisions about databases, frameworks, or third-party services.

In a domain-first approach, your entities are **rich domain models** that contain both data and behavior. They enforce business rules through their methods and protect their invariants through encapsulation. Consider the same e-commerce order example, but designed with a domain-first approach:

```
public class Order
{
    private readonly List<OrderItem> _items = new();
    private OrderStatus _status;

    public IReadOnlyCollection<OrderItem> Items =>
_items.AsReadOnly();
    public OrderStatus Status => _status;
    public decimal TotalAmount => _items.Sum(i => i.Price *
i.Quantity);

    public void AddItem(Product product, int quantity)
    {
        if (_status != OrderStatus.Draft)
            throw new InvalidOperationException("Cannot
modify submitted order");

        if (quantity <= 0)
            throw new ArgumentException("Quantity must be
positive");

        var existingItem = _items.FirstOrDefault(i =>
i.ProductId == product.Id);
        if (existingItem != null)
            existingItem.IncreaseQuantity(quantity);
        else
            _items.Add(new OrderItem(product, quantity));
    }

    public void Submit()
    {
        if (_items.Count == 0)
            throw new InvalidOperationException("Cannot
submit empty order");

        _status = OrderStatus.Submitted;
    }
}
```

`Order`

The domain-first approach emphasizes **ubiquitous language**—using the same terminology in code that business stakeholders use in conversation. Your classes, methods, and properties should reflect business concepts, not technical implementation details. Instead of `UpdateOrderStatusToTwo()`, you write `Submit()`. Instead of `OrderDTO`, you have an `Order` entity. This alignment between code and business language makes your codebase more understandable and reduces the translation overhead between technical and business discussions.

Domain-centric architecture treats the database as an implementation detail. Your domain entities don't inherit from base classes required by your ORM. They don't have attributes that specify database mappings. They don't expose public setters just to satisfy Entity Framework's requirements. Instead, your infrastructure layer handles the translation between your rich domain models and database representations. This separation means you can change persistence technologies without touching your business logic.

This approach requires a shift in how you think about application design. Instead of starting with database schema design, you start with domain modeling. You identify the key entities, value objects, and aggregates in your business domain. You define the operations these objects support and the invariants they must maintain. Only after you've modeled your domain do you consider how to persist it. This sequence ensures that technical concerns don't drive business logic design.

The benefits of domain-first design become apparent as applications grow. When business requirements change, you modify your domain layer. The changes are localized because business logic isn't scattered across service classes and database procedures. When you need to add a new feature, you extend your domain model with new methods or entities. The domain layer's independence from infrastructure means you can make these changes confidently, knowing you won't accidentally break database queries or API integrations.

Domain-first architecture also improves collaboration between developers and domain experts. When your code reflects business concepts directly, domain experts can review and validate your models. They can suggest improvements to how you've modeled their domain. This feedback loop is impossible when business logic is buried in SQL stored procedures or scattered across service classes. The domain layer becomes a shared language between technical and business stakeholders, improving the quality of both the software and the business understanding it embodies.

Testing becomes straightforward in domain-centric architectures. Your domain layer has no external dependencies, so you can test business logic with simple unit tests. You don't need databases, web servers, or external APIs. Tests run fast and reliably. You can practice test-driven development effectively because you're testing pure business logic, not infrastructure integration. This testing ease leads to better test coverage and higher confidence in your business rules.

2.3 UNDERSTANDING THE DEPENDENCY RULE

The Dependency Rule is the fundamental principle that makes domain-centric architecture work: **dependencies must point inward toward the**

domain, never outward. Inner layers define interfaces for services they need, and outer layers implement those interfaces. The domain layer sits at the center and has zero dependencies on outer layers. The application layer depends on the domain layer but not on infrastructure. The infrastructure layer depends on both domain and application layers, implementing the interfaces they define. This unidirectional dependency flow creates the isolation that makes domain-centric architecture powerful.

Understanding the Dependency Rule requires understanding the difference between *source code dependencies* and *runtime flow*. At runtime, your application layer might call a repository implementation in the infrastructure layer to retrieve data. However, at the source code level, the application layer only depends on the repository interface defined in the domain layer. The infrastructure layer depends on the domain layer by implementing that interface. This inversion of dependencies—where the flow of control crosses architectural boundaries in the opposite direction of source code dependencies—is achieved through dependency injection.

Consider a concrete example. Your domain layer defines a repository interface:

```
// Domain Layer
public interface IOrderRepository
{
    Task<Order> GetByIdAsync(OrderId id);
    Task SaveAsync(Order order);
}
// Application Layer
public class SubmitOrderUseCase
{
    private readonly IOrderRepository _repository;

    public SubmitOrderUseCase(IOrderRepository repository)
    {
        _repository = repository;
    }

    public async Task ExecuteAsync(OrderId orderId)
    {
        var order = await
_repository.GetByIdAsync(orderId);
        order.Submit();
        await _repository.SaveAsync(order);
    }
}
```

```
// Infrastructure Layer
public class EfOrderRepository : IOrderRepository
{
    private readonly ApplicationDbContext _context;

    public EfOrderRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<Order> GetByIdAsync(OrderId id)
    {
        // Entity Framework implementation details
    }

    public async Task SaveAsync(Order order)
    {
        // Entity Framework implementation details
    }
}
```

The Dependency Rule creates several important constraints that guide your design decisions:

- **Inner layers cannot reference outer layers:** Your domain layer cannot have using statements that import infrastructure namespaces. If you try to reference Entity Framework from your domain layer, you're violating the Dependency Rule.
- **Inner layers define abstractions:** When the domain needs a service, it defines an interface. It doesn't call concrete implementations directly. This keeps the domain independent and testable.

- **Outer layers implement abstractions:** Infrastructure and presentation layers implement the interfaces defined by inner layers. They adapt external technologies to match the interfaces your domain expects.
- **Dependencies are injected:** Concrete implementations are provided to inner layers through dependency injection, not through direct instantiation. This enables the inversion of dependencies.

Violating the Dependency Rule has immediate consequences. If your domain layer references Entity Framework directly, you've coupled your business logic to a specific ORM. Testing requires database connections. Switching ORMs requires modifying business logic. The benefits of domain-centric architecture evaporate. Maintaining the Dependency Rule requires discipline, but the payoff is substantial: your business logic remains independent, testable, and adaptable.

The Dependency Rule applies to all types of dependencies, not just repositories. If your domain needs to send emails, it defines an `IEmailService` interface. If it needs to generate PDFs, it defines an `IDocumentGenerator` interface. If it needs to call external APIs, it defines interfaces for those services. The infrastructure layer implements these interfaces using actual email libraries, PDF generators, or HTTP clients. Your domain remains blissfully unaware of these implementation details.

One common question is: *where do interfaces live?* In domain-centric architecture, interfaces that the domain needs are defined in the domain layer itself, even though they're implemented in the infrastructure layer. This placement reinforces that the domain defines its requirements—it's not adapting to what infrastructure provides. The domain says "I need a repository with these methods," and infrastructure says "I'll provide that." This is the essence of the Dependency Inversion Principle, one of the SOLID principles you'll explore in the next chapter.

Enforcing the Dependency Rule in C# projects is straightforward. Organize your solution into separate projects for each layer. Your domain project references no other projects. Your application project references only the domain project. Your infrastructure project references domain and application projects. Your presentation project references application and infrastructure projects (for dependency injection configuration). If you try to add a reference that violates the Dependency Rule, the compiler prevents it. This structural enforcement makes it nearly impossible to accidentally violate the rule.

The Dependency Rule enables the primary benefit of domain-centric architecture: **your business logic becomes a stable core that rarely changes for technical reasons**. When you upgrade Entity Framework, your domain layer is unaffected. When you switch from REST APIs to gRPC, your domain layer doesn't change. When you migrate from SQL Server to PostgreSQL, your business rules remain identical. Technical decisions become reversible because they're isolated in outer layers. This reversibility reduces risk and increases agility—you can experiment with new technologies without betting your entire application on them.

2.4 BENEFITS AND TRADE-OFFS

Domain-centric architecture delivers significant benefits, but it also introduces complexity and requires more upfront design effort. Understanding both the advantages and costs helps you make informed decisions about when and how to apply these patterns. For some applications, the benefits far outweigh the costs. For others, simpler architectures might be more appropriate. The key is matching architectural complexity to application complexity and business requirements.

The most significant benefit is **maintainability over time**. Applications built with domain-centric architecture remain understandable and modifiable as they grow. Business logic is centralized in the domain layer, making it easy to find and modify. The Dependency Rule prevents the coupling that typically makes large applications difficult to change. When requirements evolve—and they always do—you can adapt your domain model without cascading changes through infrastructure and presentation layers. This maintainability translates directly to reduced development costs and faster feature delivery over the application's lifetime.

Testability improves dramatically with domain-centric architecture. Your domain layer has no external dependencies, enabling fast, reliable unit tests. You can test business logic thoroughly without databases, web servers, or external services. Test-driven development becomes practical because you're testing pure business logic. This testing ease leads to better test coverage, which leads to fewer bugs and more confidence when refactoring. Teams that adopt domain-centric architecture often report test coverage increasing from 30-40% to 70-80% or higher.

Domain-centric architecture provides **technology independence**. Your business logic doesn't depend on specific frameworks, databases, or external services. This independence gives you flexibility to adopt new technologies as they emerge. You can migrate from Entity Framework to Dapper, or from SQL Server to MongoDB, without rewriting business logic. You can experiment with new frameworks in outer layers while keeping your domain stable. This flexibility reduces technical risk and prevents vendor lock-in.

The architecture enables **parallel development** by different team members or teams. One developer can work on domain logic while another implements infrastructure. Frontend developers can work against application layer interfaces while backend developers implement those interfaces. The clear boundaries between layers reduce merge conflicts and enable teams to work independently. This parallelization becomes increasingly valuable as teams grow.

However, domain-centric architecture introduces several trade-offs you must consider:

- **Increased initial complexity:** You need more projects, more interfaces, and more abstraction layers. Simple CRUD operations require more code than in traditional architectures. This overhead is only justified if your application has sufficient business logic complexity.
- **Steeper learning curve:** Developers must understand the Dependency Rule, dependency injection, and domain modeling concepts. Teams new to these patterns need time to become productive. Training and mentoring become important investments.
- **More code to write:** Interfaces, DTOs, mapping logic, and adapter classes all add lines of code. For simple applications, this additional code might not provide sufficient value. The architecture shines when business logic complexity justifies the structural overhead.
- **Potential over-engineering:** It's easy to create unnecessary abstractions or overly complex domain models. Not every application needs aggregates, domain events, and elaborate entity hierarchies. Applying domain-centric patterns to simple CRUD applications can make them harder to understand, not easier.

The question of *when to use domain-centric architecture* depends on several factors. Applications with complex business logic benefit most—think financial systems, healthcare applications, or e-commerce platforms with sophisticated rules. Applications that will evolve over many years justify the upfront investment. Projects where business logic changes frequently benefit from the isolation domain-centric architecture provides. Conversely, simple CRUD applications, short-lived prototypes, or applications with minimal business logic might not warrant the additional complexity.

Performance is sometimes raised as a concern with domain-centric architecture. The additional abstraction layers and object mapping can introduce overhead. However, in practice, this overhead is rarely significant compared to database queries, network calls, or business logic execution. The architecture doesn't prevent optimization—you can still implement caching, optimize queries, or use performance-oriented data access patterns. The difference is that these optimizations are isolated in infrastructure layers, not scattered throughout your codebase.

Team size and experience level affect the cost-benefit calculation. Small teams or junior developers might struggle with the additional complexity. The patterns require understanding dependency injection, interface design, and domain modeling—concepts that take time to master. However, once teams become proficient, they often find domain-centric architecture makes them more productive, not less. The clear structure and separation of concerns reduce the cognitive load of understanding large codebases.

The long-term benefits of domain-centric architecture compound over time. In the first few months of a project, traditional layered architecture might seem faster. You write less code and make fewer design decisions. However, as the application grows and requirements change, the benefits of domain-centric architecture become apparent. Changes that would require extensive refactoring in traditional architectures become localized modifications in domain-centric designs. Technical debt accumulates more slowly because the architecture prevents the coupling that creates debt.

Consider domain-centric architecture as an investment in your application's future. Like any investment, it has upfront costs—more planning, more code, more learning. But it pays dividends over time through reduced maintenance costs, faster feature delivery, and greater adaptability. The key is ensuring your application's complexity and longevity justify the investment. For the right applications, domain-centric architecture is one of the best investments you can make in code quality and long-term maintainability.

3 CHAPTER 2: SOLID PRINCIPLES IN PRACTICE

The SOLID principles form the bedrock of maintainable, flexible software design. These five principles, introduced by Robert C. Martin, provide guidance for writing code that adapts gracefully to changing requirements without requiring extensive rewrites. When you apply SOLID principles consistently, your codebase becomes easier to understand, test, and extend. Each principle addresses a specific aspect of software design, from how you organize responsibilities to how you manage dependencies between components.

Understanding SOLID principles is essential before diving into domain-centric architectures like Clean Architecture or Onion Architecture. These architectural patterns rely heavily on SOLID principles to maintain proper separation of concerns and dependency management. Without a solid grasp of these fundamentals, you'll struggle to understand why certain architectural decisions are made and how to implement them effectively in your own projects. The principles work together synergistically, each reinforcing the others to create cohesive, well-structured code.

In this chapter, you'll explore each SOLID principle through practical C# examples that demonstrate both violations and proper implementations. You'll see how these principles apply specifically to domain-centric architecture, where maintaining clean boundaries between layers is paramount. By the end of this chapter, you'll have concrete techniques for recognizing when your code violates these principles and strategies for refactoring toward better designs. These aren't abstract theoretical concepts—they're practical tools you'll use daily when building enterprise applications.

3.1 SINGLE RESPONSIBILITY PRINCIPLE

The Single Responsibility Principle (SRP) states that a class should have only one reason to change. This means each class should have a single, well-defined responsibility within your system. When a class handles multiple concerns, changes to one concern can inadvertently break functionality related to another concern. This coupling makes your code fragile and difficult to maintain. The principle applies not just to classes but also to methods, modules, and even entire services within your architecture.

Consider a common violation of SRP: a `Customer` class that handles both customer data and email notifications. This class has two reasons to change—when customer data structure changes or when email notification logic changes. Here's what this violation looks like:

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }

    public void Save()
    {
        // Database logic here
        var connection = new SqlConnection("...");
        // Save customer to database
    }

    public void SendWelcomeEmail()
    {
        var smtpClient = new
SmtpClient("smtp.example.com");
        var message = new MailMessage();
        message.To.Add(Email);
        message.Subject = "Welcome!";
        message.Body = $"Welcome {Name}!";
        smtpClient.Send(message);
    }
}
```

To fix this violation, you separate concerns into distinct classes, each with a single responsibility. The `Customer` class should focus solely on representing customer domain logic. Data access moves to a repository, and email functionality moves to a dedicated service:

```
public class Customer
{
    public int Id { get; private set; }
    public string Name { get; private set; }
    public string Email { get; private set; }

    public Customer(string name, string email)
    {
        if (string.IsNullOrWhiteSpace(name))
            throw new ArgumentException("Name is
required");
        if (string.IsNullOrWhiteSpace(email))
            throw new ArgumentException("Email is
required");

        Name = name;
        Email = email;
    }

    public void UpdateEmail(string newEmail)
    {
        if (string.IsNullOrWhiteSpace(newEmail))
            throw new ArgumentException("Email is
required");
        Email = newEmail;
    }
}

public interface ICustomerRepository
{
    void Save(Customer customer);
    Customer GetById(int id);
}

public interface IEmailService
{
    void SendWelcomeEmail(Customer customer);
}
```

In domain-centric architecture, SRP helps you maintain clean boundaries between layers. Your domain entities focus on business rules and invariants. Repositories handle data access. Application services orchestrate use cases. When each component has a single responsibility, you can modify infrastructure concerns like switching databases without touching your domain logic. This separation is fundamental to achieving the flexibility that domain-centric architectures promise.

Recognizing SRP violations requires asking yourself: How many reasons does this class have to change? If you can identify multiple unrelated reasons, you've found a violation. Common signs include classes with names containing and or Manager, methods that do multiple unrelated things, or classes that import namespaces from multiple layers of your architecture. When you spot these patterns, refactor by extracting responsibilities into separate classes with focused purposes.

The benefits of following SRP extend beyond maintainability. Classes with single responsibilities are inherently more testable because you can test each concern in isolation. They're also more reusable—a focused email service can be used across multiple contexts, while a monolithic class mixing concerns cannot. Your code becomes self-documenting when class names accurately reflect their single purpose. Team members can quickly understand what each component does without wading through unrelated functionality.

One common misconception is that SRP means a class should only have one method or one property. That's not accurate. A class can have multiple methods and properties as long as they all serve the same cohesive purpose. For example, a `Customer` entity might have methods for updating contact information, validating business rules, and calculating customer lifetime value—all of these relate to the single responsibility of representing customer domain logic. The key is that all members of the class should change for the same reason.

When applying SRP in practice, start with your domain layer. Ensure your entities focus purely on business logic and invariants. Move persistence concerns to repositories, validation logic to validators or domain services, and cross-cutting concerns like logging to infrastructure services. This disciplined separation creates a foundation for the other SOLID principles to build upon. As you progress through this book, you'll see how SRP enables the clean layering that makes domain-centric architectures so powerful and maintainable.

3.2 Open-Closed Principle

The Open-Closed Principle (OCP) states that software entities should be open for extension but closed for modification. This means you should be able to add new functionality without changing existing code. When you follow OCP, you reduce the risk of introducing bugs into working code while still allowing your system to evolve. This principle is particularly important in domain-centric architectures where business rules frequently change and new requirements emerge regularly. The key mechanism for achieving OCP is abstraction through interfaces and base classes.

Consider a payment processing system that violates OCP. The code uses conditional logic to handle different payment methods, requiring modification every time you add a new payment type:

```
public class PaymentProcessor
{
    public void ProcessPayment(Order order, string
paymentType)
    {
        if (paymentType == "CreditCard")
        {
            // Process credit card payment
            Console.WriteLine("Processing credit card
payment");
        }
        else if (paymentType == "PayPal")
        {
            // Process PayPal payment
            Console.WriteLine("Processing PayPal payment");
        }
        else if (paymentType == "BankTransfer")
        {
            // Process bank transfer
            Console.WriteLine("Processing bank transfer");
        }
    }
}
```

To adhere to OCP, you introduce an abstraction that allows extension without modification. Define an interface for payment processing and create separate implementations for each payment method:

```
public interface IPaymentMethod
{
    void ProcessPayment(Order order);
    bool CanProcess(PaymentDetails details);
}

public class CreditCardPayment : IPaymentMethod
{
    public void ProcessPayment(Order order)
    {
        // Credit card specific logic
        Console.WriteLine("Processing credit card
payment");
    }

    public bool CanProcess(PaymentDetails details)
    {
        return details.Type == PaymentType.CreditCard;
    }
}

public class PayPalPayment : IPaymentMethod
{
    public void ProcessPayment(Order order)
    {
        // PayPal specific logic
        Console.WriteLine("Processing PayPal payment");
    }

    public bool CanProcess(PaymentDetails details)
    {
        return details.Type == PaymentType.PayPal;
    }
}

public class PaymentProcessor
{
    private readonly IEnumerable<IPaymentMethod>
_paymentMethods;

    public PaymentProcessor(IEnumerable<IPaymentMethod>
paymentMethods)
    {
        _paymentMethods = paymentMethods;
    }

    public void ProcessPayment(Order order, PaymentDetails
details)
    {
        var method = _paymentMethods.FirstOrDefault(m =>
m.CanProcess(details));
        if (method == null)
            throw new InvalidOperationException("No payment
method available");

        method.ProcessPayment(order);
    }
}
```

The Strategy pattern, demonstrated above, is one of several design patterns that support OCP. Other patterns include Template Method, where you define an algorithm's structure in a base class and let subclasses override specific steps, and Decorator, where you add functionality by wrapping objects rather than modifying them. In domain-centric architecture, you'll frequently use these patterns to handle varying business rules across different contexts or customer segments without cluttering your core domain logic with conditional statements.

OCP becomes particularly powerful when combined with dependency injection. By injecting collections of interface implementations, you create systems that automatically discover and use new functionality without configuration changes. For example, your payment processor automatically recognizes new payment methods registered in your dependency injection container. This approach is common in plugin architectures and systems that need to support customer-specific customizations without forking the codebase.

A practical challenge with OCP is knowing when to apply it. You shouldn't prematurely abstract every piece of code—that leads to unnecessary complexity. Instead, apply OCP when you identify variation points in your system where requirements are likely to change or expand. Payment methods, notification channels, pricing strategies, and validation rules are common variation points in business applications. Use the rule of three: when you find yourself modifying the same code for the third time to handle a new case, that's a signal to refactor toward OCP.

In domain-centric architectures, OCP helps you keep your domain layer stable while allowing infrastructure and application layers to evolve. Your domain entities and value objects should be closed for modification—their core business rules shouldn't change frequently. However, you can extend behavior through domain services, specifications, or policy objects that implement domain interfaces. This keeps your domain model clean and focused while providing flexibility where you need it. The infrastructure layer, by contrast, is naturally more open to extension as you integrate new external services and technologies.

Testing becomes significantly easier when you follow OCP. Each payment method implementation can be tested independently without complex setup or mocking. You can test the payment processor with a simple stub implementation that verifies the correct method is selected. When you add a new payment method, you only need to write tests for that specific implementation—existing tests remain valid and unchanged. This stability in your test suite gives you confidence that new features haven't broken existing functionality, a critical benefit as your application grows in complexity.

3.3 Liskov Substitution Principle

The Liskov Substitution Principle (LSP) states that objects of a derived class should be substitutable for objects of the base class without affecting program correctness. Named after Barbara Liskov, this principle ensures that inheritance hierarchies are logically sound and that polymorphism works as expected. When you violate LSP, you create surprising behavior where code that works with a base class breaks when given a derived class. This principle is crucial for maintaining the contracts that interfaces and base classes establish with their consumers.

A classic LSP violation involves the square-rectangle problem. Consider a `Rectangle` class with a `Square` subclass. Mathematically, a square is a rectangle, but in object-oriented design, this relationship often violates LSP:

```
public class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : Rectangle
{
    public override int Width
    {
        get => base.Width;
        set
        {
            base.Width = value;
            base.Height = value; // Maintain square
constraint
        }
    }

    public override int Height
    {
        get => base.Height;
        set
        {
            base.Width = value; // Maintain square
constraint
```

```
            base.Height = value;
        }
    }
}
```

To demonstrate the violation, consider this test that works for rectangles but fails for squares:

```
public void TestRectangleArea(Rectangle rectangle)
{
    rectangle.Width = 5;
    rectangle.Height = 10;
    Assert.Equal(50, rectangle.CalculateArea()); // Fails
for Square!
}
```

`SquareRectangle`

A better design recognizes that rectangles and squares have different behavioral contracts. You might create a common interface for shapes without implying one is a subtype of the other:

```
public interface IShape
{
    int CalculateArea();
}

public class Rectangle : IShape
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : IShape
{
    public int SideLength { get; set; }

    public int CalculateArea()
    {
        return SideLength * SideLength;
    }
}
```

LSP violations often manifest as precondition strengthening or postcondition weakening in derived classes. A derived class that requires more restrictive inputs than its base class strengthens preconditions, violating LSP. Similarly, a derived class that provides weaker guarantees about its outputs weakens postconditions. For example, if a base class method promises never to return null but a derived class sometimes returns null, that's an LSP violation. Your code should honor the contracts established by base classes and interfaces without introducing surprising exceptions or limitations.

In domain-centric architecture, LSP is particularly important when designing entity hierarchies and domain services. Consider an order processing system with different order types. If `SubscriptionOrder` inherits from `Order`, it must support all operations that `Order` supports. If certain operations don't make sense for subscriptions—like one-time discounts—you've likely modeled the hierarchy incorrectly. Instead, consider using composition with shared interfaces or creating separate entity types that implement common contracts without inheritance.

A practical technique for ensuring LSP compliance is to write tests against base class or interface types, then run those same tests against all derived implementations. If any derived class fails tests that pass for the base class, you've found an LSP violation. This approach, sometimes called contract testing, helps you catch substitutability problems early. Your test suite becomes a specification of the behavioral contract that all implementations must honor, making LSP violations immediately visible.

When you encounter LSP violations in existing code, refactoring options include extracting common behavior into a shared interface without inheritance, using composition instead of inheritance, or splitting the hierarchy into separate types. Sometimes the violation indicates a misunderstanding of the domain—what seemed like an is-a relationship is actually a has-a or behaves-like-a relationship. Consulting with domain experts can help clarify these relationships and guide you toward designs that accurately model business concepts while maintaining proper substitutability.

The benefits of following LSP extend throughout your architecture. Polymorphism becomes reliable and predictable. You can confidently write code against abstractions knowing that any implementation will work correctly. Your domain model accurately reflects business concepts without forcing artificial inheritance relationships. Testing becomes more straightforward because you can test through interfaces without worrying about implementation-specific quirks. When combined with the other SOLID principles, LSP helps create architectures where components truly are interchangeable, giving you the flexibility to evolve your system as requirements change.

3.4 Interface Segregation Principle

The Interface Segregation Principle (ISP) states that clients should not be forced to depend on interfaces they don't use. Large, monolithic interfaces create unnecessary coupling and force implementing classes to provide implementations for methods they don't need. When you follow ISP, you create focused, cohesive interfaces that represent specific capabilities or roles. This makes your code more flexible, easier to test, and simpler to understand. ISP is closely related to SRP but applies specifically to interface design rather than class design.

Consider a violation of ISP in a document management system. A single `IDocument` interface tries to cover all possible document operations:

```
public interface IDocument
{
    void Open();
    void Close();
    void Save();
    void Print();
    void Fax();
    void Email();
    void Encrypt();
    void Decrypt();
    void ConvertToPdf();
    void Scan();
}
```

`OpenClose`

To fix this violation, you segregate the interface into smaller, focused interfaces that represent specific capabilities:

```
public interface IReadableDocument
{
    void Open();
    void Close();
}

public interface IWritableDocument
{
    void Save();
}

public interface IPrintableDocument
{
    void Print();
}

public interface IEmailableDocument
{
    void Email();
}

public interface IEncryptableDocument
{
    void Encrypt();
    void Decrypt();
}

public class ReadOnlyDocumentViewer : IReadableDocument
{
    public void Open()
    {
        // Implementation
    }

    public void Close()
    {
        // Implementation
    }
}

public class FullFeaturedDocument : IReadableDocument,
IWritableDocument,
    IPrintableDocument, IEmailableDocument
{
    // Implements all relevant interfaces
}
```

ISP becomes particularly important in domain-centric architectures when designing repository interfaces. A common mistake is creating a single `IRepository<T>` interface with methods for all possible operations—Add, Update, Delete, GetById, GetAll, Find, etc. Not all entities need all operations. Some entities might be read-only in certain contexts, while others might not support deletion due to business rules. Instead of one large interface, consider segregating based on actual needs:

```
public interface IReadRepository<T>
{
    T GetById(int id);
    IEnumerable<T> GetAll();
}

public interface IWriteRepository<T>
{
    void Add(T entity);
    void Update(T entity);
}

public interface IQueryableRepository<T>
{
    IEnumerable<T> Find(Expression<Func<T, bool>>
predicate);
}

public interface ICustomerRepository :
IReadRepository<Customer>,
    IWriteRepository<Customer>,
IQueryableRepository<Customer>
{
    // Customer-specific query methods
    IEnumerable<Customer> GetActiveCustomers();
}
```

The benefits of ISP extend to testing and mocking. When interfaces are small and focused, creating test doubles becomes trivial. You only need to implement the specific methods your test requires, rather than stubbing out an entire large interface. This reduces test setup code and makes tests more readable. For example, testing a component that only reads customers requires implementing just `IReadRepository<Customer>`, not a full repository interface with write operations you'll never call.

A practical guideline for applying ISP is the role interface pattern. Instead of designing interfaces around data structures or technical layers, design them around the roles that clients play. A reporting component plays the role of a data reader. A command handler plays the role of a data writer. An audit system plays the role of an event subscriber. By thinking in terms of roles, you naturally create focused interfaces that match how clients actually use your code. This alignment between interface design and usage patterns is the essence of ISP.

ISP also helps you manage dependencies between layers in domain-centric architectures. Your domain layer might define small, focused interfaces for capabilities it needs from infrastructure—like `IEmailSender` or `IFileStorage`. The infrastructure layer implements these interfaces without the domain layer depending on large, infrastructure-specific interfaces. This keeps your domain layer clean and prevents it from knowing about implementation details it shouldn't care about. The dependency inversion principle works hand-in-hand with ISP to maintain proper layer separation.

When refactoring toward ISP, start by analyzing how clients actually use your interfaces. If you find that different clients use completely different subsets of methods, that's a strong signal to segregate the interface. Look for implementing classes that throw `NotImplementedException` or `NotSupportedException`—these are clear ISP violations. Extract focused interfaces for each group of related methods, then have your original interface inherit from these smaller interfaces if you need to maintain backward compatibility. Over time, migrate clients to depend on the specific interfaces they actually need.

The combination of ISP with the other SOLID principles creates remarkably flexible architectures. Small, focused interfaces (ISP) with single responsibilities (SRP) that can be extended without modification (OCP) and properly substituted (LSP) through dependency inversion (DIP) give you systems that adapt gracefully to change. In domain-centric architecture, this flexibility is essential because business requirements evolve constantly. ISP ensures that changes to one capability don't ripple through unrelated parts of your system, maintaining the isolation and independence that make domain-centric architectures so maintainable.

3.5 Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules—both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions. This principle is fundamental to domain-centric architectures because it enables the dependency rule: dependencies flow inward toward the domain, never outward. By inverting dependencies through abstractions, you create systems where your business logic remains independent of infrastructure concerns like databases, frameworks, and external services.

Consider a violation of DIP where a high-level order processing service directly depends on low-level infrastructure:

```
public class OrderService
{
    private readonly SqlServerOrderRepository _repository;
    private readonly SmtpEmailService _emailService;

    public OrderService()
    {
        _repository = new SqlServerOrderRepository();
        _emailService = new SmtpEmailService();
    }

    public void PlaceOrder(Order order)
    {
        _repository.Save(order);
        _emailService.SendOrderConfirmation(order);
    }
}
```

`OrderService`

To apply DIP, you introduce abstractions that both high-level and low-level modules depend on. The high-level module defines the interfaces it needs, and low-level modules implement those interfaces:

```
// Abstractions defined in the domain or application layer
public interface IOrderRepository
{
    void Save(Order order);
    Order GetById(int id);
}

public interface IEmailService
{
    void SendOrderConfirmation(Order order);
}

// High-level module depends on abstractions
public class OrderService
{
    private readonly IOrderRepository _repository;
    private readonly IEmailService _emailService;

    public OrderService(IOrderRepository repository,
IEmailService emailService)
    {
        _repository = repository;
        _emailService = emailService;
    }

    public void PlaceOrder(Order order)
    {
        _repository.Save(order);
        _emailService.SendOrderConfirmation(order);
    }
}
```

```
// Low-level modules implement abstractions (in
infrastructure layer)
public class SqlServerOrderRepository : IOrderRepository
{
    public void Save(Order order)
    {
        // SQL Server specific implementation
    }

    public Order GetById(int id)
    {
        // SQL Server specific implementation
        return null;
    }
}

public class SmtpEmailService : IEmailService
{
    public void SendOrderConfirmation(Order order)
    {
        // SMTP specific implementation
    }
}
```

`OrderService`

The key insight of DIP is that the direction of dependency is inverted from traditional layered architectures. In traditional architectures, the business logic layer depends on the data access layer. With DIP, the business logic layer defines interfaces for what it needs, and the data access layer implements those interfaces. This means the data access layer depends on the business logic layer, not the other way around. This inversion is what allows your domain layer to remain at the center of your architecture, independent of infrastructure concerns.

DIP enables the testability that domain-centric architectures are known for. When your domain and application layers depend only on abstractions, you can easily substitute test doubles during testing. Here's how you might test the order service:

```
public class OrderServiceTests
{
    [Fact]
    public void PlaceOrder_SavesOrderAndSendsEmail()
    {
        // Arrange
        var mockRepository = new Mock<IOrderRepository>();
        var mockEmailService = new Mock<IEmailService>();
        var service = new
OrderService(mockRepository.Object,
mockEmailService.Object);
        var order = new Order { Id = 1, CustomerId = 100 };

        // Act
        service.PlaceOrder(order);

        // Assert
        mockRepository.Verify(r => r.Save(order),
Times.Once);
        mockEmailService.Verify(e =>
e.SendOrderConfirmation(order), Times.Once);
    }
}
```

In domain-centric architectures, DIP manifests in how you organize your solution structure. Your domain layer sits at the center and defines no dependencies on other layers. Your application layer depends on the domain layer and defines interfaces for infrastructure services it needs. Your infrastructure layer depends on both domain and application layers, implementing the interfaces they define. This creates a dependency graph where all arrows point inward toward the domain, ensuring your business logic remains pure and independent.

A common question is where to define interfaces when applying DIP. The answer is: define interfaces in the layer that uses them, not the layer that implements them. If your application layer needs to send emails, define `IEmailService` in the application layer, even though the implementation lives in the infrastructure layer. This keeps your application layer from depending on infrastructure. The interface represents what the application layer needs, not how the infrastructure layer works. This is sometimes called the interface ownership principle.

DIP works synergistically with dependency injection (DI) containers. While DIP is the principle of depending on abstractions, DI is the mechanism for providing concrete implementations at runtime. Your composition root—typically in your application's startup code—configures the DI container to map interfaces to implementations:

```
public void ConfigureServices(IServiceCollection services)
{
    // Register domain and application services
    services.AddScoped<OrderService>();

    // Register infrastructure implementations
    services.AddScoped<IOrderRepository,
SqlServerOrderRepository>();
    services.AddScoped<IEmailService, SmtpEmailService>();
}
```

The benefits of DIP extend beyond testability and flexibility. Your code becomes more maintainable because changes to infrastructure don't ripple through your business logic. You can swap databases, change email providers, or integrate new external services by creating new implementations of existing interfaces. Your domain logic remains stable and unchanged. This stability is crucial in enterprise applications where business rules are complex and critical, while infrastructure technologies change frequently. DIP protects your most valuable code—your domain logic—from the volatility of technical decisions.

When applying DIP in practice, be mindful of creating abstractions that are too thin or too specific to a single implementation. An interface that simply mirrors the public API of a concrete class provides little value. Instead, design interfaces around the needs of your domain and application layers, using domain language rather than technical language. For example, `ICustomerRepository` with methods like `GetActiveCustomers()` is better than `IDataAccess` with generic CRUD methods. The abstraction should represent a meaningful concept in your domain, not just a technical mechanism for decoupling.

4 CHAPTER 3: CLEAN ARCHITECTURE EXPLAINED

Clean Architecture represents a systematic approach to organizing software systems that prioritizes business logic independence and testability. Introduced by Robert C. Martin, this architectural pattern builds upon decades of software design wisdom, incorporating principles from hexagonal architecture, onion architecture, and other domain-centric approaches. The fundamental goal is to create systems where business rules remain isolated from external concerns like databases, user interfaces, and third-party frameworks. This isolation ensures that your core business logic can evolve independently of technological choices and infrastructure decisions.

The power of Clean Architecture lies in its ability to defer critical decisions about frameworks and tools until you have sufficient information to make informed choices. By keeping your domain logic free from dependencies on specific technologies, you gain the flexibility to swap out databases, change UI frameworks, or adopt new libraries without rewriting your business rules. This approach reduces technical debt and makes your applications more resilient to the rapid pace of technological change. When your business logic doesn't depend on Entity Framework, ASP.NET Core, or any specific framework, you can upgrade or replace these components without touching your core domain.

Understanding Clean Architecture requires shifting your perspective from technology-first thinking to domain-first thinking. Traditional architectures often start with database schema design or framework selection, forcing business logic to conform to technical constraints. Clean Architecture inverts this relationship, placing business rules at the center and treating technical concerns as implementation details. This chapter explores the layers that comprise Clean Architecture, examines how entities and use cases form the system's core, and provides practical guidance for implementing these patterns in C# applications. You'll learn to structure your code so that dependencies flow inward, protecting your domain from external volatility.

4.1 The Four Layers of Clean Architecture

Clean Architecture organizes code into four concentric layers, each with distinct responsibilities and dependency rules. The innermost layer contains **Entities**, which encapsulate enterprise-wide business rules and represent the most stable part of your system. These entities change only when fundamental business requirements change, not when technical decisions shift. The next layer outward contains **Use Cases**, which implement application-specific business rules and orchestrate the flow of data to and from entities. Use cases define what your application does without specifying how it does it from a technical perspective.

The third layer comprises **Interface Adapters**, which convert data between the format most convenient for use cases and entities and the format required by external agencies like databases and web frameworks. This layer contains controllers, presenters, gateways, and repository implementations. Controllers receive HTTP requests and transform them into use case inputs, while presenters format use case outputs for display. Repository implementations in this layer translate between domain entities and database-specific data structures. The outermost layer consists of **Frameworks and Drivers**, containing tools like databases, web frameworks, and UI components. This layer is where all the technical details live, isolated from your business logic.

The critical rule governing these layers is the **Dependency Rule**: source code dependencies must point only inward, toward higher-level policies. Code in the inner layers cannot know anything about code in the outer layers. This means your entities cannot reference use cases, and your use cases cannot reference controllers or repositories. When an outer layer needs to communicate with an inner layer, it does so through interfaces defined in the inner layer. This inversion of dependencies ensures that business logic remains independent of technical implementation details.

In a C# implementation, these layers typically map to separate projects within your solution:

- **Domain** project containing entities, value objects, and domain services
- **Application** project containing use cases, interfaces, and application services
- **Infrastructure** project containing repository implementations, database contexts, and external service integrations
- **Presentation** project containing controllers, views, and UI components

This physical separation enforces architectural boundaries at compile time. When your Domain project references no other projects, the compiler prevents you from accidentally introducing dependencies on frameworks or infrastructure. The Application project references only the Domain project, ensuring use cases depend solely on business entities.

Infrastructure and Presentation projects reference Application and Domain, implementing the interfaces defined in inner layers. This structure makes violations of the Dependency Rule immediately visible during development.

Consider a simple example of how data flows through these layers. When a user submits an order through your web application, the request enters through a controller in the Presentation layer. The controller extracts relevant data and invokes a use case in the Application layer, passing a simple data transfer object. The use case retrieves necessary entities through repository interfaces, applies business rules, and updates the entities. The repository implementation in the Infrastructure layer persists changes to the database. Throughout this flow, dependencies point inward—the controller depends on the use case interface, the use case depends on entity classes and repository interfaces, and the repository implementation depends on both the repository interface and the entity classes.

The layered structure also facilitates testing at different levels of granularity. You can test entities in complete isolation, verifying business rules without any infrastructure. Use case tests can mock repository interfaces, focusing on application logic without database dependencies. Integration tests can verify that repository implementations correctly persist and retrieve entities. This testing strategy provides confidence that each layer functions correctly while maintaining fast test execution times. Unit tests for entities and use cases run in milliseconds because they involve no I/O operations.

Understanding these four layers and their relationships forms the foundation for implementing Clean Architecture effectively. Each layer serves a specific purpose, and respecting the boundaries between them creates systems that are easier to understand, test, and modify. As you design your C# applications, constantly ask yourself which layer a particular piece of code belongs to and whether your dependencies flow in the correct direction. This discipline pays dividends as your application grows and evolves over time.

4.2 Entities and Use Cases

Entities represent the core business objects that embody critical business rules and data. In Clean Architecture, an entity is not merely a data container but an object that encapsulates both state and behavior relevant to your business domain. An `Order` entity, for example, doesn't just hold order data—it enforces rules about valid order states, calculates totals, and ensures that business invariants are maintained. Entities should be designed to be framework-agnostic, containing no references to databases, UI frameworks, or external libraries. This independence allows them to be reused across different applications and tested without any infrastructure dependencies.

When designing entities in C#, focus on expressing business concepts clearly and enforcing business rules through methods rather than exposing raw properties. Consider this example of an `Order` entity:

```
public class Order
{
    private readonly List<OrderLine> _lines = new();

    public Guid Id { get; private set; }
    public DateTime OrderDate { get; private set; }
    public OrderStatus Status { get; private set; }
    public IReadOnlyCollection<OrderLine> Lines =>
_lines.AsReadOnly();

    public void AddLine(Product product, int quantity)
    {
        if (Status != OrderStatus.Draft)
            throw new InvalidOperationException("Cannot
modify confirmed order");

        if (quantity <= 0)
            throw new ArgumentException("Quantity must be
positive");

        var existingLine = _lines.FirstOrDefault(l =>
l.ProductId == product.Id);
        if (existingLine != null)
            existingLine.IncreaseQuantity(quantity);
        else
            _lines.Add(new OrderLine(product, quantity));
    }

    public decimal CalculateTotal()
    {
        return _lines.Sum(line => line.Subtotal);
    }

    public void Confirm()
    {
        if (Status != OrderStatus.Draft)
            throw new InvalidOperationException("Only draft
orders can be confirmed");

        if (!_lines.Any())
            throw new InvalidOperationException("Cannot
confirm empty order");

        Status = OrderStatus.Confirmed;
    }
}
```

This entity design demonstrates several important principles. Properties have private setters, preventing external code from arbitrarily changing the entity's state. Business rules are enforced through methods like `AddLine` and `Confirm`, which validate preconditions before modifying state. The entity exposes behavior rather than just data, making it impossible to create invalid orders. Collections are exposed as read-only, preventing external code from bypassing business rules by directly manipulating the collection. This approach creates entities that are always in a valid state, eliminating entire categories of bugs.

Use cases, also called interactors or application services, orchestrate the flow of data between entities and external systems. A use case represents a single action a user can perform in your system, such as placing an order, registering a customer, or generating a report. Use cases contain application-specific business rules—logic that defines how your particular application uses the domain entities. While entities contain rules that would apply across any application in your business domain, use cases contain rules specific to your application's requirements.

A well-designed use case follows a consistent structure:

- Receive input data through a simple request object or parameters
- Validate input according to application rules
- Retrieve necessary entities through repository interfaces
- Execute business logic by calling entity methods
- Persist changes through repository interfaces
- Return results through a response object or simple data structure

Here's an example of a use case for placing an order:

```
public class PlaceOrderUseCase
{
    private readonly IOrderRepository _orderRepository;
    private readonly IProductRepository _productRepository;
    private readonly IUnitOfWork _unitOfWork;

    public PlaceOrderUseCase(
        IOrderRepository orderRepository,
        IProductRepository productRepository,
        IUnitOfWork unitOfWork)
    {
        _orderRepository = orderRepository;
        _productRepository = productRepository;
        _unitOfWork = unitOfWork;
    }

    public async Task<PlaceOrderResponse>
ExecuteAsync(PlaceOrderRequest request)
    {
        var order = new Order(request.CustomerId);

        foreach (var item in request.Items)
        {
            var product = await
_productRepository.GetByIdAsync(item.ProductId);
            if (product == null)
                throw new
ProductNotFoundException(item.ProductId);

            order.AddLine(product, item.Quantity);
        }
        order.Confirm();
        await _orderRepository.AddAsync(order);
        await _unitOfWork.CommitAsync();

        return new PlaceOrderResponse
        {
            OrderId = order.Id,
            Total = order.CalculateTotal(),
            OrderDate = order.OrderDate
        };
    }
}
```

This use case demonstrates proper separation of concerns. It depends on repository interfaces defined in the Application layer, not concrete implementations. It orchestrates the interaction between multiple entities and repositories without containing complex business logic itself—that logic lives in the entities. The use case handles application-level concerns like transaction management through the Unit of Work pattern, while business rules about valid orders remain in the `Order` entity. Input and output are handled through simple data transfer objects that contain no behavior.

The relationship between entities and use cases creates a clear division of responsibilities. Entities answer questions like "What are the rules for a valid order?" and "How do we calculate order totals?" Use cases answer questions like "What steps are required to place an order?" and "Which entities need to be retrieved and updated?" This separation makes both entities and use cases easier to understand, test, and modify. When business rules change, you modify entities. When application workflows change, you modify use cases. Each type of change is localized to the appropriate layer.

Testing entities and use cases separately provides comprehensive coverage with minimal complexity. Entity tests verify business rules in isolation, requiring no mocking or infrastructure. Use case tests mock repository interfaces, focusing on workflow orchestration without database dependencies. This testing approach is fast, reliable, and provides clear feedback when something breaks. When a test fails, you immediately know whether the problem lies in business logic or application workflow, accelerating debugging and reducing the time required to fix issues.

4.3 Interface Adapters and Frameworks

The Interface Adapters layer serves as a translation boundary between your application's core logic and the external world. This layer converts data from the format most convenient for entities and use cases into formats required by external systems like databases, web services, and user interfaces. Controllers, presenters, view models, and repository implementations all reside in this layer. The key responsibility is adaptation—transforming data structures and coordinating between different representations without containing business logic. This layer makes it possible for your use cases to remain ignorant of HTTP, JSON, SQL, or any other technical protocol.

Controllers in the Interface Adapters layer receive requests from the web framework, extract relevant data, and invoke use cases. A controller should be thin, containing minimal logic beyond request validation and response formatting. Consider this example of an ASP.NET Core controller:

```
public class OrdersController : ControllerBase
{
    private readonly PlaceOrderUseCase _placeOrderUseCase;

    public OrdersController(PlaceOrderUseCase
placeOrderUseCase)
    {
        _placeOrderUseCase = placeOrderUseCase;
    }

    [HttpPost]
    public async Task<IActionResult> PlaceOrder([FromBody]
PlaceOrderDto dto)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        var request = new PlaceOrderRequest
        {
            CustomerId = dto.CustomerId,
            Items = dto.Items.Select(i => new
OrderItemRequest
            {
                ProductId = i.ProductId,
                Quantity = i.Quantity
            }).ToList()
        };

        try
        {
            var response = await
_placeOrderUseCase.ExecuteAsync(request);
            return Ok(new { orderId = response.OrderId,
total = response.Total });
        }
        catch (ProductNotFoundException ex)
        {
            return NotFound(new { error = ex.Message });
        }
        catch (InvalidOperationException ex)
        {
            return BadRequest(new { error = ex.Message });
        }
    }
}
```

This controller demonstrates proper adapter design. It depends on the use case, not on repositories or entities directly. It translates between the HTTP-specific `PlaceOrderDto` and the application-layer `PlaceOrderRequest`. It handles HTTP-specific concerns like status codes and error responses without leaking these concerns into the use case. The use case remains testable without any web framework dependencies, while the controller remains simple enough that it requires minimal testing. This separation allows you to change web frameworks or add additional presentation layers without modifying your application logic.

Repository implementations also belong in the Interface Adapters layer, specifically in the Infrastructure project. These implementations translate between domain entities and database-specific representations. When using Entity Framework Core, your repository implementation might look like this:

```
public class OrderRepository : IOrderRepository
{
    private readonly ApplicationDbContext _context;

    public OrderRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<Order> GetByIdAsync(Guid id)
    {
        return await _context.Orders
            .Include(o => o.Lines)
            .ThenInclude(l => l.Product)
            .FirstOrDefaultAsync(o => o.Id == id);
    }

    public async Task AddAsync(Order order)
    {
        await _context.Orders.AddAsync(order);
    }

    public async Task UpdateAsync(Order order)
    {
        _context.Orders.Update(order);
    }
}
```

The repository implementation handles Entity Framework-specific concerns like eager loading and change tracking without exposing these details to use cases. The `IOrderRepository` interface is defined in the Application layer, allowing use cases to depend on the abstraction rather than the implementation. This design enables you to swap Entity Framework for a different ORM or even a completely different persistence mechanism without changing your use cases. During testing, you can provide in-memory implementations or mocks, eliminating database dependencies from your unit tests.

The Frameworks and Drivers layer represents the outermost circle of Clean Architecture, containing tools and frameworks that your application uses but doesn't depend on conceptually. This layer includes:

- Database systems like SQL Server, PostgreSQL, or MongoDB
- Web frameworks like ASP.NET Core or Blazor
- External APIs and third-party services
- UI frameworks and component libraries
- Logging frameworks and monitoring tools

Code in this layer consists primarily of configuration and glue code that connects your application to external systems. Your Entity Framework `DbContext` configuration, for example, lives in this layer. Startup configuration that registers services with the dependency injection container also belongs here. The goal is to keep this layer as thin as possible, with most logic residing in inner layers. When you need to upgrade a framework version or switch to a different technology, changes should be confined to this outermost layer.

The relationship between Interface Adapters and Frameworks creates a protective barrier around your application core. External systems communicate with your application through adapters that translate between external formats and internal representations. Your use cases never directly interact with HTTP requests, database queries, or external APIs. This isolation provides several benefits: your core logic remains testable without infrastructure, you can defer technology decisions until you have sufficient information, and you can evolve your technical stack without rewriting business logic. These advantages compound over time as your application grows and requirements change.

Understanding the role of adapters and frameworks helps you make better architectural decisions. When adding a new feature, ask yourself where each piece of code belongs. Does it contain business rules? It belongs in an entity. Does it orchestrate a workflow? It belongs in a use case. Does it translate between formats? It belongs in an adapter. Is it framework-specific configuration? It belongs in the frameworks layer. This mental model guides you toward clean, maintainable designs that respect architectural boundaries and keep your codebase organized as complexity increases.

4.4 Implementing Clean Architecture in C#

Implementing Clean Architecture in C# requires careful project organization and dependency management. A typical solution structure consists of four projects that map directly to the architectural layers. The **Domain** project contains entities, value objects, domain services, and domain events—everything that represents your core business logic. This project has no dependencies on other projects or external frameworks, ensuring that business rules remain isolated. The **Application** project contains use cases, interfaces for repositories and external services, and application-specific logic. It references only the Domain project, maintaining the dependency rule.

The **Infrastructure** project implements the interfaces defined in the Application layer, containing repository implementations, database contexts, external service integrations, and framework-specific code. This project references both Domain and Application projects, as it needs to work with entities and implement application interfaces. The **Presentation** project contains controllers, views, view models, and UI components. For a web application, this might be an ASP.NET Core project or a Blazor application. The Presentation project references

Application and Domain, allowing controllers to invoke use cases and work with domain types for input and output.

Here's how you might structure your solution in Visual Studio:

```
YourApplication.sln
├── src/
│   ├── YourApplication.Domain/
│   │   ├── Entities/
│   │   ├── ValueObjects/
│   │   ├── DomainServices/
│   │   └── DomainEvents/
│   ├── YourApplication.Application/
│   │   ├── UseCases/
│   │   ├── Interfaces/
│   │   ├── DTOs/
│   │   └── Exceptions/
│   ├── YourApplication.Infrastructure/
│   │   ├── Persistence/
│   │   ├── Repositories/
│   │   ├── ExternalServices/
│   │   └── Configuration/
│   └── YourApplication.Web/
│       ├── Controllers/
│       ├── ViewModels/
│       └── Program.cs
└── tests/
    ├── YourApplication.Domain.Tests/
    ├── YourApplication.Application.Tests/
    └── YourApplication.Infrastructure.Tests/
```

This structure enforces architectural boundaries at compile time. If you attempt to reference Infrastructure from Application, the compiler prevents it. This physical separation makes architectural violations immediately visible and prevents accidental coupling between layers. The test projects mirror the source structure, making it easy to locate tests for specific components. Domain and Application tests contain fast-running unit tests with no infrastructure dependencies, while Infrastructure tests contain integration tests that verify database interactions and external service integrations.

Dependency injection configuration serves as the composition root where you wire together interfaces and implementations. In an ASP.NET Core application, this configuration typically occurs in `Program.cs` or a separate configuration class:

```
public static class DependencyInjection
{
    public static IServiceCollection AddApplication(this
IServiceCollection services)
    {
        services.AddScoped<PlaceOrderUseCase>();
        services.AddScoped<CancelOrderUseCase>();
        services.AddScoped<GetOrderDetailsUseCase>();
        return services;
    }

    public static IServiceCollection AddInfrastructure(
        this IServiceCollection services,
        IConfiguration configuration)
    {
        services.AddDbContext<ApplicationDbContext>(options
=>

options.UseSqlServer(configuration.GetConnectionString("Def
aultConnection")));

        services.AddScoped<IOrderRepository,
OrderRepository>();
        services.AddScoped<IProductRepository,
ProductRepository>();
        services.AddScoped<IUnitOfWork, UnitOfWork>();

        return services;
    }
}
```

This configuration approach keeps dependency registration organized by layer. The `AddApplication` method registers use cases, while `AddInfrastructure` registers repository implementations and database contexts. Extension methods make the registration code reusable and testable. During testing, you can provide alternative implementations or mocks by calling different registration methods. This flexibility enables you to test different layers in isolation while maintaining the same dependency injection patterns used in production.

When implementing use cases, follow a consistent pattern that makes your code predictable and maintainable. Each use case should be a separate class with a single public method, typically named `Execute` or `ExecuteAsync`. Use request and response objects to define inputs and outputs, avoiding primitive obsession and making the use case's contract explicit. Here's a template for a typical use case:

```
public class CreateCustomerUseCase
{
    private readonly ICustomerRepository
_customerRepository;
    private readonly IUnitOfWork _unitOfWork;
    private readonly IEmailService _emailService;

    public CreateCustomerUseCase(
        ICustomerRepository customerRepository,
        IUnitOfWork unitOfWork,
        IEmailService emailService)
    {
        _customerRepository = customerRepository;
        _unitOfWork = unitOfWork;
        _emailService = emailService;
    }

    public async Task<CreateCustomerResponse>
ExecuteAsync(CreateCustomerRequest request)
    {
        // Validate request
        if (await
_customerRepository.ExistsByEmailAsync(request.Email))
            throw new
DuplicateEmailException(request.Email);

        // Create entity
        var customer = new Customer(request.Name,
request.Email);

        // Persist changes
        await _customerRepository.AddAsync(customer);
        await _unitOfWork.CommitAsync();

        // Perform side effects
        await
_emailService.SendWelcomeEmailAsync(customer.Email);

        // Return response
        return new CreateCustomerResponse
        {
            CustomerId = customer.Id,
            Name = customer.Name,
            Email = customer.Email
        };
    }
}
```

This pattern provides several advantages. Dependencies are explicit through constructor injection, making testing straightforward. The method signature clearly communicates what data is required and what results are returned. The implementation follows a logical flow: validate, execute business logic, persist changes, handle side effects, and return results. This consistency makes your codebase easier to navigate and understand, especially as the number of use cases grows. New team members can quickly learn the pattern and apply it to new features.

Error handling in Clean Architecture should use exceptions for exceptional conditions and return values for expected outcomes. Domain exceptions, defined in the Domain project, represent violations of business rules. Application exceptions, defined in the Application project, represent application-level errors like resource not found or unauthorized access. Infrastructure exceptions wrap technical failures like database connection errors. Controllers in the Presentation layer catch these exceptions and translate them into appropriate HTTP responses. This layered approach to error handling keeps error types organized and makes it clear where each type of error originates.

As you implement Clean Architecture in your C# applications, focus on maintaining the dependency rule and keeping each layer focused on its specific responsibilities. Resist the temptation to take shortcuts that violate architectural boundaries, even when they seem convenient in the short term. The discipline of respecting these boundaries pays dividends as your application grows, making it easier to add features, fix bugs, and adapt to changing requirements. Clean Architecture is not about perfection but about creating a sustainable structure that supports long-term maintainability and evolution.

5 Chapter 4: Onion Architecture Deep Dive

Onion Architecture represents a powerful approach to structuring applications that places your domain model at the absolute center of your system. Introduced by Jeffrey Palermo in 2008, this architectural pattern visualizes your application as concentric circles, with each layer depending only on layers closer to the center[3]. Unlike traditional layered architectures where dependencies can flow in multiple directions, Onion Architecture enforces a strict rule: outer layers can depend on inner layers, but inner layers never depend on outer ones. This creates a natural protection around your business logic, shielding it from the volatile concerns of infrastructure, frameworks, and user interfaces.

The visual metaphor of an onion proves remarkably effective for understanding this architecture. Just as an onion has layers that wrap around a core, your application has layers that surround your domain

[3]Palermo, Jeffrey (2008). The Onion Architecture: part 1. Programming with Palermo blog.

model. The innermost layer contains your domain entities and value objects—the pure business logic that defines what your application does. Moving outward, you encounter domain services, then application services, and finally the infrastructure and presentation layers at the periphery. Each layer serves a specific purpose and maintains clear boundaries with its neighbors. This organization ensures that changes to external concerns like databases or UI frameworks don't ripple inward to corrupt your business logic.

What makes Onion Architecture particularly appealing for C# developers is its natural alignment with modern development practices. The architecture encourages dependency injection, promotes testability, and supports the SOLID principles you've already learned. When you build applications using this pattern, you create systems that are easier to understand, modify, and extend over time. Your domain logic becomes portable—you can swap out Entity Framework for another ORM, replace a REST API with gRPC, or migrate from MVC to Blazor without rewriting your core business rules. This flexibility translates directly into reduced maintenance costs and faster feature delivery as your application evolves.

5.1 Understanding Onion Layers

The Onion Architecture organizes your application into four primary layers, each with distinct responsibilities and dependency rules. At the center sits the **Domain Model layer**, containing entities, value objects, and domain events. This layer has zero dependencies on any other layer or external framework. Surrounding it is the **Domain Services layer**, which contains domain logic that doesn't naturally fit within a single entity. The third layer, **Application Services**, orchestrates use cases and coordinates between the domain and outer layers. Finally, the outermost layer encompasses **Infrastructure and Presentation**, handling concerns like data persistence, external APIs, and user interfaces.

The dependency flow in Onion Architecture is unidirectional and always points inward. The infrastructure layer depends on application services, which depend on domain services, which depend on the domain model. This creates a protective barrier around your business logic. Consider a practical example: your domain entity `Order` knows nothing about Entity Framework, SQL Server, or how it gets persisted. The infrastructure layer knows about `Order` and implements the persistence mechanism, but `Order` remains blissfully ignorant of these details. This separation means you can test your domain logic without spinning up a database or mocking infrastructure concerns.

Each layer communicates with inner layers through well-defined interfaces. The application layer defines repository interfaces that the infrastructure layer implements. The presentation layer calls application service interfaces without knowing implementation details. This interface-based communication enables the dependency inversion principle and makes your application highly testable. Here's how these interfaces typically look in C#:

```
// Defined in Application Layer
public interface IOrderRepository
{
    Task<Order> GetByIdAsync(Guid orderId);
    Task<IEnumerable<Order>> GetOrdersByCustomerAsync(Guid
customerId);
    Task AddAsync(Order order);
    Task UpdateAsync(Order order);
}

// Implemented in Infrastructure Layer
public class OrderRepository : IOrderRepository
{
    private readonly ApplicationDbContext _context;

    public OrderRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<Order> GetByIdAsync(Guid orderId)
    {
        return await _context.Orders
            .Include(o => o.OrderItems)
            .FirstOrDefaultAsync(o => o.Id == orderId);
    }

    // Other implementations...
}
```

The boundaries between layers aren't just conceptual—they manifest as separate projects in your Visual Studio solution. A typical Onion

Architecture solution contains projects like `YourApp.Domain`, `YourApp.Application`, `YourApp.Infrastructure`, and `YourApp.Web`. The project references enforce the dependency rule at compile time. Your domain project references nothing except perhaps some utility libraries. Your application project references only the domain project. Infrastructure and presentation projects reference application and domain, but never the reverse. This physical separation prevents accidental violations of the architectural boundaries.

Layer responsibilities must remain clear and focused to maintain architectural integrity. The domain layer contains only business logic—calculations, validations, and business rules. It never performs I/O operations, calls external services, or handles presentation concerns. The application layer orchestrates workflows but doesn't contain business rules. It retrieves entities from repositories, invokes domain methods, and coordinates transactions. The infrastructure layer handles all technical concerns: database access, file systems, email services, and external APIs. The presentation layer manages user interaction, input validation, and response formatting. When each layer respects these boundaries, your application remains maintainable and testable.

Understanding layer coupling is crucial for successful implementation. **Tight coupling** occurs when layers directly reference concrete implementations from other layers, creating brittle dependencies. **Loose coupling** through interfaces allows layers to interact without knowing implementation details. Consider this anti-pattern where an application service directly instantiates a repository:

```
// BAD: Tight coupling to infrastructure
public class OrderService
{
    public async Task<OrderDto> GetOrder(Guid orderId)
    {
        var repository = new OrderRepository(new
ApplicationDbContext());
        var order = await repository.GetByIdAsync(orderId);
        return MapToDto(order);
    }
}
```

The power of layered isolation becomes apparent when requirements change. Imagine your application currently uses SQL Server, but you need to support MongoDB for certain customers. With proper layering, you create a new `MongoOrderRepository` that implements `IOrderRepository`. Your domain and application layers require zero changes—they continue working with the interface. You configure dependency injection to provide the appropriate implementation based on configuration. This flexibility extends to all infrastructure concerns: swap logging frameworks, change caching strategies, or replace email providers without touching business logic. The layers protect your core domain from the chaos of changing technical requirements.

Testing benefits dramatically from this layered approach. Unit tests for domain entities require no mocking—they're pure logic with no external dependencies. Application service tests mock repository interfaces, allowing you to verify orchestration logic without database access. Integration tests can use in-memory implementations or test databases without modifying production code. This testing pyramid becomes natural and efficient:

- **Domain layer tests:** Fast, numerous, no dependencies—test business rules exhaustively
- **Application layer tests:** Mock repositories and external services—verify use case orchestration
- **Infrastructure tests:** Test actual database operations, API integrations—fewer but more comprehensive
- **Presentation tests:** Test UI components and controllers—verify user interaction flows

5.2 The Domain Core

The domain core represents the heart of your application—the innermost circle of the onion where your business logic lives. This layer contains entities, value objects, domain events, and the business rules that define your application's purpose. Everything in this layer should be expressible in terms your business stakeholders understand, without technical jargon or infrastructure concerns. When you open a file in the domain core, you should see code that reads like a description of business processes, not database operations or API calls. This clarity makes the domain core the most stable part of your application, changing only when business requirements change, not when you upgrade frameworks or switch databases.

Entities form the backbone of your domain core, representing concepts with unique identity that persist over time. An `Order`, `Customer`, or `Product` entity maintains identity even as its attributes change. Each entity encapsulates both data and behavior, ensuring business rules are enforced at the point where data changes. Consider this example of a well-designed domain entity:

```
public class Order
{
    public Guid Id { get; private set; }
    public Guid CustomerId { get; private set; }
    public OrderStatus Status { get; private set; }
    private readonly List<OrderItem> _items = new();
    public IReadOnlyCollection<OrderItem> Items =>
_items.AsReadOnly();
    public DateTime CreatedAt { get; private set; }
    public DateTime? ShippedAt { get; private set; }

    private Order() { } // For EF Core

    public Order(Guid customerId)
    {
        Id = Guid.NewGuid();
        CustomerId = customerId;
        Status = OrderStatus.Draft;
        CreatedAt = DateTime.UtcNow;
    }

    public void AddItem(Guid productId, int quantity,
decimal unitPrice)
    {
        if (Status != OrderStatus.Draft)
            throw new InvalidOperationException("Cannot
modify a submitted order");

        if (quantity <= 0)
            throw new ArgumentException("Quantity must be
positive", nameof(quantity));
```

```
        if (unitPrice < 0)
            throw new ArgumentException("Price cannot be 
negative", nameof(unitPrice));

        var existingItem = _items.FirstOrDefault(i => 
i.ProductId == productId);
        if (existingItem != null)
        {
            existingItem.IncreaseQuantity(quantity);
        }
        else
        {
            _items.Add(new OrderItem(productId, quantity, 
unitPrice));
        }
    }

    public void Submit()
    {
        if (Status != OrderStatus.Draft)
            throw new InvalidOperationException("Order has 
already been submitted");

        if (!_items.Any())
            throw new InvalidOperationException("Cannot 
submit an empty order");

        Status = OrderStatus.Submitted;
    }
    public decimal CalculateTotal()
    {
        return _items.Sum(item => item.Quantity *
item.UnitPrice);
    }
}
```

Value objects complement entities by representing concepts without unique identity. Two value objects with identical attributes are considered equal, unlike entities where identity matters. Money, addresses, date ranges, and email addresses make excellent value objects. They're immutable—once created, their values never change. If you need different values, you create a new instance. This immutability eliminates entire categories of bugs and makes your code easier to reason about. Here's a practical value object implementation:

```
public class Money : IEquatable<Money>
{
    public decimal Amount { get; }
    public string Currency { get; }

    public Money(decimal amount, string currency)
    {
        if (string.IsNullOrWhiteSpace(currency))
            throw new ArgumentException("Currency is
required", nameof(currency));

        Amount = amount;
        Currency = currency.ToUpperInvariant();
    }

    public Money Add(Money other)
    {
        if (Currency != other.Currency)
            throw new InvalidOperationException(
                $"Cannot add {other.Currency} to
{Currency}");

        return new Money(Amount + other.Amount, Currency);
    }

    public bool Equals(Money other)
    {
        if (other is null) return false;
        return Amount == other.Amount && Currency ==
other.Currency;
    }
    public override bool Equals(object obj) => Equals(obj
as Money);

    public override int GetHashCode() =>
HashCode.Combine(Amount, Currency);

    public static bool operator ==(Money left, Money right)
=>
        Equals(left, right);

    public static bool operator !=(Money left, Money right)
=>
        !Equals(left, right);
}
```

Domain events capture significant occurrences within your domain that business stakeholders care about. When an order is submitted, a payment is processed, or inventory falls below threshold, these events represent facts that have happened. Other parts of your system can react to these events without coupling to the entities that raised them. Domain events enable loose coupling between aggregates and support eventual consistency patterns. They're typically implemented as simple immutable classes:

```
public class OrderSubmittedEvent
{
    public Guid OrderId { get; }
    public Guid CustomerId { get; }
    public decimal TotalAmount { get; }
    public DateTime SubmittedAt { get; }

    public OrderSubmittedEvent(Guid orderId, Guid
customerId,
        decimal totalAmount, DateTime submittedAt)
    {
        OrderId = orderId;
        CustomerId = customerId;
        TotalAmount = totalAmount;
        SubmittedAt = submittedAt;
    }
}
```

The domain core must remain framework-agnostic to maintain its purity and testability. This means no references to Entity Framework, ASP.NET Core, or any other infrastructure framework. Your entities shouldn't inherit from framework base classes or use framework attributes for configuration. When you need Entity Framework to work with your entities, you configure mappings in the infrastructure layer using the Fluent API, keeping your domain clean. This separation might feel awkward initially, but it pays enormous dividends in testability and flexibility. Your domain tests run in milliseconds because they have no framework overhead.

Business rule enforcement belongs exclusively in the domain core. Every invariant—a condition that must always be true—should be protected by the entity or value object that owns it. Never allow entities to enter invalid states, even temporarily. Use private setters, validate in constructors, and provide methods that enforce rules when changing state. This approach eliminates the need for validation logic scattered throughout your application. Consider these principles:

- **Make invalid states unrepresentable:** Use types and encapsulation to prevent invalid data
- **Validate at boundaries:** Constructor and method parameters should be validated immediately
- **Fail fast:** Throw exceptions when business rules are violated—don't return error codes
- **Use domain exceptions:** Create specific exception types that express business rule violations
- **Encapsulate collections:** Expose read-only collections and provide methods to modify them safely

The domain core should be rich with behavior, not just data containers. An anemic domain model—entities with only properties and no behavior—pushes business logic into services, defeating the purpose of domain-centric architecture. Your entities should answer questions and perform actions. An `Order` should know how to calculate its total, determine if it can be cancelled, and apply discounts. A `Customer` should know if they're eligible for premium features based on their purchase history. This behavior makes your domain model a living representation of your business, not just a database schema mapped to classes.

Organizing your domain core requires thoughtful structure as your application grows. Start with a flat structure when you have few entities, but introduce bounded contexts and aggregates as complexity increases. Group related entities together, separate different business capabilities, and use namespaces to reflect your domain's structure. A typical organization might look like:

```
YourApp.Domain/
├── Orders/
│   ├── Order.cs
│   ├── OrderItem.cs
│   ├── OrderStatus.cs
│   └── Events/
│       └── OrderSubmittedEvent.cs
├── Customers/
│   ├── Customer.cs
│   ├── CustomerType.cs
│   └── ValueObjects/
│       └── EmailAddress.cs
├── Products/
│   ├── Product.cs
│   ├── ProductCategory.cs
│   └── ValueObjects/
│       └── Sku.cs
└── Common/
    └── ValueObjects/
        ├── Money.cs
        └── Address.cs
```

5.3 Application Services Layer

The application services layer orchestrates your domain logic to fulfill specific use cases. While the domain core contains business rules, the application layer coordinates how those rules are applied to accomplish user goals. This layer sits between your domain and the outside world, translating external requests into domain operations and domain results back into formats the outside world understands. Application services are the conductors of your domain orchestra—they don't play instruments themselves, but they coordinate when and how each section performs to create a cohesive result.

Application services implement use cases as discrete operations that represent what users want to accomplish. Each service method typically corresponds to a single use case: submit an order, register a customer, process a refund, or generate a report. These methods follow a consistent pattern: receive input, retrieve necessary domain objects from repositories, invoke domain methods to perform business logic, persist changes, and return results. Here's a typical application service implementation:

```
public class OrderApplicationService
{
    private readonly IOrderRepository _orderRepository;
    private readonly IProductRepository _productRepository;
    private readonly IUnitOfWork _unitOfWork;
    private readonly IDomainEventDispatcher
_eventDispatcher;

    public OrderApplicationService(
        IOrderRepository orderRepository,
        IProductRepository productRepository,
        IUnitOfWork unitOfWork,
        IDomainEventDispatcher eventDispatcher)
    {
        _orderRepository = orderRepository;
        _productRepository = productRepository;
        _unitOfWork = unitOfWork;
        _eventDispatcher = eventDispatcher;
    }

    public async Task<OrderDto>
SubmitOrderAsync(SubmitOrderCommand command)
    {
        // Retrieve domain objects
        var order = await
_orderRepository.GetByIdAsync(command.OrderId);
        if (order == null)
            throw new
OrderNotFoundException(command.OrderId);

        // Validate products exist and are available
        foreach (var item in command.Items)
        {
            var product = await
_productRepository.GetByIdAsync(item.ProductId);
            if (product == null)
                throw new
ProductNotFoundException(item.ProductId);

            if (!product.IsAvailable)
                throw new
ProductUnavailableException(item.ProductId);

            order.AddItem(item.ProductId, item.Quantity,
product.Price);
        }

        // Execute domain logic
        order.Submit();

        // Persist changes
        await _orderRepository.UpdateAsync(order);
        await _unitOfWork.CommitAsync();

        // Dispatch domain events
        await _eventDispatcher.DispatchAsync(
            new OrderSubmittedEvent(order.Id,
order.CustomerId,
                order.CalculateTotal(), DateTime.UtcNow));

        // Return result
        return MapToDto(order);
```

```
    }
}
```

Transaction management belongs in the application services layer because use cases define transactional boundaries. A single use case should succeed or fail atomically—either all changes are persisted or none are. The Unit of Work pattern helps manage this by tracking changes and committing them together. Your application service begins a transaction, performs domain operations, and commits if everything succeeds or rolls back if exceptions occur. This ensures data consistency without polluting your domain with transaction management code. The domain remains focused on business rules while the application layer handles technical concerns like transactions.

Data Transfer Objects (DTOs) serve as the contract between your application layer and the outside world. They represent the data structure that external layers send to and receive from application services. DTOs are simple, serializable classes with no behavior—just properties. They protect your domain from external coupling and allow you to shape data differently for different consumers. Your API might need one DTO shape, your Blazor components another, and your background jobs yet another. The application layer maps between domain entities and DTOs:

```
public class OrderDto
{
    public Guid Id { get; set; }
    public Guid CustomerId { get; set; }
    public string Status { get; set; }
    public List<OrderItemDto> Items { get; set; }
    public decimal TotalAmount { get; set; }
    public DateTime CreatedAt { get; set; }
}

public class OrderItemDto
{
    public Guid ProductId { get; set; }
    public string ProductName { get; set; }
    public int Quantity { get; set; }
    public decimal UnitPrice { get; set; }
    public decimal Subtotal { get; set; }
}

private OrderDto MapToDto(Order order)
{
    return new OrderDto
    {
        Id = order.Id,
        CustomerId = order.CustomerId,
        Status = order.Status.ToString(),
        Items = order.Items.Select(i => new OrderItemDto
        {
            ProductId = i.ProductId,
            ProductName = i.ProductName,
            Quantity = i.Quantity,
            UnitPrice = i.UnitPrice,
            Subtotal = i.Quantity * i.UnitPrice
        }).ToList(),
        TotalAmount = order.CalculateTotal(),
        CreatedAt = order.CreatedAt
    };
}
```

Command and query separation (CQS) provides a powerful organizational principle for application services. Commands change system state but don't return data (except perhaps an identifier or success indicator). Queries return data but don't change state. This separation clarifies intent and enables different optimization strategies. Commands might use your full domain model and enforce all business rules. Queries might bypass the domain entirely, reading directly from optimized read models or views. This asymmetry is perfectly acceptable—writes need business rule enforcement, reads need performance. Your application services might be organized into separate command and query services:

- **Command services:**`CreateOrderCommand`, `SubmitOrderCommand`, `CancelOrderCommand`
- **Query services:**`GetOrderByIdQuery`, `GetCustomerOrdersQuery`, `SearchOrdersQuery`

Application services define interfaces for infrastructure dependencies they need. These interfaces live in the application layer, while implementations live in infrastructure. This inverts the dependency—infrastructure depends on application, not the reverse. Your application service declares `IOrderRepository`, `IEmailService`, or `IPaymentGateway` interfaces describing what it needs. The infrastructure layer provides concrete implementations. This inversion enables testing with mocks and allows infrastructure to be swapped without changing application code. It's the Dependency Inversion Principle in action.

Error handling in application services should distinguish between different failure types. Business rule violations throw domain exceptions that the presentation layer can translate into user-friendly messages. Infrastructure failures (database unavailable, external service timeout) throw different exceptions that might trigger retry logic or circuit breakers. Validation failures return structured error information. Your application services should catch and handle exceptions appropriately:

```
public async Task<Result<OrderDto>>
SubmitOrderAsync(SubmitOrderCommand command)
{
    try
    {
        // Validation
        var validationResult = await
_validator.ValidateAsync(command);
        if (!validationResult.IsValid)
            return
Result<OrderDto>.Failure(validationResult.Errors);

        // Domain operations
        var order = await
_orderRepository.GetByIdAsync(command.OrderId);
        if (order == null)
            return Result<OrderDto>.NotFound($"Order
{command.OrderId} not found");

        order.Submit();
        await _orderRepository.UpdateAsync(order);
        await _unitOfWork.CommitAsync();

        return Result<OrderDto>.Success(MapToDto(order));
    }
    catch (DomainException ex)
    {
        return Result<OrderDto>.Failure(ex.Message);
    }
    catch (InfrastructureException ex)
    {
        _logger.LogError(ex, "Infrastructure error 
submitting order");
        throw; // Let infrastructure handle retries
    }
}
```

The application services layer should remain thin, delegating to the domain for business logic and to infrastructure for technical concerns. If your application services contain complex conditional logic or calculations, that logic probably belongs in the domain. Application services coordinate and orchestrate—they're the glue between layers, not the place for business rules. Keep them focused on their orchestration role, and your architecture will remain clean and maintainable. When you find yourself writing complex logic in an application service, ask whether it represents a business rule that belongs in an entity or a technical concern that belongs in infrastructure.

5.4 Comparing Onion and Clean Architecture

Onion Architecture and Clean Architecture share fundamental principles but differ in emphasis and layer organization. Both architectures place the domain at the center, enforce the dependency rule, and promote separation of concerns. Both use dependency inversion to keep the domain independent of infrastructure. However, Clean Architecture explicitly defines four layers (Entities, Use Cases, Interface Adapters, Frameworks) while Onion Architecture focuses on the domain core surrounded by application services and infrastructure. Understanding these differences helps you choose the right approach for your context and recognize that both patterns solve similar problems with slightly different perspectives.

The layer structure represents the most visible difference between these architectures. Clean Architecture's four layers create clear separation between entities (domain objects), use cases (application logic), interface adapters (controllers, presenters, gateways), and frameworks (UI, database, external services). Onion Architecture typically uses three main layers: domain core, application services, and infrastructure, with the presentation layer sometimes considered part of infrastructure. Here's how the layers map between architectures:

- **Clean Architecture Entities↔Onion Domain Core:** Both contain domain entities and business rules
- **Clean Architecture Use Cases↔Onion Application Services:** Both orchestrate domain logic for specific scenarios
- **Clean Architecture Interface Adapters↔Onion Infrastructure:** Both handle external concerns and adapt between layers
- **Clean Architecture Frameworks↔Onion Infrastructure:** Both contain framework-specific implementations

Domain modeling emphasis differs between the two approaches. Onion Architecture explicitly emphasizes Domain-Driven Design concepts like aggregates, repositories, and domain services. The architecture emerged from the DDD community and naturally incorporates DDD tactical patterns. Clean Architecture takes a more general approach, focusing on dependency management and layer separation without prescribing specific domain modeling techniques. You can implement Clean Architecture with or without DDD patterns, while Onion Architecture assumes you're using DDD concepts. This makes Onion Architecture more opinionated about domain design but also more prescriptive for teams new to domain-centric thinking.

The dependency rule operates identically in both architectures: dependencies point inward toward the domain, never outward. However, Clean Architecture makes this rule more explicit through its concentric circle diagram and detailed explanation of how each layer depends on inner layers. Onion Architecture presents the same concept but focuses more on the practical implementation of keeping the domain at the center.

Both architectures achieve the same goal—protecting business logic from infrastructure concerns—but Clean Architecture provides more theoretical foundation while Onion Architecture offers more practical guidance for implementation.

Testing strategies align closely between both architectures because they share the same dependency structure. Both enable testing the domain without infrastructure dependencies, testing application logic with mocked repositories, and testing infrastructure components in isolation. The testability benefits are identical—you can test business logic independently of databases, UI frameworks, and external services. However, Clean Architecture's explicit separation of use cases into their own layer makes it slightly easier to test orchestration logic independently. Onion Architecture achieves the same testability but combines use case orchestration with application services, which some teams find more pragmatic.

Practical implementation differences emerge when organizing your Visual Studio solution. A Clean Architecture solution might have projects like `Domain`, `Application`, `Infrastructure`, `WebUI`, and `Persistence`, with clear separation between interface adapters and frameworks. An Onion Architecture solution typically uses `Domain`, `Application`, `Infrastructure`, and `Web`, with infrastructure encompassing both persistence and external services. Here's a comparison:

```
// Clean Architecture Solution Structure
YourApp.Domain/              // Entities
YourApp.Application/         // Use Cases
YourApp.Infrastructure/      // Interface Adapters
(implementations)
YourApp.Persistence/         // Database (Framework)
YourApp.WebUI/              // Web Framework

// Onion Architecture Solution Structure
YourApp.Domain/              // Domain Core
YourApp.Application/         // Application Services
YourApp.Infrastructure/      // All infrastructure concerns
YourApp.Web/                // Presentation layer
```

The choice between Onion and Clean Architecture often comes down to team background and project context. Teams with strong DDD experience naturally gravitate toward Onion Architecture because it explicitly incorporates DDD patterns. Teams coming from traditional layered architectures might find Clean Architecture's explicit layer definitions easier to understand initially. For greenfield projects with complex business logic, Onion Architecture's DDD emphasis provides valuable structure. For projects migrating from existing architectures or with simpler domains, Clean Architecture's flexibility might be advantageous. Both architectures deliver the same core benefits: maintainable code, testable systems, and business logic protected from infrastructure concerns.

In practice, many successful applications blend concepts from both architectures. You might use Clean Architecture's four-layer structure but incorporate Onion Architecture's emphasis on aggregates and repositories. Or you might follow Onion Architecture's layer organization while adopting Clean Architecture's explicit separation of use cases. The principles matter more than strict adherence to one pattern. Focus on keeping dependencies pointing inward, isolating your domain from infrastructure, and maintaining clear layer boundaries. Whether you call it Clean Architecture, Onion Architecture, or domain-centric architecture, you're building systems that prioritize business logic and resist the forces of entropy that plague poorly structured applications.

Both architectures continue evolving as the development community gains experience with domain-centric design. Modern variations incorporate concepts like CQRS (Command Query Responsibility Segregation), event sourcing, and microservices while maintaining the core principle of domain independence. The fundamental insight remains constant: place your business logic at the center, protect it from external concerns, and structure dependencies to flow inward. This insight transcends any specific architectural pattern and represents a mature approach to software design that serves teams well across diverse projects and technologies. Master these principles, and you'll build better software regardless of which specific architectural pattern you choose to follow.

6 Chapter 5: Designing the Domain Layer

The domain layer represents the heart of your application. It contains the business logic, rules, and behaviors that define what your system does and why it exists. Unlike infrastructure concerns such as databases or web frameworks, the domain layer focuses exclusively on modeling the problem space your application addresses. When designed correctly, this layer becomes independent of external technologies, making it easier to test, maintain, and evolve over time. Your domain layer should express business concepts in code that domain experts can understand and validate.

Building a rich domain layer requires understanding several key building blocks. Entities represent objects with unique identities that persist over time. Value objects capture descriptive characteristics without identity. Domain services encapsulate operations that don't naturally belong to any single entity. Domain events communicate important state changes within your system. Each of these elements plays a specific role in creating an expressive, maintainable domain model. The challenge lies in knowing when to use each pattern and how to combine them effectively.

Many developers struggle with domain layer design because they've worked primarily with anemic domain models—classes that contain only data with no behavior. These models push all logic into service layers, creating procedural code disguised as object-oriented design. A properly designed domain layer inverts this approach. It places behavior alongside data, enforces invariants through encapsulation, and makes invalid states unrepresentable. This chapter explores how to design domain layers that truly capture business complexity while remaining clean and testable.

6.1 Entities and Value Objects

Entities are objects defined primarily by their identity rather than their attributes. A customer with ID 12345 remains the same customer even if their name, address, or email changes. The identity persists throughout the entity's lifetime, providing continuity across state changes. In C#, entities typically contain a unique identifier property and implement equality based on that identifier. They also contain business logic methods that modify their state while maintaining invariants. Entities are mutable by nature because they represent concepts that evolve over time.

Consider a practical example of an `Order` entity in an e-commerce system:

```
public class Order
{
    public Guid Id { get; private set; }
    public CustomerId CustomerId { get; private set; }
    public OrderStatus Status { get; private set; }
    private readonly List<OrderLine> _orderLines;
    public IReadOnlyCollection<OrderLine> OrderLines =>
_orderLines.AsReadOnly();

    public Order(CustomerId customerId)
    {
        Id = Guid.NewGuid();
        CustomerId = customerId ?? throw new
ArgumentNullException(nameof(customerId));
        Status = OrderStatus.Draft;
        _orderLines = new List<OrderLine>();
    }

    public void AddLine(ProductId productId, int quantity,
decimal unitPrice)
    {
        if (Status != OrderStatus.Draft)
            throw new InvalidOperationException("Cannot
modify a submitted order");

        if (quantity <= 0)
            throw new ArgumentException("Quantity must be
positive", nameof(quantity));

        _orderLines.Add(new OrderLine(productId, quantity,
unitPrice));
```

```
    }

    public void Submit()
    {
        if (!_orderLines.Any())
            throw new InvalidOperationException("Cannot
submit an empty order");

        Status = OrderStatus.Submitted;
    }
}
```

Value objects, in contrast, have no conceptual identity. Two addresses with identical street, city, and postal code are interchangeable—they're the same address. Value objects are defined entirely by their attributes. They should be immutable, meaning once created, their state cannot change. If you need a different value, you create a new value object. This immutability makes value objects thread-safe and eliminates entire categories of bugs related to unexpected state changes. Value objects also implement equality based on all their properties rather than identity.

Here's an example of a `Money` value object that encapsulates currency and amount:

```csharp
public class Money : IEquatable<Money>
{
    public decimal Amount { get; }
    public string Currency { get; }

    public Money(decimal amount, string currency)
    {
        if (string.IsNullOrWhiteSpace(currency))
            throw new ArgumentException("Currency is
required", nameof(currency));

        Amount = amount;
        Currency = currency.ToUpperInvariant();
    }

    public Money Add(Money other)
    {
        if (Currency != other.Currency)
            throw new InvalidOperationException("Cannot add
money with different currencies");

        return new Money(Amount + other.Amount, Currency);
    }

    public bool Equals(Money other)
    {
        if (other is null) return false;
        return Amount == other.Amount && Currency ==
other.Currency;
    }

    public override bool Equals(object obj) => Equals(obj
as Money);

    public override int GetHashCode() =>
HashCode.Combine(Amount, Currency);
}
```

The distinction between entities and value objects matters for several reasons. First, it affects how you implement equality. Entities compare by ID, while value objects compare by all properties. Second, it influences persistence strategies. Entities typically map to their own database tables with primary keys, while value objects often embed within entity tables. Third, it impacts how you handle changes. Entities track state transitions, while value objects get replaced entirely. Understanding these differences helps you model your domain accurately.

Value objects also reduce primitive obsession—the tendency to represent domain concepts using basic types like strings and integers. Instead of passing around a string for an email address, create an `EmailAddress` value object that validates format and encapsulates email-related behavior. Instead of using a decimal for money, use a `Money` value object that prevents currency mismatches. This approach makes your code more expressive and catches errors at compile time rather than runtime. Your domain model becomes a rich vocabulary that speaks the language of your business.

When deciding between entities and value objects, ask yourself: does this concept have a lifecycle and identity that matters to the business? If a customer cares about tracking a specific order over time, it's an entity. If two phone numbers with the same digits are completely interchangeable, it's a value object. Some concepts blur the line. An address might be a value object in one context (shipping address on an order) but an entity in another (a customer's saved addresses with IDs). Context determines the appropriate pattern. Let business requirements guide your modeling decisions.

6.2 Domain Services

Domain services encapsulate business logic that doesn't naturally fit within a single entity or value object. When an operation involves multiple aggregates, requires external information, or represents a significant business process, a domain service provides the appropriate home. Unlike application services that orchestrate use cases, domain services contain pure business logic. They operate on domain objects and return domain objects. Domain services should be stateless, receiving all necessary information through method parameters rather than maintaining internal state.

Consider a pricing service that calculates order totals with complex business rules:

```
public class PricingService
{
    public Money CalculateOrderTotal(Order order, Customer
customer,
        IEnumerable<Discount> applicableDiscounts)
    {
        var subtotal = order.OrderLines
            .Select(line => line.Quantity * line.UnitPrice)
            .Aggregate(Money.Zero(order.Currency), (sum,
price) => sum.Add(price));

        var discountAmount = CalculateDiscounts(subtotal,
customer, applicableDiscounts);
        var afterDiscount =
subtotal.Subtract(discountAmount);

        var taxAmount = CalculateTax(afterDiscount,
customer.TaxRegion);

        return afterDiscount.Add(taxAmount);
    }

    private Money CalculateDiscounts(Money subtotal,
Customer customer,
        IEnumerable<Discount> discounts)
    {
        var totalDiscount = Money.Zero(subtotal.Currency);

        foreach (var discount in discounts.Where(d =>
d.IsApplicableTo(customer)))
        {
            totalDiscount =
totalDiscount.Add(discount.Calculate(subtotal));
        }

        return totalDiscount;
    }

    private Money CalculateTax(Money amount, TaxRegion
region)
    {
        return amount.Multiply(region.TaxRate);
    }
}
```

Domain services differ from application services in important ways. Application services handle cross-cutting concerns like transactions, security, and validation. They translate between external requests and domain operations. Domain services, however, contain core business logic that domain experts would recognize and validate. A domain expert might say "we calculate the total by applying customer discounts and then adding tax based on their region." That's domain logic. They wouldn't say "we start a transaction, load the order from the repository, and commit changes." That's application logic.

Avoid overusing domain services, as they can lead to anemic domain models. Before creating a domain service, ask whether the logic truly spans multiple aggregates or whether it belongs within an entity. If an operation primarily affects one entity and uses others only for reference data, place the logic in that entity. For example, an order calculating its own total using injected pricing rules belongs in the `Order` entity. But a transfer between two bank accounts requires a domain service because it coordinates changes to two separate aggregates.

Here's an example of a domain service that coordinates a funds transfer:

```
public class FundsTransferService
{
    public TransferResult Transfer(BankAccount fromAccount,
BankAccount toAccount,
        Money amount)
    {
        if (fromAccount.Currency != amount.Currency)
            throw new InvalidOperationException("Currency
mismatch");

        if (!fromAccount.CanWithdraw(amount))
            return TransferResult.InsufficientFunds();

        fromAccount.Withdraw(amount);
        toAccount.Deposit(amount);

        return TransferResult.Success();
    }
}
```

Domain services should have clear, intention-revealing names that reflect business operations. Names like `OrderPricingService`, `InventoryAllocationService`, or `ShippingCostCalculator` communicate business purpose. Avoid generic names like `OrderService` or `CustomerManager` that could mean anything. Each domain service should have a focused responsibility. If a service grows too large or handles unrelated concerns, split it into multiple services. Keep your domain services cohesive and aligned with business capabilities.

Testing domain services is straightforward because they're stateless and depend only on domain objects. You can instantiate a service, pass in test data, and verify the results without mocking frameworks or complex setup. This testability is a key benefit of keeping domain services pure. They contain business logic without infrastructure dependencies, making them fast to test and easy to understand. Your domain services should be some of the most thoroughly tested code in your application because they encode critical business rules.

6.3 Domain Events

Domain events represent something significant that happened in your domain. They capture state changes that other parts of the system might care about. When an order is placed, a payment is processed, or inventory falls below a threshold, a domain event communicates that fact. Events are immutable records of things that occurred in the past, so they use past-tense names like `OrderPlaced`, `PaymentProcessed`, or `InventoryDepleted`. Domain events enable loose coupling between aggregates and support eventual consistency in distributed systems.

A domain event is typically a simple class containing relevant data:

```
public class OrderPlacedEvent
{
    public Guid OrderId { get; }
    public CustomerId CustomerId { get; }
    public Money TotalAmount { get; }
    public DateTime OccurredAt { get; }

    public OrderPlacedEvent(Guid orderId, CustomerId
customerId,
        Money totalAmount, DateTime occurredAt)
    {
        OrderId = orderId;
        CustomerId = customerId;
        TotalAmount = totalAmount;
        OccurredAt = occurredAt;
    }
}
```

Entities raise domain events when significant state changes occur. Rather than directly calling other aggregates or services, entities add events to a collection. This approach maintains aggregate boundaries and prevents tight coupling. Here's how an `Order` entity might raise an event:

```
public class Order
{
    private readonly List<IDomainEvent> _domainEvents = new
List<IDomainEvent>();
    public IReadOnlyCollection<IDomainEvent> DomainEvents
=> _domainEvents.AsReadOnly();

    public void Submit()
    {
        if (!_orderLines.Any())
            throw new InvalidOperationException("Cannot
submit an empty order");

        Status = OrderStatus.Submitted;
        SubmittedAt = DateTime.UtcNow;

        _domainEvents.Add(new OrderPlacedEvent(Id,
CustomerId, CalculateTotal(),
            SubmittedAt.Value));
    }

    public void ClearEvents()
    {
        _domainEvents.Clear();
    }
}
```

Domain event handlers respond to events by performing additional business logic. A handler might update a different aggregate, send a notification, or trigger a workflow. Handlers should be small and focused, each handling one specific reaction to an event. Here's an example handler that updates customer statistics when an order is placed:

```
public class UpdateCustomerStatisticsHandler :
IDomainEventHandler<OrderPlacedEvent>
{
    private readonly ICustomerRepository
_customerRepository;

    public
UpdateCustomerStatisticsHandler(ICustomerRepository
customerRepository)
    {
        _customerRepository = customerRepository;
    }

    public async Task Handle(OrderPlacedEvent domainEvent)
    {
        var customer = await
_customerRepository.GetByIdAsync(domainEvent.CustomerId);

        customer.RecordOrder(domainEvent.TotalAmount);

        await _customerRepository.UpdateAsync(customer);
    }
}
```

Domain events support two dispatch patterns: immediate and deferred. Immediate dispatch publishes events as soon as they're raised, within the same transaction. This works well for in-process handlers that must complete before the transaction commits. Deferred dispatch collects events during aggregate operations and publishes them after the transaction commits successfully. This approach prevents handlers from executing if the transaction rolls back. Choose based on your consistency requirements and whether handlers need to participate in the same transaction.

Here's a simple implementation of deferred event dispatch:

```
public class DomainEventDispatcher
{
    private readonly IServiceProvider _serviceProvider;

    public DomainEventDispatcher(IServiceProvider
serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    public async Task
DispatchEventsAsync(IEnumerable<IDomainEvent> events)
    {
        foreach (var domainEvent in events)
        {
            var eventType = domainEvent.GetType();
            var handlerType =
typeof(IDomainEventHandler<>).MakeGenericType(eventType);
            var handlers =
_serviceProvider.GetServices(handlerType);

            foreach (var handler in handlers)
            {
                var handleMethod =
handlerType.GetMethod("Handle");
                await (Task)handleMethod.Invoke(handler,
new object[] { domainEvent });
            }
        }
    }
}
```

Domain events provide an audit trail of what happened in your system. By storing events, you create a historical record of state changes. This supports debugging, compliance requirements, and business analytics. Event sourcing takes this further by using events as the primary source of truth, reconstructing aggregate state by replaying events. Even without full event sourcing, storing domain events provides valuable insights into system behavior and supports temporal queries about past states.

Keep domain events focused on business-significant occurrences. Not every property change deserves an event. Raise events when something happens that other parts of the business care about. If a customer changes their email address, that might warrant a `CustomerEmailChanged` event so the notification system can update its records. But if an internal counter increments, that's probably not event-worthy. Let business significance guide your event design. Your events should tell the story of what's happening in your domain.

6.4 Avoiding Anemic Domain Models

An anemic domain model is one where entities contain only data with getters and setters, while all business logic lives in service classes. These models look object-oriented on the surface but behave like procedural code. Martin Fowler identified this as an anti-pattern because it violates the fundamental principle of object-oriented design: combining data and behavior[4]. Anemic models lead to scattered business logic, duplicated validation, and code that's difficult to maintain. They emerge when developers treat entities as mere data containers rather than rich domain objects.

Consider this anemic `Order` entity:

[4] Fowler, Martin (2003). Anemic Domain Model. martinfowler.com

```
// Anemic - DON'T DO THIS
public class Order
{
    public Guid Id { get; set; }
    public CustomerId CustomerId { get; set; }
    public OrderStatus Status { get; set; }
    public List<OrderLine> OrderLines { get; set; }
    public DateTime? SubmittedAt { get; set; }
}
// Anemic service - DON'T DO THIS
public class OrderService
{
    public void SubmitOrder(Order order)
    {
        if (order.OrderLines == null ||
!order.OrderLines.Any())
            throw new InvalidOperationException("Cannot
submit empty order");

        order.Status = OrderStatus.Submitted;
        order.SubmittedAt = DateTime.UtcNow;
    }
}
```

`order.Status`

A rich domain model places behavior within entities, encapsulating business rules and protecting invariants. Here's the same `Order` designed properly:

```
// Rich domain model - DO THIS
public class Order
{
    public Guid Id { get; private set; }
    public CustomerId CustomerId { get; private set; }
    public OrderStatus Status { get; private set; }
    private readonly List<OrderLine> _orderLines;
    public IReadOnlyCollection<OrderLine> OrderLines =>
_orderLines.AsReadOnly();
    public DateTime? SubmittedAt { get; private set; }

    public void Submit()
    {
        if (!_orderLines.Any())
            throw new InvalidOperationException("Cannot
submit empty order");

        if (Status != OrderStatus.Draft)
            throw new InvalidOperationException("Order
already submitted");

        Status = OrderStatus.Submitted;
        SubmittedAt = DateTime.UtcNow;
    }
}
```

Rich domain models use private setters and expose behavior through methods. Instead of `order.Status = OrderStatus.Submitted`, you call `order.Submit()`. This shift from property manipulation to method invocation makes business operations explicit. The code reads like a conversation with domain experts. You're not setting properties; you're submitting orders, processing payments, or shipping products. This expressiveness makes code easier to understand and maintain. New developers can read the entity's public methods and understand what operations the business supports.

Encapsulation is key to rich domain models. Keep collections private and expose them through read-only interfaces. This prevents external code from adding items directly, bypassing validation. Here's how to properly encapsulate a collection:

```
public class Order
{
    private readonly List<OrderLine> _orderLines = new
List<OrderLine>();
    public IReadOnlyCollection<OrderLine> OrderLines =>
_orderLines.AsReadOnly();

    public void AddLine(ProductId productId, int quantity,
decimal unitPrice)
    {
        if (Status != OrderStatus.Draft)
            throw new InvalidOperationException("Cannot
modify submitted order");

        if (quantity <= 0)
            throw new ArgumentException("Quantity must be
positive");

        var existingLine = _orderLines.FirstOrDefault(l =>
l.ProductId == productId);
        if (existingLine != null)
        {
            existingLine.IncreaseQuantity(quantity);
        }
        else
        {
            _orderLines.Add(new OrderLine(productId,
quantity, unitPrice));
        }
    }
}
```

Some developers worry that rich domain models create "fat" entities with too many responsibilities. This concern is valid but often misplaced. The solution isn't to strip behavior from entities but to properly distribute responsibilities. Use value objects to encapsulate complex attributes. Use domain services for operations spanning multiple aggregates. Use domain events to decouple reactions to state changes. When entities focus on their core responsibilities and delegate appropriately, they remain manageable. The alternative—anemic models with bloated service layers—is far worse.

Transitioning from anemic to rich domain models requires a mindset shift. Stop thinking about entities as data structures. Start thinking about them as objects with responsibilities. Ask "what can this entity do?" rather than "what data does it hold?" When you need to change state, add a method that encapsulates the business rule. When you find validation logic in services, move it into the entity. Gradually, your domain model becomes richer and more expressive. Your services become thinner, focusing on orchestration rather than business logic. This is the essence of domain-centric design.

Testing rich domain models is easier than testing anemic ones. You can instantiate an entity, call its methods, and verify the results without any infrastructure. No database, no mocking, no complex setup. Just pure business logic. This testability is a strong indicator of good design. If your entities are hard to test, they probably depend on too much infrastructure. If your tests require extensive mocking, your business logic probably lives in the wrong place. Rich domain models with proper encapsulation are naturally testable because they're self-contained and focused on business rules.

7 Chapter 6: Aggregate Design Patterns

Aggregates represent one of the most powerful yet misunderstood concepts in domain-centric architecture. They serve as the guardians of your business rules, ensuring that data remains consistent and valid throughout its lifecycle. When you design aggregates correctly, you create natural boundaries that protect your domain model from corruption and make your codebase significantly easier to reason about. Poor aggregate design, conversely, leads to tangled dependencies, performance problems, and business logic scattered across multiple layers.

The challenge with aggregates lies not in understanding the theory but in applying it to real-world scenarios. You must balance competing concerns: maintaining consistency, achieving acceptable performance, and keeping your code maintainable. An aggregate that's too large becomes a bottleneck, forcing unnecessary locks and degrading system performance. An aggregate that's too small fails to enforce critical business rules, allowing invalid states to creep into your system. Finding the right size and boundaries requires careful analysis of your domain's invariants and transactional requirements.

This chapter guides you through the practical aspects of aggregate design in C# applications. You'll learn how to identify aggregate boundaries by analyzing consistency requirements, how to implement aggregate roots that enforce invariants, and how to avoid common pitfalls that plague many domain models. The patterns and techniques presented here apply whether you're building a new system from scratch or refactoring an existing application toward a cleaner architecture. By the end of this chapter, you'll have concrete strategies for designing aggregates that protect your business rules while maintaining system performance.

7.1 WHAT ARE AGGREGATES

An aggregate is a cluster of domain objects that you treat as a single unit for data changes. Think of it as a consistency boundary within your domain model. Everything inside an aggregate must remain consistent according to your business rules, while relationships between aggregates can be eventually consistent. This distinction fundamentally shapes how you structure your domain layer and determines which operations can be performed atomically within a single transaction.

Every aggregate has exactly one **aggregate root**, which serves as the entry point for all external access. Other objects can reference the aggregate only through its root, never directly accessing internal entities or value objects. This restriction prevents external code from bypassing business rules or placing the aggregate in an invalid state. For example, in an order management system, the `Order` entity serves as the aggregate root, and external code cannot directly modify `OrderLine` items without going through the `Order` itself.

Aggregates enforce **invariants**—business rules that must always hold true. When you add a line item to an order, the order's total must be recalculated. When you change a product's price, you might need to update minimum order quantities. These rules define what makes your domain model valid, and aggregates ensure these rules cannot be violated. The aggregate root contains methods that encapsulate these rules, making it impossible for external code to create invalid states.

Consider a simple e-commerce scenario with products and inventory. You might initially think each product and its inventory should be separate entities. However, if your business rule states that you cannot sell more items than you have in stock, then `Product` and `Inventory` must be part of the same aggregate. The `Product` aggregate root would expose a method like `ReserveStock(int quantity)` that checks inventory levels and either succeeds or throws a domain exception. This design makes it impossible to violate the inventory rule.

Aggregates also define transactional boundaries in your persistence layer. When you save an aggregate, you save all its internal entities and value objects together in a single database transaction. This ensures that your database never contains partially updated aggregates. If you need to update multiple aggregates together, you're working across transactional boundaries, which requires different patterns like domain events or sagas to maintain consistency.

The aggregate pattern originated in Domain-Driven Design, introduced by Eric Evans in 2003[5]. While the concept has evolved over two decades of practical application, its core purpose remains unchanged: providing a clear, enforceable boundary around related objects that must change together. Understanding this fundamental purpose helps you make better design decisions when modeling your own domain.

In C#, you typically implement aggregates as classes with private setters and internal collections. The aggregate root exposes public methods that represent domain operations, while keeping its internal state protected. Here's a basic structure:

[5] Evans, Eric (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional.

```
public class Order
{
    private readonly List<OrderLine> _lines = new();

    public OrderId Id { get; private set; }
    public CustomerId CustomerId { get; private set; }
    public OrderStatus Status { get; private set; }
    public IReadOnlyCollection<OrderLine> Lines =>
_lines.AsReadOnly();

    public void AddLine(ProductId productId, int quantity,
decimal price)
    {
        if (Status != OrderStatus.Draft)
            throw new InvalidOperationException("Cannot
modify submitted orders");

        var existingLine = _lines.FirstOrDefault(l =>
l.ProductId == productId);
        if (existingLine != null)
            existingLine.IncreaseQuantity(quantity);
        else
            _lines.Add(new OrderLine(productId, quantity,
price));
    }
}
```

This design prevents external code from directly manipulating the `_lines` collection. All modifications flow through the `AddLine` method, which enforces the business rule that submitted orders cannot be modified. The aggregate root maintains complete control over its internal state, ensuring invariants are never violated regardless of how external code attempts to use it.

7.2 Defining Consistency Boundaries

Identifying where one aggregate ends and another begins represents the most critical decision in aggregate design. The key question to ask is: *which objects must be consistent with each other at all times?* Objects that must maintain immediate consistency belong in the same aggregate. Objects that can tolerate temporary inconsistency should be separated into different aggregates. This distinction directly impacts your system's scalability, performance, and ability to handle concurrent operations.

Start by identifying your domain's **invariants**—the rules that must never be broken. An invariant might state that an order's total must equal the sum of its line items, or that a bank account balance must never go negative. These rules define your consistency boundaries. If violating a rule would corrupt your business data or lead to incorrect behavior, the objects involved in that rule must be part of the same aggregate. Document these invariants explicitly; they form the foundation of your aggregate design.

Consider a conference management system where attendees register for sessions. You might have these potential invariants:

- A session cannot exceed its maximum capacity
- An attendee cannot register for overlapping sessions
- An attendee must have a valid ticket to register for sessions
- The conference cannot have more attendees than the venue capacity

`SessionRegistrationsAttendeeSessionRegistrations`

Beware of the temptation to create large aggregates that enforce every possible business rule immediately. Large aggregates create contention in concurrent scenarios and degrade performance. If a hundred users try to register for the same conference simultaneously, and you've made `Conference` an aggregate containing all attendees and sessions, your system will serialize those operations, creating a bottleneck. Instead, consider which rules truly require immediate consistency and which can be eventually consistent.

The **eventual consistency** approach allows you to split aggregates while still maintaining business rules. In the conference example, you might enforce session capacity immediately within the `Session` aggregate, but enforce venue capacity through a background process that monitors total registrations and closes registration when the limit is reached. This introduces a small window where you might slightly exceed capacity, but the trade-off often makes sense for system scalability.

Transaction boundaries provide another lens for identifying aggregates. Each aggregate should be loadable and savable in a single database transaction. If you find yourself needing to load multiple aggregates and save them together to maintain consistency, you've likely drawn your boundaries incorrectly. Either those objects should be part of the same aggregate, or you should redesign to use eventual consistency between them. Your repository pattern should reinforce this: one repository per aggregate root, and each repository operation works with exactly one aggregate instance.

Real-world example: In an e-commerce system, consider the relationship between `Order`, `Customer`, and `Product`. The order contains line items referencing products, and belongs to a customer. Should these all be one aggregate? No. The order needs product IDs and prices, but not the full product details. The order needs the customer ID, but not the customer's full profile. Here's how you'd structure it:

When defining consistency boundaries, also consider the **user's mental model** of your domain. Users naturally think in terms of certain groupings. An order with its line items feels like a single unit. A product catalog with thousands of products does not. Aligning your aggregates with user expectations makes your code more intuitive and reduces the likelihood of design errors. If your aggregate boundaries feel unnatural or require complex explanations, that's a signal to reconsider your design.

7.3 Aggregate Roots and Invariants

The aggregate root serves as the gatekeeper for all operations on the aggregate. It's the only entity within the aggregate that external code can hold a reference to, and it's responsible for maintaining all invariants across the entire aggregate. This centralization of responsibility makes your domain model predictable and safe. When you need to understand how a particular business rule is enforced, you look at the aggregate root's methods. When you need to modify behavior, you change the aggregate root's implementation.

Design your aggregate root's public interface around **domain operations**, not data access. Instead of exposing setters that allow external code to manipulate state directly, provide methods that represent meaningful business actions. For example, an `Order` aggregate root should have methods like `AddItem()`, `RemoveItem()`, `ApplyDiscount()`, and `Submit()`. Each method encapsulates the business rules for that operation, ensuring the aggregate remains valid after the operation completes.

Invariants fall into two categories: **single-entity invariants** and **aggregate-wide invariants**. Single-entity invariants apply to one entity within the aggregate, like "an order line quantity must be positive." Aggregate-wide invariants span multiple entities, like "the order total must equal the sum of all line totals." Your aggregate root must enforce both types. Single-entity invariants can be enforced within the entity itself, but the aggregate root must coordinate aggregate-wide invariants.

Here's a comprehensive example of an aggregate root enforcing multiple invariants:

```
public class ShoppingCart
{
    private readonly List<CartItem> _items = new();
    private const int MaxItems = 50;
    private const decimal MaxTotalValue = 10000m;

    public CartId Id { get; private set; }
    public CustomerId CustomerId { get; private set; }
    public IReadOnlyCollection<CartItem> Items =>
_items.AsReadOnly();

    public decimal TotalValue => _items.Sum(i =>
i.Subtotal);

    public void AddItem(ProductId productId, int quantity,
decimal unitPrice)
    {
        if (quantity <= 0)
            throw new DomainException("Quantity must be
positive");

        if (_items.Count >= MaxItems)
            throw new DomainException($"Cannot exceed
{MaxItems} items in cart");

        var existingItem = _items.FirstOrDefault(i =>
i.ProductId == productId);

        if (existingItem != null)
        {
            var newQuantity = existingItem.Quantity +
quantity;
            var newSubtotal = newQuantity * unitPrice;
            var newTotal = TotalValue -
existingItem.Subtotal + newSubtotal;

            if (newTotal > MaxTotalValue)
                throw new DomainException($"Cart value
cannot exceed {MaxTotalValue:C}");

            existingItem.UpdateQuantity(newQuantity);
        }
        else
        {
            var newSubtotal = quantity * unitPrice;
            if (TotalValue + newSubtotal > MaxTotalValue)
                throw new DomainException($"Cart value
cannot exceed {MaxTotalValue:C}");

            _items.Add(new CartItem(productId, quantity,
unitPrice));
        }
    }

    public void RemoveItem(ProductId productId)
    {
        var item = _items.FirstOrDefault(i => i.ProductId
== productId);
        if (item == null)
            throw new DomainException("Item not found in
cart");
```

```
        _items.Remove(item);
    }
}
```

`_items`

The aggregate root should also control the **lifecycle** of entities within the aggregate. When you add an order line to an order, the order creates the order line instance. When you remove a line, the order removes it from its collection. This prevents orphaned entities and ensures that all entities within the aggregate are properly initialized and maintained. The aggregate root acts as a factory for its internal entities, guaranteeing they're created in valid states.

Identity management within aggregates requires careful consideration. The aggregate root always has a globally unique identifier that external code uses to reference it. Internal entities might have identifiers that are only unique within the aggregate, or they might not have explicit identifiers at all if they're value objects. For example, an `OrderLine` might have a line number that's unique within its order but not globally unique. This design reinforces that order lines have no meaning outside their parent order.

When invariants span multiple aggregates, you cannot enforce them immediately within a single transaction. Instead, use **domain events** to maintain eventual consistency. For example, if you have a business rule that "a customer cannot have more than three active orders," you cannot enforce this within the `Order` aggregate because it would require loading all of the customer's orders. Instead, the `Order` aggregate raises an `OrderCreated` event, and a separate process monitors these events and enforces the limit asynchronously.

Aggregate roots should be **self-validating**. They should never allow themselves to be placed in an invalid state. This means validating inputs in every public method and throwing domain exceptions when operations would violate invariants. Don't rely on external validation or assume that calling code will check preconditions. The aggregate root is the ultimate authority on what constitutes a valid state, and it must defend that validity aggressively. This defensive approach prevents bugs from propagating through your system and makes your domain model trustworthy.

Consider using **factory methods** on your aggregate root for complex creation scenarios. Instead of exposing a public constructor with many parameters, provide static factory methods that clearly express different ways to create the aggregate. For example, `Order.CreateFromCart(ShoppingCart cart)` or `Order.CreateRecurring(OrderTemplate template)`. These factory methods can enforce creation-time invariants and make your code more readable by explicitly naming the creation scenario.

7.4 Sizing Aggregates Correctly

Determining the right size for your aggregates represents a critical balancing act. Aggregates that are too large create performance bottlenecks, increase contention in concurrent scenarios, and make your code harder to understand and maintain. Aggregates that are too small fail to enforce important business rules and scatter related logic across multiple classes. The goal is to find the sweet spot where your aggregates are just large enough to maintain consistency but no larger.

Start with **small aggregates** as your default approach. Many developers err on the side of making aggregates too large, grouping together entities that don't truly need to be consistent with each other. A good rule of thumb: if you can enforce a business rule through eventual consistency rather than immediate consistency, prefer eventual consistency. This keeps your aggregates small and your system scalable. You can always combine aggregates later if you discover that eventual consistency isn't sufficient, but splitting large aggregates is much more difficult.

The **single aggregate per transaction** principle provides a practical guideline. If you find yourself regularly loading multiple aggregates and modifying them together in a single transaction, that's a strong signal that your aggregate boundaries are wrong. Either those aggregates should be combined, or you should redesign to use eventual consistency. Each business transaction should ideally work with one aggregate instance, load it, modify it through its public methods, and save it back.

Consider the performance implications of aggregate size. Every time you load an aggregate, you load all its internal entities and value objects. If your `Customer` aggregate includes the customer's entire order history, loading a customer to update their email address means loading potentially thousands of orders. This is wasteful and slow. Instead, keep only the data that must be immediately consistent within the aggregate. In this case, `Customer` should contain profile information and perhaps recent activity, but not the full order history.

Collection size within aggregates deserves special attention. If an aggregate contains a collection that can grow unbounded, you have a design problem. An `Order` with order lines is fine because orders typically have a reasonable number of lines. A `Product` with all historical price changes is problematic because that collection grows forever. For unbounded collections, consider these strategies:

- Move the collection to a separate aggregate and reference it by ID
- Keep only recent items in the aggregate and archive older items
- Use a separate read model for historical queries

- Implement pagination within the aggregate if you must keep the collection

Concurrent access patterns should influence your aggregate sizing decisions. If multiple users frequently need to modify the same aggregate simultaneously, you'll experience lock contention and reduced throughput. For example, if you model a `Conference` as a single aggregate containing all sessions and registrations, every registration operation locks the entire conference. Instead, make each `Session` its own aggregate. This allows concurrent registrations for different sessions, dramatically improving system performance under load.

Here's a practical example comparing different aggregate sizing approaches for a blog system:

Watch for **aggregate sprawl** in your codebase. If you find yourself with dozens of tiny aggregates that are always loaded and modified together, you've gone too far in the other direction. Some developers, after learning about small aggregates, create separate aggregates for every entity. This is equally problematic. The test is whether the entities truly can be inconsistent with each other. If they cannot, they belong in the same aggregate.

Use **aggregate metrics** to evaluate your design decisions. Track how often aggregates are loaded, how long load operations take, how many entities are typically loaded with each aggregate, and how often concurrent modifications cause conflicts. These metrics reveal whether your aggregates are sized appropriately. If load times are slow, your aggregates might be too large. If you're seeing many cross-aggregate transactions, your aggregates might be too small or your boundaries might be wrong.

Consider the **temporal aspects** of your domain when sizing aggregates. Some data is hot—frequently accessed and modified—while other data is cold—rarely changed once created. Keep hot data in your aggregates and move cold data to separate aggregates or read models. For example, an `Order` aggregate might contain current order details but not the full audit trail of every change ever made to the order. The audit trail can be stored separately and queried when needed.

Finally, remember that aggregate design is not set in stone. As you learn more about your domain and as your system's usage patterns evolve, you may need to adjust aggregate boundaries. This is normal and expected. The key is to make these changes deliberately, understanding the trade-offs involved. Document your reasoning for aggregate boundaries so that future developers (including yourself) understand why the design is the way it is. Good aggregate design is as much about communication and maintainability as it is about technical correctness.

8 Chapter 7: Building the Application Layer

The application layer serves as the orchestrator of your domain-centric architecture, coordinating the flow of data between the presentation layer and your domain model. This layer contains the use cases of your system—the specific actions users can perform and the business workflows they trigger. Unlike the domain layer, which focuses on business rules and invariants, the application layer focuses on application-specific logic and workflow coordination. It translates requests from the outside world into domain operations and prepares responses suitable for presentation.

Think of the application layer as a conductor leading an orchestra. The domain layer contains the talented musicians with their instruments and skills, but the conductor determines when each section plays, how they harmonize, and what the final performance sounds like. Similarly, your application layer doesn't contain business rules itself—those belong in the domain—but it orchestrates how those rules are applied in specific scenarios. This separation ensures your business logic remains pure and testable while your application logic handles the practical concerns of coordinating operations, managing transactions, and preparing data for consumers.

The application layer sits between the domain core and the outer layers of your architecture. It depends on the domain layer but remains independent of infrastructure concerns like databases, external APIs, or UI frameworks. This positioning allows you to change how data is stored or how users interact with your system without modifying your application logic. The layer typically contains application services, command and query handlers, data transfer objects, and mapping logic. Understanding how to structure this layer effectively determines whether your architecture remains maintainable as complexity grows or devolves into a tangled mess of dependencies.

In this chapter, you'll learn how to design and implement a robust application layer that coordinates domain operations without leaking business logic. You'll discover how to structure use cases as discrete, testable units of work. You'll explore the Command Query Separation pattern and understand when to apply it. You'll see how application services differ from domain services and learn strategies for mapping between domain entities and data transfer objects. By the end of this chapter, you'll have the knowledge to build an application layer that keeps your architecture clean, your code testable, and your business logic properly isolated.

8.1 Implementing Use Cases

A use case represents a single, specific action a user can perform in your system. Each use case should accomplish one clear goal, such as registering a new customer, placing an order, or updating a product price. In domain-centric architecture, use cases live in the application layer and orchestrate domain objects to fulfill their purpose. They retrieve entities from repositories, invoke domain methods that enforce business rules, and persist changes back through repositories. This approach keeps your use cases focused on workflow coordination rather than business logic implementation.

Consider a use case for placing an order in an e-commerce system. The use case needs to retrieve the customer, validate their account status, check product availability, create an order aggregate, apply any relevant discounts, and save the order. Notice that the use case coordinates these steps but doesn't implement the business rules itself. The customer aggregate determines if the account is valid, the product aggregate manages inventory, and the order aggregate calculates totals and applies discounts. The use case simply orchestrates these domain operations in the correct sequence:

```
public class PlaceOrderUseCase
{
    private readonly ICustomerRepository
_customerRepository;
    private readonly IProductRepository _productRepository;
    private readonly IOrderRepository _orderRepository;
    private readonly IUnitOfWork _unitOfWork;

    public PlaceOrderUseCase(
        ICustomerRepository customerRepository,
        IProductRepository productRepository,
        IOrderRepository orderRepository,
        IUnitOfWork unitOfWork)
    {
        _customerRepository = customerRepository;
        _productRepository = productRepository;
        _orderRepository = orderRepository;
        _unitOfWork = unitOfWork;
    }

    public async Task<OrderResult>
ExecuteAsync(PlaceOrderRequest request)
    {
        // Retrieve customer aggregate
        var customer = await
_customerRepository.GetByIdAsync(request.CustomerId);
        if (customer == null)
            return OrderResult.Failure("Customer not
found");

        // Validate customer can place orders (domain
logic)
        if (!customer.CanPlaceOrders())
            return OrderResult.Failure("Customer account is
not active");

        // Retrieve products and check availability
        var products = await _productRepository
            .GetByIdsAsync(request.Items.Select(i =>
i.ProductId));

        // Create order aggregate (domain logic enforces
rules)
        var order = Order.Create(customer.Id,
request.ShippingAddress);

        foreach (var item in request.Items)
        {
            var product = products.FirstOrDefault(p => p.Id
== item.ProductId);
            if (product == null)
                return OrderResult.Failure($"Product
{item.ProductId} not found");

            // Domain method checks inventory and adds item
            var result = order.AddItem(product,
item.Quantity);
            if (!result.IsSuccess)
                return OrderResult.Failure(result.Error);
        }

        // Apply customer discounts (domain logic)
        customer.ApplyDiscountsTo(order);

        // Persist changes
        await _orderRepository.AddAsync(order);
        await _unitOfWork.CommitAsync();

        return OrderResult.Success(order.Id);
    }
}
```

Each use case should be a separate class with a single public method, typically named `Execute` or `ExecuteAsync`. This structure makes use cases easy to test, understand, and maintain. You can test each use case in isolation by mocking its repository dependencies. You can understand what the use case does by reading one method. You can modify one use case without affecting others. This granular approach scales better than large service classes with dozens of methods handling different scenarios.

Use cases accept request objects as input and return result objects as output. Request objects contain all the data needed to execute the use case, such as identifiers, values, and options. Result objects indicate success or failure and contain any data the caller needs, such as created entity identifiers or error messages. This pattern decouples your use cases from specific presentation technologies. Whether the request comes from a REST API, a Blazor component, or a background job, the use case remains the same:

```
public class PlaceOrderRequest
{
    public Guid CustomerId { get; set; }
    public Address ShippingAddress { get; set; }
    public List<OrderItemRequest> Items { get; set; }
}

public class OrderItemRequest
{
    public Guid ProductId { get; set; }
    public int Quantity { get; set; }
}

public class OrderResult
{
    public bool IsSuccess { get; private set; }
    public Guid? OrderId { get; private set; }
    public string Error { get; private set; }

    public static OrderResult Success(Guid orderId) =>
        new OrderResult { IsSuccess = true, OrderId =
orderId };

    public static OrderResult Failure(string error) =>
        new OrderResult { IsSuccess = false, Error = error
};
}
```

Transaction management belongs in the application layer, not the domain layer. Your use cases should define transaction boundaries by coordinating with a Unit of Work pattern. This ensures that all changes within a use case either succeed together or fail together, maintaining data consistency. The domain layer shouldn't know about transactions—it focuses on enforcing business rules. The application layer handles the practical concern of ensuring those rules are applied atomically. This separation keeps your domain pure while giving you control over transaction scope and performance.

Error handling in use cases should distinguish between expected business rule violations and unexpected technical failures. When a customer tries to place an order but their account is suspended, that's an expected scenario your domain model handles by returning a failure result. When the database connection fails, that's an unexpected technical error that should propagate as an exception. Your use cases should catch and handle domain-level failures gracefully, converting them into appropriate result objects, while allowing infrastructure exceptions to bubble up for centralized handling. This approach gives callers clear information about what went wrong and whether they can take corrective action.

Consider organizing use cases by feature or aggregate rather than by technical layer. Instead of a single `OrderService` with twenty methods, create separate use case classes like `PlaceOrderUseCase`, `CancelOrderUseCase`, and `ShipOrderUseCase`. This organization makes your codebase easier to navigate and reduces merge conflicts when multiple developers work on different features. It also makes testing more focused—each test class corresponds to one use case with one responsibility. As your application grows, this structure scales naturally without creating massive service classes that become maintenance nightmares.

8.2 Command and Query Separation

Command Query Separation (CQS) is a principle stating that methods should either change state or return data, but not both. Commands perform actions and modify state but don't return data beyond success or failure indicators. Queries return data but don't modify state. This separation creates clearer, more predictable code. When you call a query, you know it won't have side effects. When you execute a command, you know it's changing something. This predictability makes code easier to reason about, test, and optimize.

In the application layer, CQS manifests as separate command handlers and query handlers. Command handlers orchestrate domain operations that change state, such as creating orders, updating customer information, or processing payments. Query handlers retrieve data for display or reporting, often bypassing the domain model entirely to query the database directly for performance. This asymmetry is intentional—commands flow through your rich domain model to enforce business rules, while queries take the shortest path to the data consumers need:

- **Commands:** Modify state, enforce business rules through domain model, return minimal data (success/failure, created IDs)
- **Queries:** Read-only operations, can bypass domain model, return DTOs optimized for specific views
- **Validation:** Commands validate through domain invariants, queries validate input parameters only
- **Performance:** Commands prioritize correctness, queries prioritize speed and can use denormalized views

Command handlers follow the use case pattern described earlier. They accept a command object, coordinate domain operations, and return a result indicating success or failure. Here's an example of a command handler for updating customer information:

```
public class UpdateCustomerInfoCommand
{
    public Guid CustomerId { get; set; }
    public string Email { get; set; }
    public string PhoneNumber { get; set; }
}

public class UpdateCustomerInfoCommandHandler
{
    private readonly ICustomerRepository _repository;
    private readonly IUnitOfWork _unitOfWork;

    public UpdateCustomerInfoCommandHandler(
        ICustomerRepository repository,
        IUnitOfWork unitOfWork)
    {
        _repository = repository;
        _unitOfWork = unitOfWork;
    }

    public async Task<Result>
HandleAsync(UpdateCustomerInfoCommand command)
    {
        var customer = await
_repository.GetByIdAsync(command.CustomerId);
        if (customer == null)
            return Result.Failure("Customer not found");

        // Domain method enforces email format rules
        var emailResult =
customer.UpdateEmail(command.Email);
```

```
        if (!emailResult.IsSuccess)
            return emailResult;

        // Domain method enforces phone number rules
        var phoneResult = 
customer.UpdatePhoneNumber(command.PhoneNumber);
        if (!phoneResult.IsSuccess)
            return phoneResult;

        await _unitOfWork.CommitAsync();
        return Result.Success();
    }
}
```

Query handlers retrieve data without modifying state. They can use Entity Framework directly, Dapper for raw SQL queries, or even read from denormalized read models. The key is that queries don't load full aggregate graphs—they fetch only the data needed for a specific view. This approach improves performance and reduces complexity. You don't need to load an entire order aggregate with all its items, customer details, and shipping information just to display the order number and status in a list:

```
public class GetOrderListQuery
{
    public Guid CustomerId { get; set; }
    public int PageNumber { get; set; }
    public int PageSize { get; set; }
}

public class OrderListItem
{
    public Guid OrderId { get; set; }
    public DateTime OrderDate { get; set; }
    public string Status { get; set; }
    public decimal Total { get; set; }
}

public class GetOrderListQueryHandler
{
    private readonly ApplicationDbContext _context;

    public GetOrderListQueryHandler(ApplicationDbContext
context)
    {
        _context = context;
    }

    public async Task<List<OrderListItem>>
HandleAsync(GetOrderListQuery query)
    {
        return await _context.Orders
            .Where(o => o.CustomerId == query.CustomerId)
            .OrderByDescending(o => o.OrderDate)
            .Skip((query.PageNumber - 1) * query.PageSize)
            .Take(query.PageSize)
            .Select(o => new OrderListItem
            {
                OrderId = o.Id,
                OrderDate = o.OrderDate,
                Status = o.Status.ToString(),
                Total = o.Total
            })
            .ToListAsync();
    }
}
```

The separation between commands and queries enables different optimization strategies. Commands can use eventual consistency, publish domain events, and trigger background processes. Queries can use caching, read replicas, and materialized views. You can scale your read and write workloads independently. In systems with high read-to-write ratios, this flexibility provides significant performance benefits. You might serve queries from a denormalized read database optimized for fast lookups while processing commands through your domain model against a normalized write database.

CQRS (Command Query Responsibility Segregation) takes CQS further by using completely separate models for commands and queries. While CQS separates methods, CQRS separates models and potentially even data stores. This pattern suits complex domains where read and write requirements differ significantly. However, CQRS adds substantial complexity and isn't necessary for most applications. Start with simple CQS in your application layer—separate command handlers from query handlers but use the same database. If you encounter performance bottlenecks or find that your read and write models are diverging significantly, then consider evolving toward CQRS.

When implementing CQS, establish clear naming conventions. Commands use imperative verbs: `CreateOrder`, `UpdateCustomer`, `CancelSubscription`. Queries use descriptive nouns: `GetOrderDetails`, `GetCustomerList`, `GetProductCatalog`. This naming makes the intent immediately clear. You know at a glance whether an operation modifies state or just retrieves data. Consistency in naming and structure across your application layer makes the codebase more navigable and reduces cognitive load when switching between different features.

8.3 Application Services in Detail

Application services are the primary building blocks of your application layer. They expose your use cases to the outside world, providing a clean API that presentation layers can consume. Unlike domain services, which contain business logic that doesn't naturally fit within a single entity, application services contain workflow coordination logic. They orchestrate calls to repositories, domain services, and infrastructure services to fulfill use cases. Application services depend on abstractions defined in the domain layer but implement application-specific concerns like transaction management, security checks, and data transformation.

A well-designed application service focuses on a specific area of functionality, typically aligned with an aggregate or bounded context. For example, an `OrderApplicationService` handles all order-related use cases, while a `CustomerApplicationService` manages customer operations. This organization creates clear boundaries and makes services easier to understand and maintain. Each service should have a manageable number of methods—if a service grows beyond ten to fifteen methods, consider splitting it into more focused services:

```
public interface IOrderApplicationService
{
    Task<Result<Guid>> PlaceOrderAsync(PlaceOrderRequest
request);
    Task<Result> CancelOrderAsync(Guid orderId, string
reason);
    Task<Result> ShipOrderAsync(Guid orderId,
ShippingDetails details);
    Task<Result> CompleteOrderAsync(Guid orderId);
    Task<OrderDetailsDto> GetOrderDetailsAsync(Guid
orderId);
    Task<List<OrderSummaryDto>> GetCustomerOrdersAsync(Guid
customerId);
}

public class OrderApplicationService :
IOrderApplicationService
{
    private readonly IOrderRepository _orderRepository;
    private readonly ICustomerRepository
_customerRepository;
    private readonly IProductRepository _productRepository;
    private readonly IInventoryService _inventoryService;
    private readonly IUnitOfWork _unitOfWork;
    private readonly ILogger<OrderApplicationService>
_logger;

    public OrderApplicationService(
        IOrderRepository orderRepository,
        ICustomerRepository customerRepository,
        IProductRepository productRepository,
        IInventoryService inventoryService,
        IUnitOfWork unitOfWork,
        ILogger<OrderApplicationService> logger)
    {
        _orderRepository = orderRepository;
        _customerRepository = customerRepository;
        _productRepository = productRepository;
        _inventoryService = inventoryService;
        _unitOfWork = unitOfWork;
        _logger = logger;
    }

    public async Task<Result<Guid>>
PlaceOrderAsync(PlaceOrderRequest request)
    {
        try
        {
            var customer = await
_customerRepository.GetByIdAsync(request.CustomerId);
            if (customer == null)
                return Result<Guid>.Failure("Customer not
found");

            if (!customer.CanPlaceOrders())
                return Result<Guid>.Failure("Customer
cannot place orders");

            var order = Order.Create(customer.Id,
request.ShippingAddress);

            foreach (var item in request.Items)
            {
                var product = await
_productRepository.GetByIdAsync(item.ProductId);
                if (product == null)
                    return Result<Guid>.Failure($"Product
{item.ProductId} not found");

                // Check inventory through domain service
                if (!await
_inventoryService.IsAvailableAsync(item.ProductId,
item.Quantity))
                    return
Result<Guid>.Failure($"Insufficient inventory for
{product.Name}");

                var addResult = order.AddItem(product,
item.Quantity);
                if (!addResult.IsSuccess)
                    return
Result<Guid>.Failure(addResult.Error);
            }

            customer.ApplyDiscountsTo(order);

            await _orderRepository.AddAsync(order);
            await
_inventoryService.ReserveInventoryAsync(order);
            await _unitOfWork.CommitAsync();

            _logger.LogInformation("Order {OrderId} placed
successfully", order.Id);
```

```
            return Result<Guid>.Success(order.Id);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error placing order for 
customer {CustomerId}", request.CustomerId);
            throw;
        }
    }

    // Other methods...
}
```

Application services handle cross-cutting concerns that don't belong in the domain layer. Logging, performance monitoring, and audit trails are application-level concerns. The domain layer shouldn't know about logging frameworks or audit systems—it focuses purely on business rules. Application services provide the appropriate place to add these concerns without polluting your domain model. You can log when use cases start and complete, measure execution time, and record who performed which actions, all within the application service without touching domain code.

Security and authorization checks also belong in application services. Before executing a use case, verify that the current user has permission to perform the action. This might involve checking roles, permissions, or ownership. For example, a user should only be able to cancel their own orders, not orders belonging to other customers. Application services enforce these rules before invoking domain operations:

```
public async Task<Result> CancelOrderAsync(Guid orderId,
Guid userId, string reason)
{
    var order = await
_orderRepository.GetByIdAsync(orderId);
    if (order == null)
        return Result.Failure("Order not found");

    // Authorization check - application layer concern
    if (order.CustomerId != userId)
    {
        _logger.LogWarning(
            "User {UserId} attempted to cancel order
{OrderId} belonging to {CustomerId}",
            userId, orderId, order.CustomerId);
        return Result.Failure("You can only cancel your own
orders");
    }

    // Business rule check - domain layer concern
    var cancelResult = order.Cancel(reason);
    if (!cancelResult.IsSuccess)
        return cancelResult;

    await _unitOfWork.CommitAsync();
    return Result.Success();
}
```

Application services coordinate between multiple aggregates when necessary, but they should do so carefully to avoid creating tight coupling. When a use case requires data from multiple aggregates, the application service retrieves them from their respective repositories and coordinates their interactions. However, if you find yourself frequently loading multiple aggregates together, reconsider your aggregate boundaries. Perhaps those entities should be part of the same aggregate, or perhaps you need a domain service to encapsulate the coordination logic.

Exception handling in application services should distinguish between recoverable and non-recoverable errors. Domain rule violations are recoverable—return a failure result that the caller can handle gracefully. Infrastructure failures like database connection errors are typically non-recoverable at the application service level—let them propagate to a global exception handler. Log the error with sufficient context for troubleshooting, but don't try to handle infrastructure exceptions within individual application services. This approach keeps your error handling logic centralized and consistent.

Consider using the mediator pattern to decouple your presentation layer from specific application service implementations. Libraries like MediatR allow you to send commands and queries through a mediator that routes them to appropriate handlers. This pattern reduces direct dependencies and makes it easier to add cross-cutting concerns like validation, logging, or caching through pipeline behaviors. However, the mediator pattern adds indirection and complexity, so evaluate whether the benefits justify the costs for your specific application. For smaller applications, direct dependency injection of application services often provides sufficient flexibility with less complexity.

8.4 DTOs and Mapping Strategies

Data Transfer Objects (DTOs) are simple objects that carry data between layers without containing business logic. They serve as contracts between your application layer and external consumers, isolating your domain model from presentation concerns. DTOs prevent your domain entities from being directly exposed to the outside world, which would create tight coupling and make it difficult to evolve your domain model independently. When your presentation layer needs customer data, you return a `CustomerDto` rather than the `Customer` entity itself. This separation gives you freedom to change your domain model without breaking external contracts.

DTOs should be designed for specific use cases rather than trying to represent entire domain entities. A `CustomerListItemDto` for displaying customers in a grid contains only the fields needed for that view—perhaps just name, email, and registration date. A `CustomerDetailsDto` for a detail page includes more information like address, phone number, and order history. A `CreateCustomerDto` for registration contains only the fields required to create a new customer. This targeted approach keeps DTOs simple and prevents over-fetching data:

```
// For list views
public class CustomerListItemDto
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public DateTime RegisteredDate { get; set; }
    public string Status { get; set; }
}

// For detail views
public class CustomerDetailsDto
{
    public Guid Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string PhoneNumber { get; set; }
    public AddressDto BillingAddress { get; set; }
    public AddressDto ShippingAddress { get; set; }
    public List<OrderSummaryDto> RecentOrders { get; set; }
    public decimal TotalSpent { get; set; }
}

// For create operations
public class CreateCustomerDto
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
}
```

Mapping between domain entities and DTOs can be done manually or with mapping libraries like AutoMapper. Manual mapping gives you complete control and makes the transformation explicit, but it requires more code and maintenance. AutoMapper reduces boilerplate through convention-based mapping but can hide complexity and make debugging harder. For simple mappings where property names match, AutoMapper works well. For complex transformations involving business logic or multiple source objects, manual mapping provides better clarity and control.

Here's an example of manual mapping in an application service. The mapping logic is explicit and easy to understand. You can see exactly how each DTO property is populated:

```
public async Task<CustomerDetailsDto>
GetCustomerDetailsAsync(Guid customerId)
{
    var customer = await
_customerRepository.GetByIdAsync(customerId);
    if (customer == null)
        return null;

    var recentOrders = await
_orderRepository.GetRecentOrdersAsync(customerId, 5);

    return new CustomerDetailsDto
    {
        Id = customer.Id,
        FirstName = customer.FirstName,
        LastName = customer.LastName,
        Email = customer.Email.Value, // Value object
unwrapping
        PhoneNumber = customer.PhoneNumber?.Value,
        BillingAddress =
MapAddress(customer.BillingAddress),
        ShippingAddress =
MapAddress(customer.ShippingAddress),
        RecentOrders =
recentOrders.Select(MapOrderSummary).ToList(),
        TotalSpent = customer.CalculateTotalSpent()
    };
}

private AddressDto MapAddress(Address address)
{
    if (address == null)
        return null;

    return new AddressDto
    {
        Street = address.Street,
        City = address.City,
        State = address.State,
        PostalCode = address.PostalCode,
        Country = address.Country
    };
}

private OrderSummaryDto MapOrderSummary(Order order)
{
    return new OrderSummaryDto
    {
        OrderId = order.Id,
        OrderDate = order.OrderDate,
        Status = order.Status.ToString(),
        Total = order.Total
    };
}
```

When using AutoMapper, configure your mappings in profiles that define how entities map to DTOs. AutoMapper handles simple property-to-property mappings automatically and allows you to specify custom logic for complex transformations. Register your profiles during application startup, and inject `IMapper` into your application services:

```
public class CustomerMappingProfile : Profile
{
    public CustomerMappingProfile()
    {
        CreateMap<Customer, CustomerDetailsDto>()
            .ForMember(dest => dest.Email, opt =>
opt.MapFrom(src => src.Email.Value))
            .ForMember(dest => dest.PhoneNumber, opt =>
opt.MapFrom(src => src.PhoneNumber.Value))
            .ForMember(dest => dest.TotalSpent, opt =>
opt.MapFrom(src => src.CalculateTotalSpent()));

        CreateMap<Address, AddressDto>();
        CreateMap<Order, OrderSummaryDto>()
            .ForMember(dest => dest.Status, opt =>
opt.MapFrom(src => src.Status.ToString()));
    }
}

public class CustomerApplicationService
{
    private readonly ICustomerRepository
_customerRepository;
    private readonly IMapper _mapper;

    public CustomerApplicationService(ICustomerRepository
customerRepository, IMapper mapper)
    {
        _customerRepository = customerRepository;
        _mapper = mapper;
    }
    public async Task<CustomerDetailsDto>
GetCustomerDetailsAsync(Guid customerId)
    {
        var customer = await
_customerRepository.GetByIdAsync(customerId);
        if (customer == null)
            return null;

        return _mapper.Map<CustomerDetailsDto>(customer);
    }
}
```

Be cautious about mapping domain entities to DTOs too early in your application services. If you need to perform business operations on an entity, keep it as a domain object until those operations are complete. Only map to DTOs when preparing data for return to the caller. Premature mapping loses the rich behavior of your domain model and forces you to work with anemic data structures. This mistake often leads to business logic leaking into application services because the domain objects are no longer available to enforce rules.

For command operations, DTOs flow in the opposite direction—from the presentation layer to the application layer. These input DTOs should contain only the data needed to execute the command, validated at the presentation layer for format and required fields. The application service maps the input DTO to domain objects or passes the data to domain methods. Don't pass DTOs directly to domain methods—extract the values and pass them as method parameters. This keeps your domain layer independent of application-layer concerns:

```
public async Task<Result<Guid>>
CreateProductAsync(CreateProductDto dto)
{
    // Validate DTO (could also use FluentValidation)
    if (string.IsNullOrWhiteSpace(dto.Name))
        return Result<Guid>.Failure("Product name is
required");

    if (dto.Price <= 0)
        return Result<Guid>.Failure("Price must be greater
than zero");

    // Create domain entity using factory method
    var product = Product.Create(
        dto.Name,
        dto.Description,
        Money.FromDecimal(dto.Price, dto.Currency),
        dto.Sku);

    if (!product.IsSuccess)
        return Result<Guid>.Failure(product.Error);

    await _productRepository.AddAsync(product.Value);
    await _unitOfWork.CommitAsync();

    return Result<Guid>.Success(product.Value.Id);
}
```

Consider using separate DTOs for requests and responses even when they contain similar data. A `UpdateCustomerRequest` and `UpdateCustomerResponse` make the API contract explicit and allow each to evolve independently. The request might contain fields that don't appear in the response, or vice versa. This separation also makes it clear which direction data flows and prevents confusion about whether a DTO is for input or output. Clear, purpose-specific DTOs improve code readability and reduce errors caused by misunderstanding data flow.

Validation of DTOs should occur at the application layer boundary, before any domain operations begin. Use data annotations, FluentValidation, or custom validation logic to ensure DTOs contain valid data. This validation checks format, required fields, and basic business rules that don't require domain knowledge. More complex business rules that depend on current state or multiple entities are validated by the domain model itself. This layered validation approach catches simple errors early while ensuring complex business rules are enforced where they belong—in the domain layer.

9 CHAPTER 8: THE REPOSITORY PATTERN

The Repository Pattern stands as one of the most critical abstractions in domain-centric architecture. It creates a clear boundary between your domain logic and data persistence mechanisms, allowing your business rules to remain completely ignorant of how data is stored or retrieved. This separation enables you to change database technologies, switch ORMs, or even move from relational to NoSQL databases without touching your domain code. The pattern acts as an in-memory collection facade, presenting your domain objects as if they exist in memory while handling all the complexity of persistence behind the scenes.

Understanding repositories goes beyond simply wrapping database calls in an interface. A well-designed repository respects aggregate boundaries, maintains consistency, and provides meaningful query methods that align with your domain's ubiquitous language. When implemented correctly, repositories make your code more testable by allowing you to substitute in-memory implementations during testing. They also centralize data access logic, preventing the scattering of database queries throughout your application and making it easier to optimize performance or implement caching strategies.

This chapter explores the Repository Pattern from both theoretical and practical perspectives. You'll learn how to design repository interfaces that serve your domain's needs, understand when to use generic versus specific repositories, and discover how the Unit of Work pattern complements repositories to manage transactions. By the end of this chapter, you'll have the knowledge to implement repositories that truly serve your domain layer while keeping infrastructure concerns properly isolated. The examples use C# and Entity Framework Core, but the principles apply regardless of your technology stack.

9.1 Purpose and Benefits of Repositories

The primary purpose of the Repository Pattern is to mediate between the domain layer and data mapping layers using a collection-like interface for accessing domain objects. Rather than having your application services or domain logic directly interact with Entity Framework's `DbContext` or execute SQL queries, repositories provide a clean abstraction that speaks the language of your domain. This abstraction allows your domain layer to request objects using business-meaningful operations like `GetActiveCustomersByRegion()` rather than constructing LINQ queries or writing SQL statements.

Repositories enforce the Dependency Inversion Principle by having the domain layer define repository interfaces while the infrastructure layer provides concrete implementations. Your domain layer declares what data access operations it needs without knowing how those operations are fulfilled. This inversion means your core business logic depends on abstractions rather than concrete database technologies. When you need to change your persistence strategy, you modify only the infrastructure layer implementation, leaving your domain and application layers untouched and unaware of the change.

One of the most significant benefits repositories provide is improved testability. When your application services depend on repository interfaces rather than concrete database implementations, you can easily substitute mock or in-memory implementations during testing. This substitution allows you to test your business logic in isolation without requiring a database connection, making your tests faster, more reliable, and easier to set up. You can verify that your domain logic behaves correctly under various data scenarios without the overhead of database transactions or test data management.

Repositories also serve as a natural place to implement cross-cutting concerns related to data access. You can add caching logic, logging, performance monitoring, or retry policies within repository implementations without polluting your domain layer with these infrastructure concerns. For example, a repository might cache frequently accessed reference data or log slow queries for performance analysis. These concerns remain invisible to the domain layer, which simply requests objects and receives them without knowing about the optimization strategies employed behind the scenes.

The pattern helps maintain aggregate boundaries and consistency rules. Each repository typically corresponds to an aggregate root, ensuring that you can only retrieve and persist complete, consistent aggregates. This alignment prevents partial updates that might violate business invariants. When you load an `Order` aggregate through its repository, you receive the complete order with all its line items, ensuring that any business rules involving the order total and its items can be properly enforced. The repository becomes the gatekeeper that ensures aggregates are always in a valid state.

Repositories centralize query logic, preventing the proliferation of data access code throughout your application. Instead of having LINQ queries scattered across multiple services, you consolidate them in repository implementations. This centralization makes it easier to optimize queries, add indexes, or refactor data access patterns. When you need to improve the performance of customer lookups, you know exactly where to look—in the `CustomerRepository` implementation. This organization also makes it easier for new team members to understand how data is accessed and where to add new query methods.

The collection-like interface that repositories provide simplifies reasoning about data access. Your application code can treat repositories as if they were in-memory collections, using methods like `Add()`, `Remove()`, and `GetById()` without worrying about database connections, transactions, or SQL syntax. This simplification reduces cognitive load and allows developers to focus on business logic rather than data access mechanics. The repository handles all the complexity of translating these collection-like operations into appropriate database commands.

Repositories also facilitate the implementation of domain-driven design patterns like specifications. You can create specification objects that encapsulate query criteria in a reusable, testable way, then pass these specifications to repository methods. This approach allows you to build complex queries from composable pieces while keeping the query logic close to the domain. A `CustomerIsActiveSpecification` can be reused across different repository methods and even tested independently to ensure it correctly identifies active customers according to business rules.

9.2 Designing Repository Interfaces

Designing effective repository interfaces requires careful consideration of your domain's needs and aggregate boundaries. Each repository interface should correspond to an aggregate root, not to every entity in your system. If you have an `Order` aggregate containing `OrderItem` entities, you create an `IOrderRepository` but not an `IOrderItemRepository`. Order items are accessed through their parent order, maintaining the aggregate's consistency boundary. This design prevents external code from modifying order items independently, which could violate business rules about order totals or item quantities.

Repository interfaces should be defined in the domain layer, typically in a namespace like `YourApp.Domain.Repositories`. This placement ensures that the domain layer declares its data access needs without depending on infrastructure concerns. The interface methods should use domain language and return domain entities, not data transfer objects or database-specific types. A method like `GetActiveCustomersByRegion(string region)` clearly expresses a domain concept, while `ExecuteQuery(string sql)` exposes infrastructure details that don't belong in the domain layer.

When designing repository methods, focus on the queries your use cases actually need rather than trying to anticipate every possible query. Start with specific, intention-revealing methods that align with your application's use cases. If you need to find customers who haven't placed orders in the last six months, create a method called `GetInactiveCustomers(TimeSpan inactivityPeriod)` rather than exposing a generic query mechanism. This specificity makes your code more readable and allows repository implementations to optimize these particular queries. You can always add more methods as new use cases emerge.

Repository interfaces should include methods for basic CRUD operations, but expressed in domain terms. Instead of generic `Insert()`, `Update()`, and `Delete()` methods, consider using `Add()`, `Remove()`, and potentially omitting an explicit update method altogether. In many domain-centric architectures, entities are tracked by the Unit of Work, and changes are automatically detected and persisted. The repository's role is to retrieve and add entities to the tracked collection, not to explicitly update them. This approach aligns with how you'd work with an in-memory collection.

```
public interface IOrderRepository
{
    Task<Order?> GetByIdAsync(OrderId id, CancellationToken
cancellationToken = default);
    Task<IReadOnlyList<Order>>
GetByCustomerIdAsync(CustomerId customerId,
CancellationToken cancellationToken = default);
    Task<IReadOnlyList<Order>>
GetPendingOrdersAsync(CancellationToken cancellationToken =
default);
    Task<Order?> GetByOrderNumberAsync(string orderNumber,
CancellationToken cancellationToken = default);

    void Add(Order order);
    void Remove(Order order);

    Task<bool> ExistsAsync(OrderId id, CancellationToken
cancellationToken = default);
}
```

Notice how the interface above uses strongly-typed identifiers like `OrderId` and `CustomerId` rather than primitive types. This approach leverages value objects to make the interface more type-safe and expressive. The methods return domain entities (`Order`) rather than DTOs or anonymous types. Query methods are asynchronous and accept cancellation tokens, following modern C# best practices. The `Add()` and `Remove()` methods are synchronous because they typically just mark entities for addition or removal without immediately hitting the database—the actual persistence happens when the Unit of Work commits.

Consider whether your repository methods should return single entities, collections, or potentially null values. Use nullable reference types (`Order?`) to clearly indicate when a method might not find a matching entity. Return `IReadOnlyList<T>` or `IEnumerable<T>` for collections rather than `List<T>` to prevent callers from modifying the collection directly. This immutability reinforces that changes to entities should go through the repository's `Add()` and `Remove()` methods rather than by manipulating returned collections.

Avoid exposing `IQueryable<T>` from repository interfaces. While it might seem convenient to let callers build their own queries, this approach leaks infrastructure concerns into your domain and application layers. Callers become coupled to LINQ and potentially to your database schema. Instead, add specific query methods to your repository interface as needed. If you find yourself adding many similar query methods, consider implementing the Specification pattern to provide more flexible querying while keeping the abstraction clean.

Repository interfaces should not include methods for complex queries that span multiple aggregates or require joins across aggregate boundaries. Such queries often indicate a need for a separate read model or query service. For example, generating a report that combines customer data, order history, and product information shouldn't go through repositories. Instead, create a dedicated query service in your application layer that can directly access the database using optimized queries or even a separate read database. Repositories focus on loading and persisting aggregates, not on complex reporting queries.

When designing repository interfaces, think about the transactional boundaries your use cases require. If a use case needs to load multiple aggregates and ensure they're all saved or none are saved, you'll need a Unit of Work to coordinate the transaction. The repository interfaces themselves typically don't expose transaction management—that responsibility belongs to the Unit of Work pattern, which we'll explore later in this chapter. Repositories focus on aggregate retrieval and persistence, while the Unit of Work manages the transactional context.

9.3 Generic vs Specific Repositories

The debate between generic and specific repositories has persisted throughout the evolution of domain-driven design. Generic repositories provide a common set of CRUD operations for all entity types through a single interface like `IRepository<T>`. Specific repositories create dedicated interfaces for each aggregate root, such as `IOrderRepository` or `ICustomerRepository`, with methods tailored to that aggregate's needs. Each approach has distinct advantages and trade-offs that affect your architecture's maintainability, expressiveness, and flexibility.

Generic repositories appeal to developers seeking to reduce code duplication. By defining a single `IRepository<T>` interface with methods like `GetByIdAsync()`, `AddAsync()`, and `RemoveAsync()`, you avoid writing similar interfaces for each aggregate. A generic implementation using Entity Framework can handle all entity types with minimal code. This approach seems efficient and follows the DRY (Don't Repeat Yourself) principle. However, this efficiency comes at the cost of losing domain-specific expressiveness and potentially exposing operations that don't make sense for certain aggregates.

The primary weakness of generic repositories is their inability to express domain-specific operations. A generic `IRepository<Customer>` might offer `GetByIdAsync()`, but it can't provide `GetInactiveCustomers()` or `GetCustomersByRegion()` without additional mechanisms. You end up either adding these methods through extension methods, creating a hybrid approach, or exposing `IQueryable<T>` to let callers build custom queries. Each solution compromises the abstraction in different ways, either by scattering repository logic across multiple locations or by leaking infrastructure concerns into calling code.

```
// Generic repository interface
public interface IRepository<T> where T : class
{
    Task<T?> GetByIdAsync(object id, CancellationToken
cancellationToken = default);
    Task<IReadOnlyList<T>> GetAllAsync(CancellationToken
cancellationToken = default);
    void Add(T entity);
    void Remove(T entity);
}

// Usage loses domain expressiveness
var customer = await repository.GetByIdAsync(customerId);
// How do we get inactive customers? GetAllAsync() and
filter in memory?
```

Specific repositories, in contrast, create dedicated interfaces for each aggregate root with methods that express domain concepts. An `ICustomerRepository` includes `GetInactiveCustomers()`, `GetByRegion()`, and `GetByEmailAddress()`—methods that clearly communicate domain operations. This specificity makes your code more readable and maintainable. When you see `customerRepository.GetInactiveCustomers()`, you immediately understand the business intent. The repository interface becomes part of your ubiquitous language, bridging the gap between domain experts and developers.

Specific repositories also allow you to optimize implementations for each aggregate's unique needs. An `OrderRepository` might eagerly load order items and apply specific indexes, while a `ProductRepository` might implement caching for frequently accessed catalog data. These optimizations happen in the infrastructure layer without affecting the domain layer's interface. Generic repositories make such aggregate-specific optimizations more difficult because the implementation must work for all entity types, limiting your ability to fine-tune performance for specific scenarios.

The concern about code duplication with specific repositories is often overstated. While you do create multiple interfaces, each interface is typically small and focused. The implementation can still share common code through base classes or helper methods in the infrastructure layer. You might create an abstract `RepositoryBase<T>` class that handles common Entity Framework operations, then have specific repository implementations inherit from it and add their custom query methods. This approach gives you the expressiveness of specific interfaces with the code reuse of generic implementations.

```
// Specific repository interface
public interface ICustomerRepository
{
    Task<Customer?> GetByIdAsync(CustomerId id,
CancellationToken cancellationToken = default);
    Task<Customer?> GetByEmailAsync(string email,
CancellationToken cancellationToken = default);
    Task<IReadOnlyList<Customer>>
GetInactiveCustomersAsync(TimeSpan inactivityPeriod,
CancellationToken cancellationToken = default);
    Task<IReadOnlyList<Customer>> GetByRegionAsync(string
region, CancellationToken cancellationToken = default);
    void Add(Customer customer);
    void Remove(Customer customer);
}

// Implementation can inherit common functionality
public class CustomerRepository : RepositoryBase<Customer>,
ICustomerRepository
{
    public CustomerRepository(ApplicationDbContext context)
: base(context) { }

    public async Task<Customer?> GetByEmailAsync(string
email, CancellationToken cancellationToken = default)
    {
        return await Context.Customers
            .FirstOrDefaultAsync(c => c.Email == email,
cancellationToken);
    }
    // Other specific methods...
}
```

A hybrid approach combines elements of both strategies. You might define a generic `IRepository<T>` interface for basic operations, then extend it with specific interfaces for domain operations. For example, `ICustomerRepository : IRepository<Customer>` inherits basic CRUD operations while adding customer-specific methods. This approach provides a consistent baseline of operations across all repositories while allowing domain-specific extensions. However, it adds complexity and can confuse developers about where to add new methods—in the generic base or the specific extension.

The recommendation for domain-centric architecture is to favor specific repositories. They better express your domain's ubiquitous language, allow for aggregate-specific optimizations, and maintain clear boundaries between aggregates. The small amount of interface duplication is a worthwhile trade-off for improved readability and maintainability. If you're concerned about implementation duplication, use base classes or composition in the infrastructure layer to share common code. Your domain layer benefits from clear, expressive interfaces, while your infrastructure layer remains DRY through implementation techniques.

Consider your team's experience level when choosing between generic and specific repositories. Teams new to domain-driven design often benefit from the explicit guidance that specific repositories provide. The interfaces clearly show what operations are available for each aggregate, serving as documentation and preventing inappropriate data access patterns. More experienced teams might successfully use generic repositories with discipline, but the risk of leaking abstractions or losing domain expressiveness remains. When in doubt, choose the approach that makes your domain model clearer and your code more maintainable.

9.4 UNIT OF WORK PATTERN

The Unit of Work pattern complements repositories by managing transactions and tracking changes to entities during a business operation. While repositories handle the retrieval and persistence of individual aggregates, the Unit of Work coordinates the saving of multiple aggregates within a single transaction. It maintains a list of objects affected by a business transaction and coordinates the writing of changes to the database. This coordination ensures that either all changes succeed together or all fail together, maintaining data consistency across aggregate boundaries.

Entity Framework Core's `DbContext` is itself an implementation of the Unit of Work pattern. It tracks changes to entities, manages database connections, and coordinates transaction commits. When you call `SaveChangesAsync()` on a `DbContext`, it generates and executes the necessary SQL statements to persist all tracked changes within a single transaction. This built-in functionality means you often don't need to create a separate Unit of Work abstraction—you can use `DbContext` directly in your infrastructure layer while keeping it hidden from your domain layer.

However, exposing `DbContext` directly to your application layer creates a dependency on Entity Framework, violating the Dependency Inversion Principle. Your application services would depend on a concrete infrastructure implementation rather than an abstraction. To maintain proper layering, you can create an `IUnitOfWork` interface in your domain layer that abstracts the transaction coordination responsibility. This interface typically includes methods like `SaveChangesAsync()` and potentially `BeginTransactionAsync()` for explicit transaction management.

```
public interface IUnitOfWork
{
    Task<int> SaveChangesAsync(CancellationToken
cancellationToken = default);
    Task BeginTransactionAsync(CancellationToken
cancellationToken = default);
    Task CommitTransactionAsync(CancellationToken
cancellationToken = default);
    Task RollbackTransactionAsync(CancellationToken
cancellationToken = default);
}
```

The implementation of `IUnitOfWork` in your infrastructure layer wraps your `DbContext` and delegates to its transaction management methods. This wrapper is thin—it simply forwards calls to the underlying `DbContext` without adding significant logic. The value lies in the abstraction it provides, allowing your application layer to coordinate transactions without knowing about Entity Framework. If you later switch to a different ORM or data access technology, you only need to change the infrastructure implementation, not your application services.

```
public class UnitOfWork : IUnitOfWork
{
    private readonly ApplicationDbContext _context;

    public UnitOfWork(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<int>
SaveChangesAsync(CancellationToken cancellationToken =
default)
    {
        return await
_context.SaveChangesAsync(cancellationToken);
    }

    public async Task
BeginTransactionAsync(CancellationToken cancellationToken =
default)
    {
        await
_context.Database.BeginTransactionAsync(cancellationToken);
    }

    public async Task
CommitTransactionAsync(CancellationToken cancellationToken
= default)
    {
        await
_context.Database.CommitTransactionAsync(cancellationToken)
;
    }

    public async Task
RollbackTransactionAsync(CancellationToken
cancellationToken = default)
    {
        await
_context.Database.RollbackTransactionAsync(cancellationToke
n);
    }
}
```

In your application services, you inject both repositories and the Unit of Work. The repositories handle loading and adding aggregates, while the Unit of Work handles saving changes. A typical use case loads one or more aggregates through repositories, invokes domain methods to modify them, potentially adds new aggregates, then calls `SaveChangesAsync()` on the Unit of Work to persist all changes atomically. This separation of concerns keeps your code organized and makes the transactional boundaries explicit.

The Unit of Work pattern becomes particularly valuable when a use case spans multiple aggregates. Consider a scenario where transferring inventory between warehouses requires updating both the source and destination warehouse aggregates. You load both aggregates through their repositories, invoke domain methods to transfer the inventory, then save both changes through the Unit of Work. The transaction ensures that inventory isn't lost if the operation fails partway through—either both warehouses update successfully, or neither does.

Some architectures question whether you need an explicit `IUnitOfWork` interface when using Entity Framework Core. An alternative approach has repositories accept a `DbContext` in their constructors, and application services inject the `DbContext` directly to call `SaveChangesAsync()`. This approach is simpler but couples your application layer to Entity Framework. The trade-off depends on how important you consider the abstraction. If you're confident you'll always use Entity Framework, the simpler approach might suffice. If you value the flexibility to change persistence technologies, the `IUnitOfWork` abstraction provides better isolation.

The Unit of Work pattern also provides a natural place to implement cross-cutting concerns related to persistence. You might override `SaveChangesAsync()` to automatically set audit fields like `ModifiedDate` and `ModifiedBy` on all changed entities. You could dispatch domain events after successfully saving changes, ensuring events only fire when the transaction commits. You might add logging to track how many entities were modified in each transaction. These concerns belong in the infrastructure layer's Unit of Work implementation, not scattered throughout your application services.

```
public class UnitOfWork : IUnitOfWork
{
    private readonly ApplicationDbContext _context;
    private readonly IDomainEventDispatcher
_eventDispatcher;

    public UnitOfWork(ApplicationDbContext context,
IDomainEventDispatcher eventDispatcher)
    {
        _context = context;
        _eventDispatcher = eventDispatcher;
    }

    public async Task<int>
SaveChangesAsync(CancellationToken cancellationToken =
default)
    {
        // Set audit fields
        SetAuditFields();

        // Save changes
        var result = await
_context.SaveChangesAsync(cancellationToken);

        // Dispatch domain events after successful save
        await
_eventDispatcher.DispatchEventsAsync(_context,
cancellationToken);

        return result;
    }
    private void SetAuditFields()
    {
        var entries = _context.ChangeTracker.Entries()
            .Where(e => e.State == EntityState.Added ||
e.State == EntityState.Modified);

        foreach (var entry in entries)
        {
            if (entry.Entity is IAuditable auditable)
            {
                if (entry.State == EntityState.Added)
                {
                    auditable.CreatedDate =
DateTime.UtcNow;
                }
                auditable.ModifiedDate = DateTime.UtcNow;
            }
        }
    }
}
```

When working with distributed transactions or multiple databases, the Unit of Work pattern becomes more complex. Entity Framework Core's `DbContext` manages transactions for a single database connection. If your use case needs to update multiple databases or coordinate with external services, you might need distributed transaction coordination through technologies like TransactionScope or saga patterns. These scenarios go beyond basic Unit of Work implementation and require careful consideration of consistency models and failure handling. In most applications, however, transactions within a single database suffice, and the standard Unit of Work pattern provides the coordination you need.

The relationship between repositories and Unit of Work should be clear in your architecture. Repositories don't save changes—they add entities to the tracked collection or mark them for removal. The Unit of Work handles the actual persistence. This separation means repository methods like `Add()` and `Remove()` are typically synchronous and fast, while `SaveChangesAsync()` is asynchronous and potentially slow. Your application services orchestrate the workflow: load aggregates, modify them, add or remove entities, then save changes. This explicit orchestration makes the transactional boundaries visible and easier to reason about.

10 Chapter 9: Infrastructure Layer Implementation

The infrastructure layer sits at the outermost ring of your domain-centric architecture. It contains all the technical details that your business logic shouldn't care about. This layer handles persistence, external APIs, file systems, email services, and any other technical concern that connects your application to the outside world. While the domain layer defines what data needs to be stored or retrieved, the infrastructure layer determines how that actually happens. This separation allows you to swap out databases, change email providers, or modify logging frameworks without touching your core business logic.

Understanding the infrastructure layer's role is crucial for maintaining the dependency rule. Your domain and application layers define interfaces that describe what they need from external systems. The infrastructure layer implements these interfaces, providing concrete implementations that handle the messy details of working with real-world technologies. This inversion of dependencies means your business logic remains pure and testable, while the infrastructure layer adapts external systems to meet your domain's needs. The infrastructure layer depends on your domain, never the other way around.

In this chapter, you'll learn how to design and implement an effective infrastructure layer. You'll discover what responsibilities belong here, how to integrate external services without coupling them to your domain, and how to manage configuration properly. You'll see concrete examples of implementing repository patterns with Entity Framework, integrating third-party APIs, and handling cross-cutting concerns like logging and caching. By the end of this chapter, you'll understand how to build an infrastructure layer that serves your domain without constraining it.

10.1 Infrastructure Layer Responsibilities

The infrastructure layer handles all technical implementation details that support your application but aren't part of your business logic. This includes data persistence, external service integration, file system access, email delivery, and logging. These concerns are essential for your application to function, but they represent technical decisions rather than business rules. By isolating them in the infrastructure layer, you protect your domain from being polluted by technical details that might change as technology evolves or business needs shift.

Data persistence represents one of the infrastructure layer's primary responsibilities. This includes implementing repository interfaces defined in your domain layer, configuring database contexts, managing migrations, and handling connection strings. Your infrastructure layer translates between your domain's aggregate roots and the database's relational structure. For example, if your domain defines an `Order` aggregate with `OrderItem` entities, your infrastructure layer maps these to database tables and handles the complexity of loading and saving them together as a single unit.

External service integration also belongs in the infrastructure layer. When your application needs to send emails, process payments, or call third-party APIs, the infrastructure layer provides implementations of interfaces defined by your application layer. Consider a payment processing scenario:

- **Application Layer:** Defines `IPaymentProcessor` interface with a `ProcessPayment` method
- **Infrastructure Layer:** Implements `StripePaymentProcessor` that calls Stripe's API
- **Domain Layer:** Remains completely unaware of Stripe or any payment provider

Cross-cutting concerns like logging, caching, and monitoring live in the infrastructure layer. These concerns affect multiple parts of your application but aren't business logic themselves. Your infrastructure layer might implement decorators that add logging to repository calls, or provide caching implementations that wrap your data access. For instance, you might create a `CachedProductRepository` that decorates your standard `ProductRepository`, checking a cache before hitting the database. This keeps caching logic separate from both your domain and your core data access implementation.

File system operations and external resource management represent another infrastructure responsibility. Whether you're storing uploaded files locally, in Azure Blob Storage, or Amazon S3, these details belong in the infrastructure layer. Your application layer might define an `IFileStorage` interface with methods like `SaveFile` and `GetFile`, while your infrastructure layer provides concrete implementations for different storage providers. This abstraction allows you to start with local file storage during development and switch to cloud storage in production without changing your application code.

Configuration management and dependency registration also fall under infrastructure responsibilities. Your infrastructure layer typically contains the composition root where you configure your dependency injection container. This is where you register all your services, repositories, and external integrations. In an ASP.NET Core application, this happens in your `Program.cs` or startup configuration. Here's what this registration might look like:

- Register `ApplicationDbContext` with connection string from configuration
- Register repository implementations for their interfaces
- Configure external service clients with API keys and endpoints
- Set up logging providers and their configurations
- Register application services and their dependencies

Message queue integration and event publishing mechanisms belong in the infrastructure layer when you're building distributed systems. If your application publishes domain events to RabbitMQ, Azure Service Bus, or Kafka, the infrastructure layer handles the technical details of connecting to these systems and serializing messages. Your domain layer raises events like `OrderPlacedEvent`, your application layer decides when to publish them, and your infrastructure layer handles the actual mechanics of sending them to your message broker. This separation allows you to change messaging technologies without affecting your business logic.

Security infrastructure, including authentication and authorization implementations, resides in this layer. While your application layer might define authorization policies and your domain enforces business rules, the infrastructure layer handles the technical details of verifying JWT tokens, checking user credentials against a database, or integrating with external identity providers like Azure AD or Auth0. Your infrastructure layer translates between the security tokens and claims provided by these systems and the user identity concepts your application layer understands.

10.2 Integrating External Services

External service integration requires careful design to prevent third-party dependencies from leaking into your domain. The key principle is to define interfaces in your application layer that describe what your application needs, then implement those interfaces in your infrastructure layer using whatever external services you choose. This approach keeps your business logic independent of external vendors and makes it possible to swap services without rewriting your application. Your domain should never directly reference NuGet packages for external APIs or services.

Consider integrating an email service into your application. Your application layer defines an interface that captures your email needs without specifying how emails are sent:

```
public interface IEmailService
{
    Task SendOrderConfirmationAsync(string recipientEmail,
Order order);
    Task SendPasswordResetAsync(string recipientEmail,
string resetToken);
    Task SendWelcomeEmailAsync(string recipientEmail,
string userName);
}
```

Your infrastructure layer then implements this interface using your chosen email provider. Here's an example using SendGrid:

```
public class SendGridEmailService : IEmailService
{
    private readonly ISendGridClient _client;
    private readonly EmailSettings _settings;

    public SendGridEmailService(ISendGridClient client,
IOptions<EmailSettings> settings)
    {
        _client = client;
        _settings = settings.Value;
    }

    public async Task SendOrderConfirmationAsync(string
recipientEmail, Order order)
    {
        var message = new SendGridMessage
        {
            From = new EmailAddress(_settings.FromAddress,
_settings.FromName),
            Subject = $"Order Confirmation -
{order.OrderNumber}",
            HtmlContent = BuildOrderConfirmationHtml(order)
        };
        message.AddTo(recipientEmail);

        await _client.SendEmailAsync(message);
    }

    // Other methods implemented similarly
}
```

`MailgunEmailService`

Payment processing integration follows the same pattern. Your application layer defines what payment operations your business needs, while the infrastructure layer handles the complexity of working with payment gateways. Your interface might look like this:

```
public interface IPaymentProcessor
{
    Task<PaymentResult> ProcessPaymentAsync(PaymentRequest
request);
    Task<RefundResult> RefundPaymentAsync(string
transactionId, decimal amount);
    Task<PaymentStatus> GetPaymentStatusAsync(string
transactionId);
}
```

`PaymentRequestPaymentResult`

When implementing external service integrations, you often need to translate between your domain models and the external service's data structures. This translation happens in the infrastructure layer through adapter or mapper classes. For example, when integrating with a shipping API, you might have:

- **Domain Model:**`ShippingAddress` with properties your business understands
- **External API Model:**`FedExAddress` with properties FedEx requires
- **Infrastructure Mapper:** Converts between these two representations

Error handling for external services requires special attention in the infrastructure layer. External services can fail in ways your domain shouldn't need to understand—network timeouts, rate limiting, temporary outages, or API changes. Your infrastructure implementations should catch these technical exceptions and translate them into domain-meaningful results or exceptions. For instance, if a payment API returns a 503 Service Unavailable error, your infrastructure layer might throw a `PaymentServiceUnavailableException` that your application layer can handle appropriately, perhaps by queuing the payment for retry.

Resilience patterns like retry logic, circuit breakers, and timeouts belong in the infrastructure layer. Using libraries like Polly, you can add these patterns to your external service integrations without cluttering your application logic:

```
public class ResilientPaymentProcessor : IPaymentProcessor
{
    private readonly IPaymentProcessor _innerProcessor;
    private readonly IAsyncPolicy _retryPolicy;

    public ResilientPaymentProcessor(IPaymentProcessor
innerProcessor)
    {
        _innerProcessor = innerProcessor;
        _retryPolicy = Policy
            .Handle<HttpRequestException>()
            .WaitAndRetryAsync(3, retryAttempt =>
                TimeSpan.FromSeconds(Math.Pow(2,
retryAttempt)));
    }

    public async Task<PaymentResult>
ProcessPaymentAsync(PaymentRequest request)
    {
        return await _retryPolicy.ExecuteAsync(() =>
            _innerProcessor.ProcessPaymentAsync(request));
    }
}
```

Testing external service integrations becomes much easier with this architecture. Your application layer tests can use mock implementations of your interfaces, while your infrastructure layer tests verify that your implementations correctly call external APIs. You might create a `FakeEmailService` for testing that captures sent emails in memory, allowing you to verify that your application sends the right emails at the right times without actually sending them. For integration testing, you can use the real implementations against test environments or sandbox APIs provided by external services.

10.3 Configuration and Settings Management

Configuration management in the infrastructure layer involves organizing application settings, connection strings, API keys, and environment-specific values in a way that keeps them separate from your code while making them easily accessible where needed. Modern C# applications typically use the configuration system built into ASP.NET Core, which supports multiple configuration sources including JSON files, environment variables, user secrets, and Azure Key Vault. Your infrastructure layer is responsible for reading these configurations and providing them to the services that need them.

The strongly-typed configuration pattern using the Options pattern provides type-safe access to configuration values. Instead of reading configuration strings directly throughout your code, you define classes that represent configuration sections and bind them during startup. Here's how you might structure email configuration:

```
public class EmailSettings
{
    public string ApiKey { get; set; }
    public string FromAddress { get; set; }
    public string FromName { get; set; }
    public int MaxRetries { get; set; }
    public int TimeoutSeconds { get; set; }
}

// In your appsettings.json
{
    "EmailSettings": {
        "ApiKey": "your-api-key",
        "FromAddress": "noreply@example.com",
        "FromName": "Example Store",
        "MaxRetries": 3,
        "TimeoutSeconds": 30
    }
}
```

`IOptions<EmailSettings>`

Organizing configuration by feature or service keeps your settings manageable as your application grows. Rather than having one massive configuration file, you can split settings into logical sections that correspond to different infrastructure concerns. Your configuration structure might look like:

- **ConnectionStrings:** Database connection strings for different environments
- **EmailSettings:** Email service configuration and credentials
- **PaymentSettings:** Payment gateway endpoints and API keys

- **CacheSettings:** Redis connection strings and cache expiration policies
- **LoggingSettings:** Log levels and output destinations

Environment-specific configuration allows your application to behave differently in development, staging, and production without code changes. ASP.NET Core supports this through multiple `appsettings.json` files like `appsettings.Development.json` and `appsettings.Production.json`. Your infrastructure layer reads the appropriate file based on the environment. For example, you might use a local SQL Server database in development but Azure SQL Database in production, configured through environment-specific connection strings. This approach eliminates the need to modify configuration files when deploying to different environments.

Sensitive configuration values like API keys, database passwords, and encryption keys should never be stored in source control. For local development, use the Secret Manager tool (user secrets) to store sensitive values outside your project directory. In production, use environment variables or secure key management services like Azure Key Vault or AWS Secrets Manager. Your infrastructure layer configuration setup might look like this:

```
public static IHostBuilder CreateHostBuilder(string[] args)
=>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            if (context.HostingEnvironment.IsProduction())
            {
                var builtConfig = config.Build();
                var keyVaultEndpoint =
builtConfig["KeyVaultEndpoint"];
                config.AddAzureKeyVault(new
Uri(keyVaultEndpoint),
                    new DefaultAzureCredential());
            }
        });
```

Configuration validation ensures your application fails fast with clear error messages if required configuration is missing or invalid. You can add validation to your configuration classes using data annotations:

```
public class DatabaseSettings
{
    [Required]
    public string ConnectionString { get; set; }

    [Range(1, 100)]
    public int MaxRetryCount { get; set; }

    [Range(1, 300)]
    public int CommandTimeoutSeconds { get; set; }
}

// Register with validation
services.AddOptions<DatabaseSettings>()
    .Bind(configuration.GetSection("DatabaseSettings"))
    .ValidateDataAnnotations()
    .ValidateOnStart();
```

`ValidateOnStart()`

Feature flags and toggles can be managed through configuration to enable or disable functionality without deploying new code. Your infrastructure layer might read feature flag settings from configuration and provide them through a feature management service. This allows you to gradually roll out new features, perform A/B testing, or quickly disable problematic features in production. Libraries like Microsoft.FeatureManagement integrate with the configuration system to provide this capability:

```
// In appsettings.json
{
    "FeatureManagement": {
        "NewCheckoutFlow": true,
        "RecommendationEngine": false
    }
}

// In your service
public class OrderService
{
    private readonly IFeatureManager _featureManager;

    public async Task<CheckoutResult> CheckoutAsync(Cart
cart)
    {
        if (await
_featureManager.IsEnabledAsync("NewCheckoutFlow"))
        {
            return await
_newCheckoutService.ProcessAsync(cart);
        }
        return await
_legacyCheckoutService.ProcessAsync(cart);
    }
}
```

Configuration change detection and hot reload capabilities allow your application to respond to configuration changes without restarting. The ASP.NET Core configuration system supports this through `IOptionsSnapshot<T>` and `IOptionsMonitor<T>`. Use

`IOptionsSnapshot` when you want configuration to be consistent for the duration of a request, and `IOptionsMonitor` when you need to react to configuration changes immediately. This is particularly useful for settings like feature flags or cache expiration times that you might want to adjust without redeploying your application.

Dependency injection registration based on configuration allows you to swap implementations based on settings. For example, you might use different cache providers in different environments:

```
var cacheSettings =
configuration.GetSection("CacheSettings").Get<CacheSettings
>();

if (cacheSettings.Provider == "Redis")
{
    services.AddStackExchangeRedisCache(options =>
    {
        options.Configuration =
cacheSettings.ConnectionString;
    });
}
else
{
    services.AddDistributedMemoryCache();
}
```

11 Chapter 10: Entity Framework in Clean Architecture

Entity Framework Core represents one of the most popular object-relational mapping frameworks in the .NET ecosystem, yet its integration into domain-centric architectures often raises questions about proper boundaries and responsibilities. Many developers struggle with the tension between EF Core's conventions and the strict separation of concerns that Clean Architecture demands. The framework's powerful features, from change tracking to lazy loading, can easily blur the lines between your domain logic and persistence concerns if not carefully managed.

The key to successfully using Entity Framework Core in a domain-centric architecture lies in treating it as an implementation detail confined entirely to the infrastructure layer. Your domain entities should remain pure, focused solely on business logic and invariants, without any awareness of how they'll be persisted. EF Core becomes a tool that translates between your rich domain model and the relational database structure, never dictating how your domain should be designed. This approach requires deliberate configuration and mapping strategies that respect aggregate boundaries while leveraging EF Core's capabilities.

Throughout this chapter, you'll learn practical techniques for integrating Entity Framework Core without compromising architectural principles. You'll discover how to configure DbContext instances that serve infrastructure needs without leaking into your domain, map complex aggregates to database schemas while maintaining consistency boundaries, and implement repositories that provide clean abstractions over EF Core's data access patterns. These patterns enable you to benefit from EF Core's productivity features while keeping your domain layer pristine and testable. The goal is achieving a balance where persistence concerns remain invisible to your business logic, yet database operations remain efficient and maintainable.

11.1 Setting Up Entity Framework Core

Establishing Entity Framework Core within a clean architecture requires careful consideration of project structure and dependency flow. The EF Core packages should be installed exclusively in your infrastructure project, never in the domain or application layers. This physical separation enforces the architectural boundary, ensuring that your core business logic cannot accidentally depend on persistence implementation details. Your infrastructure project will reference the domain project to access entity definitions, but this dependency flows in the correct direction—inward toward the domain core.

Begin by installing the necessary NuGet packages in your infrastructure project. You'll need `Microsoft.EntityFrameworkCore` as the core package, along with a database provider such as `Microsoft.EntityFrameworkCore.SqlServer` for SQL Server or `Microsoft.EntityFrameworkCore.Sqlite` for SQLite during development. Additionally, install `Microsoft.EntityFrameworkCore.Design` to enable migration tooling. These packages provide the foundation for data access without polluting your domain layer with persistence concerns:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package
Microsoft.EntityFrameworkCore.SqlServer
dotnet add package
Microsoft.EntityFrameworkCore.Design
```

The infrastructure project structure should organize EF Core components logically, separating configuration from implementation. Create a `Persistence` folder within your infrastructure project to house all database-related code. Within this folder, establish subdirectories for

`Configurations` (entity type configurations), `Repositories` (repository implementations), and `Migrations` (database migration files). This organization makes it immediately clear which code handles persistence concerns and keeps related components together for easier maintenance.

Your application layer defines repository interfaces that express domain needs without any EF Core dependencies. These interfaces live in the application layer because they represent use case requirements, not implementation details. The infrastructure layer then provides concrete implementations using EF Core. This arrangement respects the Dependency Inversion Principle, allowing your application layer to depend on abstractions while the infrastructure layer depends on both the abstractions and the concrete EF Core framework:

Connection string management belongs in the infrastructure layer's configuration, typically injected through dependency injection during application startup. Avoid hardcoding connection strings or placing them in domain or application projects. Instead, use the options pattern to configure your DbContext, allowing different connection strings for development, testing, and production environments. This approach maintains flexibility while keeping configuration concerns separate from business logic. The presentation layer (such as a Blazor application or API) registers the DbContext with the dependency injection container, specifying the connection string from configuration files.

Consider creating a separate class library project specifically for database migrations if your solution grows large. This separation allows you to manage schema changes independently from runtime infrastructure code. The migrations project would reference your infrastructure project to access DbContext and entity configurations, but it exists solely to generate and apply database schema updates. This pattern proves particularly valuable in team environments where database administrators might need to review and apply migrations separately from application deployments.

Testing infrastructure requires special consideration during setup. Create a separate test project that references your infrastructure layer and uses an in-memory database provider or a test database instance. EF Core's `Microsoft.EntityFrameworkCore.InMemory` provider works well for fast unit tests, though it doesn't perfectly replicate relational database behavior. For integration tests that verify actual database interactions, consider using SQLite in-memory mode or Docker containers running your production database engine. This testing strategy ensures your EF Core configurations and mappings work correctly without requiring a persistent database during development.

Version compatibility between EF Core and your target .NET version matters significantly for long-term maintainability. EF Core 6 requires .NET 6 or later, while EF Core 7 and 8 bring additional features and performance improvements. Choose a version that aligns with your project's .NET runtime and stick with it consistently across all projects in your solution. Mixing EF Core versions can lead to subtle bugs and compatibility issues. Document your chosen version in a solution-level README file to guide future developers and ensure consistent package updates across the team.

11.2 DbContext Design and Configuration

The DbContext class serves as your application's gateway to the database, coordinating entity tracking, change detection, and query execution. In a clean architecture, your DbContext implementation resides entirely within the infrastructure layer, configured to map domain entities to database tables without requiring those entities to know about persistence. Design your DbContext as a focused component responsible solely for database operations, avoiding the temptation to add business logic or validation rules that belong in the domain layer. A well-designed

DbContext remains thin, delegating domain concerns to aggregates and entities.

Create a dedicated DbContext class that inherits from `DbContext` and exposes DbSet properties for your aggregate roots only, not for every entity in your domain. Remember that aggregates define consistency boundaries, and you should only query and persist complete aggregates, never their internal entities independently. This approach enforces aggregate boundaries at the persistence level:

```
public class ApplicationDbContext : DbContext
{
    public DbSet<Order> Orders { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Product> Products { get; set; }

    public
ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {

modelBuilder.ApplyConfigurationsFromAssembly(typeof(Applica
tionDbContext).Assembly);
        base.OnModelCreating(modelBuilder);
    }
}
```

The `OnModelCreating` method provides the central location for applying entity configurations. Rather than cluttering this method with inline configuration code, use the `ApplyConfigurationsFromAssembly` method to automatically discover and apply all `IEntityTypeConfiguration<T>` implementations in your infrastructure assembly. This pattern keeps your DbContext clean while organizing entity-specific configuration into separate, focused classes. Each entity gets its own configuration class, making configurations easier to locate, understand, and modify without affecting other entities.

Configure your DbContext to disable certain EF Core features that conflict with domain-centric principles. Lazy loading, while convenient, can lead to unintentional database queries scattered throughout your codebase and makes it difficult to reason about when data access occurs. Disable lazy loading by not installing the proxies package and not configuring navigation properties as virtual. Similarly, consider disabling automatic change tracking for read-only queries using `AsNoTracking()`, improving performance when you're simply retrieving data without intending to modify it:

- **Disable lazy loading:** Avoid the `Microsoft.EntityFrameworkCore.Proxies` package and don't mark navigation properties as virtual
- **Use explicit loading:** Load related entities intentionally using `Include()` or explicit `Load()` calls
- **Apply AsNoTracking:** Use `.AsNoTracking()` for queries that retrieve data without modification intent
- **Configure query splitting:** Use `AsSplitQuery()` for complex includes to avoid cartesian explosion

Connection resilience and retry logic belong in DbContext configuration, particularly for cloud-based databases that may experience transient failures. EF Core provides built-in retry strategies that automatically handle temporary connection issues without requiring application-level retry logic. Configure these strategies in your DbContext options during service registration, specifying maximum retry counts and delay intervals appropriate for your environment. This infrastructure-level concern remains invisible to your domain and application layers, which simply execute operations without worrying about transient failures.

Consider implementing a base DbContext class if you have multiple bounded contexts in your application, each with its own database. The base class can contain common configuration logic, such as audit field handling, soft delete filters, or value object conversions. Each bounded context then inherits from this base class and adds its specific entity configurations. This pattern reduces duplication while maintaining clear boundaries between different parts of your domain:

Seeding initial data through DbContext configuration provides a convenient way to populate lookup tables or test data during development. Use the `HasData` method within entity configurations to specify seed data that EF Core will include in migrations. However, be cautious with this approach in production environments—seed data in migrations can cause issues during updates if the data has been modified. For production data seeding, consider creating separate initialization services that run after migrations, checking for existing data before inserting records.

Query filters configured at the DbContext level enable global filtering logic that applies automatically to all queries. This feature proves particularly useful for implementing soft deletes or multi-tenant data isolation. Define query filters in your entity configurations using `HasQueryFilter`, and EF Core will automatically append the filter conditions to every query for that entity type. You can temporarily disable filters for specific queries using `IgnoreQueryFilters()` when needed, such as when administrators need to view deleted records.

Performance monitoring and logging integration should be configured at the DbContext level to provide visibility into database operations without instrumenting your domain code. EF Core's logging integration with Microsoft.Extensions.Logging allows you to capture SQL queries, execution times, and errors. Configure logging levels appropriately—verbose logging helps during development but should be reduced in production to avoid performance overhead. Consider using Application Insights or similar monitoring tools to track query performance and identify optimization opportunities in production environments.

11.3 Mapping Domain Aggregates to Database

Mapping domain aggregates to relational database structures requires careful attention to aggregate boundaries and consistency requirements. Your domain model prioritizes business rules and invariants, while database schemas optimize for storage and query efficiency. These different concerns often lead to impedance mismatch—the domain model's structure doesn't naturally align with an ideal database schema. Entity Framework Core's Fluent API provides powerful mapping capabilities that bridge this gap, allowing you to maintain a rich domain model while achieving efficient database storage.

Create separate configuration classes for each aggregate root using the `IEntityTypeConfiguration<T>` interface. These configuration classes live in your infrastructure layer's Configurations folder and contain all mapping logic for an aggregate and its internal entities. This approach keeps configuration organized and separates persistence concerns from domain logic. Each configuration class focuses on a single aggregate, making it easy to understand how that aggregate maps to database tables:

```
public class OrderConfiguration :
IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> builder)
    {
        builder.ToTable("Orders");

        builder.HasKey(o => o.Id);

        builder.Property(o => o.Id)
            .HasConversion(
                id => id.Value,
                value => new OrderId(value));

        builder.Property(o => o.OrderNumber)
            .HasMaxLength(50)
            .IsRequired();

        builder.OwnsMany(o => o.LineItems, li =>
        {
            li.ToTable("OrderLineItems");
            li.WithOwner().HasForeignKey("OrderId");
            li.Property<int>("Id");
            li.HasKey("Id");
        });
    }
}
```

Value objects require special mapping consideration since they represent concepts without identity. EF Core's owned entity types feature provides an elegant solution for mapping value objects. Use `OwnsOne` for singlevalued properties like Address or Money, and `OwnsMany` for collections of value objects. Owned entities can be stored in the same table as their owner (table splitting) or in separate tables, depending on your schema preferences. The key distinction is that owned entities cannot be queried independently—they're always loaded as part of their owning aggregate.

Strongly-typed identifiers, a common pattern in domain-driven design, need explicit conversion configuration. Rather than exposing primitive types like `int` or `Guid` directly, domain entities often wrap identifiers in value objects like `OrderId` or `CustomerId`. Configure value conversions using `HasConversion` to translate between your domain's strongly-typed IDs and the database's primitive types. This mapping happens transparently—your domain code works with type-safe identifiers while the database stores efficient primitives:

- **Define the conversion:** Specify how to convert from domain type to database type and back
- **Apply to properties:** Use `HasConversion` on ID properties in entity configurations
- **Maintain type safety:** Your domain code never sees primitive IDs, only strongly-typed wrappers
- **Enable comparisons:** Implement equality members on ID types for proper EF Core tracking

Aggregate boundaries must be respected in your mapping configuration by carefully controlling navigation properties and foreign keys. Internal entities within an aggregate should not be directly accessible from outside the aggregate root. Configure relationships so that EF Core loads entire aggregates together, never allowing partial aggregate loading. Use private setters or no setters on collection navigation properties, forcing all modifications to go through aggregate root methods that enforce invariants. This configuration ensures that persistence operations respect the consistency boundaries you've defined in your domain model.

Handling domain events during persistence requires coordination between your domain model and EF Core's change tracking. Domain entities raise events when significant business actions occur, but these events should be dispatched after successful persistence to ensure consistency. One effective pattern involves collecting domain events from tracked entities before calling `SaveChanges`, then dispatching them afterward. Override `SaveChangesAsync` in your DbContext to implement this pattern:

Complex value objects containing multiple properties benefit from table splitting or separate tables depending on their usage patterns. An Address value object with street, city, state, and postal code properties could be stored in the same table as its owning entity using table splitting, or in a separate table if addresses are large or optional. Consider query patterns when making this decision—if you frequently query entities without needing their value objects, separate tables with explicit loading might perform better. Conversely, if value objects are almost always needed, table splitting reduces joins and improves query performance.

Enumeration types in your domain model should be mapped thoughtfully, choosing between string and integer storage based on your needs. String storage makes database queries more readable and protects against issues when enum values are reordered, but consumes more space. Integer storage is more efficient but requires careful management when enum definitions change. EF Core's `HasConversion` supports both approaches, and you can even create custom conversions for smart enumerations—classes that behave like enums but contain additional behavior and properties.

Inheritance hierarchies in your domain model can be mapped using table-per-hierarchy, table-per-type, or table-per-concrete-type strategies. Table-per-hierarchy stores all types in a single table with a discriminator column, offering the best query performance but potentially wasting space on nullable columns. Table-per-type creates separate tables for each type in the hierarchy, normalizing storage but requiring joins for queries. Choose the strategy that best fits your domain model's inheritance patterns and query requirements. Configure the chosen strategy using `HasDiscriminator` for table-per-hierarchy or `ToTable` for other strategies in your entity configurations.

11.4 Implementing Repositories with EF Core

Repository implementations using Entity Framework Core bridge the gap between your domain's data access abstractions and EF Core's concrete data access mechanisms. These implementations live in the infrastructure layer, accepting a DbContext through constructor injection and using it to fulfill the contracts defined by repository interfaces. The repository pattern provides several benefits in this context: it abstracts EF Core details from your application layer, provides a natural place for query logic that doesn't belong in the domain, and creates a testable seam where you can substitute in-memory implementations during testing.

A typical repository implementation accepts the DbContext as a constructor parameter and exposes methods that correspond to the repository interface. Each method translates domain-oriented operations into EF Core queries or commands. Keep repository methods focused on data access concerns—filtering, sorting, paging, and including related entities—while leaving business logic in the domain layer. Your repository should never validate business rules or enforce invariants; those responsibilities belong to aggregates:

```
public class OrderRepository : IOrderRepository
{
    private readonly ApplicationDbContext _context;

    public OrderRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<Order?> GetByIdAsync(OrderId id,
CancellationToken cancellationToken = default)
    {
        return await _context.Orders
            .Include(o => o.LineItems)
            .FirstOrDefaultAsync(o => o.Id == id,
cancellationToken);
    }

    public async Task<IReadOnlyList<Order>>
GetByCustomerIdAsync(
        CustomerId customerId,
        CancellationToken cancellationToken = default)
    {
        return await _context.Orders
            .Include(o => o.LineItems)
            .Where(o => o.CustomerId == customerId)
            .OrderByDescending(o => o.OrderDate)
            .ToListAsync(cancellationToken);
    }

    public async Task AddAsync(Order order,
CancellationToken cancellationToken = default)
    {
        await _context.Orders.AddAsync(order,
cancellationToken);
    }

    public void Update(Order order)
    {
        _context.Orders.Update(order);
    }

    public void Remove(Order order)
    {
        _context.Orders.Remove(order);
    }
}
```

Query methods in repositories should eagerly load all parts of an aggregate using `Include` statements. Since aggregates define consistency boundaries, you should always load complete aggregates, never partial ones. This approach prevents lazy loading issues and makes data access patterns explicit. If an aggregate contains multiple levels of owned entities or related entities, include all of them in your query. The performance cost of loading complete aggregates is typically acceptable and preferable to the complexity and bugs that arise from partial aggregate loading.

The Unit of Work pattern is implicitly implemented by EF Core's DbContext, which tracks changes and coordinates persistence operations. You don't need to create a separate Unit of Work class in most cases—the DbContext already serves this purpose. However, you do need to explicitly call `SaveChangesAsync` to persist changes to the database. This call typically happens in your application layer's use case handlers, after all domain operations complete successfully. Placing `SaveChangesAsync` in the application layer rather than repository methods gives you transaction control and allows multiple repository operations to participate in a single transaction.

Specification pattern implementation can enhance repository flexibility when you need complex, reusable query logic. Rather than creating numerous repository methods for every possible query combination, define specification classes that encapsulate query criteria. Your repository accepts specifications and applies them to queries. This pattern proves particularly valuable for complex filtering scenarios where business rules determine which records to retrieve:

Handling optimistic concurrency in repositories requires coordination with EF Core's concurrency tokens. Configure a concurrency token property (such as a RowVersion or timestamp column) in your entity configuration, and EF Core will automatically check it during updates. When a concurrency conflict occurs, EF Core throws a `DbUpdateConcurrencyException`. Your repository or application layer should catch this exception and handle it appropriately—either by retrying the operation with fresh data or by notifying the user of the conflict. This mechanism prevents lost updates when multiple users modify the same aggregate simultaneously.

Read-only queries that don't need change tracking should use `AsNoTracking()` to improve performance. When retrieving data for display purposes without intending to modify it, disabling change tracking reduces memory overhead and speeds up query execution. Create separate query methods or even separate query repositories specifically for read operations, applying `AsNoTracking()` consistently. This separation aligns well with Command Query Responsibility Segregation

(CQRS) principles, where commands and queries follow different paths through your architecture.

Pagination support in repositories requires careful implementation to avoid performance issues with large datasets. Accept skip and take parameters in your repository methods, and apply them using `Skip()` and `Take()` after filtering and sorting. Always include sorting in paginated queries—without explicit ordering, pagination results become unpredictable. Consider returning a wrapper object that includes both the page of results and total count information, enabling proper pagination UI in your presentation layer:

- **Accept pagination parameters:** Include skip, take, and sorting parameters in repository methods
- **Apply filtering first:** Filter data before applying pagination to get accurate counts
- **Include total count:** Execute a separate count query to determine total records
- **Return structured results:** Use a result object containing items and total count

Bulk operations like deleting or updating multiple entities require special consideration for performance. EF Core's standard change tracking approach works well for small numbers of entities but becomes inefficient for bulk operations. For large-scale updates or deletes, consider using EF Core's `ExecuteUpdateAsync` and `ExecuteDeleteAsync` methods, which generate direct SQL commands without loading entities into memory. These methods bypass change tracking and domain logic, so use them carefully and only when performance requirements justify skipping domain validation.

Testing repository implementations requires either integration tests against a real database or unit tests with mocked DbContext. Integration tests provide higher confidence that your mappings and queries work correctly, but they're slower and require database setup. Use SQLite in-memory databases or Docker containers for integration tests to avoid dependencies on shared database instances. For unit tests, consider whether mocking DbContext provides sufficient value—the complexity of mocking EF Core's query infrastructure often outweighs the benefits. Focus integration tests on repositories while unit testing domain logic and application services separately.

12 CHAPTER 11: DEPENDENCY INJECTION AND IOC

Dependency Injection represents one of the most powerful techniques for achieving loose coupling in domain-centric architectures. When you build applications that follow the Dependency Rule, you need a mechanism to wire together components from different layers without violating architectural boundaries. Dependency Injection provides exactly this capability by inverting the traditional flow of control, allowing high-level modules to remain independent of low-level implementation details. This inversion enables you to swap implementations, test components in isolation, and maintain clean separation between layers.

The relationship between Dependency Injection and the Dependency Inversion Principle creates the foundation for maintainable architecture. While the Dependency Inversion Principle states that abstractions should not depend on details, Dependency Injection provides the practical mechanism for achieving this goal. Your domain layer defines interfaces for the services it needs, and the infrastructure layer provides concrete implementations. Dependency Injection frameworks handle the complex task of instantiating these implementations and injecting them where needed, all without requiring your domain code to know anything about the concrete types.

Understanding Dependency Injection goes beyond simply using a framework or container. You need to grasp the underlying principles of object composition, lifetime management, and the composition root pattern. These concepts determine how your application assembles its components at runtime, how long those components live, and where the responsibility for wiring dependencies belongs. Mastering these patterns transforms how you design classes, structure your solutions, and think about component relationships. This chapter explores these concepts in depth, providing you with the knowledge to implement Dependency Injection effectively in your domain-centric C# applications.

12.1 Dependency Injection Fundamentals

Dependency Injection is a design pattern where objects receive their dependencies from external sources rather than creating them internally. When a class needs to use another class or service, instead of instantiating that dependency using the `new` keyword, it receives the dependency through its constructor, a property, or a method parameter. This simple shift in responsibility has profound implications for your code's flexibility, testability, and maintainability. The class no longer controls which implementation it receives, making it easier to substitute different implementations for different contexts.

Consider a traditional approach where a class creates its own dependencies. An `OrderService` might instantiate a `SqlOrderRepository` directly within its constructor or methods. This creates tight coupling between the service and the specific repository implementation. Testing becomes difficult because you cannot easily substitute a test double for the real database repository. The service also violates the Single Responsibility Principle by taking on the additional responsibility of knowing how to construct its dependencies. Every time you want to change the repository implementation, you must modify the service class itself.

Dependency Injection solves these problems by inverting the dependency relationship. The `OrderService` declares that it needs an `IOrderRepository` through its constructor parameter, but it does not specify which concrete implementation. Some external component, typically a Dependency Injection container, decides which implementation to provide and handles the instantiation. This separation of concerns means your service focuses solely on its business logic while remaining completely ignorant of infrastructure details. The service becomes more flexible, more testable, and more aligned with SOLID principles.

Three primary injection methods exist in C#: constructor injection, property injection, and method injection. Constructor injection is the most common and preferred approach because it makes dependencies explicit and ensures that a class cannot be instantiated without its required dependencies. Property injection allows optional dependencies to be set after construction, useful for dependencies that have sensible defaults. Method injection passes dependencies as parameters to specific methods, appropriate when a dependency is only needed for that particular operation. Each method has its place, but constructor injection should be your default choice for required dependencies.

```
public class OrderService
{
    private readonly IOrderRepository _orderRepository;
    private readonly IEmailService _emailService;

    public OrderService(
        IOrderRepository orderRepository,
        IEmailService emailService)
    {
        _orderRepository = orderRepository
            ?? throw new
ArgumentNullException(nameof(orderRepository));
        _emailService = emailService
            ?? throw new
ArgumentNullException(nameof(emailService));
    }

    public async Task<Order>
CreateOrderAsync(CreateOrderCommand command)
    {
        var order = new Order(command.CustomerId,
command.Items);
        await _orderRepository.AddAsync(order);
        await
_emailService.SendOrderConfirmationAsync(order);
        return order;
    }
}
```

The benefits of Dependency Injection extend throughout your application architecture. Testing becomes straightforward because you can inject mock implementations that verify behavior without touching external systems. Your code becomes more modular because components depend on abstractions rather than concrete types. Configuration changes become simpler because you can swap implementations by changing registration code rather than modifying business logic. The pattern also supports the Open-Closed Principle by allowing you to extend behavior through new implementations without modifying existing code.

Dependency Injection does introduce some complexity, particularly around understanding how objects are composed and managing object lifetimes. You need to think carefully about which dependencies are required versus optional, how long instances should live, and where the composition logic belongs. However, these considerations lead to better design decisions and more maintainable code. The initial investment in understanding Dependency Injection pays dividends throughout your application's lifetime, making it easier to adapt to changing requirements and maintain clean architectural boundaries.

In domain-centric architectures, Dependency Injection becomes essential for maintaining the Dependency Rule. Your domain layer defines interfaces for repositories, external services, and infrastructure concerns. The infrastructure layer provides concrete implementations of these interfaces. Dependency Injection allows the application layer to use domain interfaces while the composition root wires up infrastructure implementations. This arrangement keeps your domain pure and independent while still allowing it to interact with external systems through well-defined contracts.

12.2 IoC Containers in C#

Inversion of Control containers, commonly called IoC containers or Dependency Injection containers, automate the process of creating objects and injecting their dependencies. Rather than manually instantiating every class and passing dependencies through constructors, you register types with the container and let it handle the complex task of object composition. The container maintains a registry of type mappings, understands how to construct objects, and resolves entire dependency graphs automatically. This automation becomes invaluable as your application grows and dependency chains become more complex.

The built-in Microsoft.Extensions.DependencyInjection container provides a lightweight, capable solution for most C# applications. Introduced with ASP.NET Core, this container has become the standard for .NET applications, offering a simple API and integration with the broader .NET ecosystem. While third-party containers like Autofac, Ninject, and StructureMap offer additional features, the built-in container handles the vast majority of scenarios you will encounter. Starting with the built-in container keeps your dependencies minimal and your code portable across different .NET application types.

Registering services with the container involves mapping interfaces to concrete implementations and specifying lifetimes. The registration process typically occurs in your application's startup code, often in a `Program.cs` file or a dedicated composition root class. You use extension methods like `AddTransient`, `AddScoped`, and `AddSingleton` to register services with different lifetimes. The container then uses this registration information to resolve dependencies whenever you request a service or construct an object that requires dependencies.

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Register application services
        builder.Services.AddScoped<IOrderService,
OrderService>();
        builder.Services.AddScoped<ICustomerService,
CustomerService>();

        // Register repositories
        builder.Services.AddScoped<IOrderRepository,
OrderRepository>();
        builder.Services.AddScoped<ICustomerRepository,
CustomerRepository>();

        // Register infrastructure services
        builder.Services.AddSingleton<IEmailService,
SmtpEmailService>();
        builder.Services.AddScoped<IPaymentGateway,
StripePaymentGateway>();

        // Register DbContext

builder.Services.AddDbContext<ApplicationDbContext>(options
=>
            options.UseSqlServer(

builder.Configuration.GetConnectionString("DefaultConnectio
n")));

        var app = builder.Build();
        app.Run();
    }
}
```

The container resolves dependencies through a process called dependency resolution or service location. When you request a service from the container, it examines the type's constructor, identifies all required dependencies, recursively resolves those dependencies, and constructs the entire object graph. This process happens automatically and transparently. If the container cannot resolve a dependency because no registration exists, it throws an exception at runtime. This fail-fast behavior helps you catch configuration errors early rather than discovering them deep in your application logic.

Convention-based registration reduces the boilerplate code required for service registration. Rather than manually registering each service individually, you can use reflection to scan assemblies and register types based on naming conventions or attributes. For example, you might register all classes ending in "Service" as their corresponding interfaces, or all classes implementing `IRepository<T>` automatically. This approach works well for large applications with many services, though it can make the registration process less explicit and harder to understand for newcomers to the codebase.

```
public static class ServiceCollectionExtensions
{
    public static IServiceCollection
AddApplicationServices(
        this IServiceCollection services)
    {
        var assembly = Assembly.GetExecutingAssembly();

        // Register all classes ending with "Service" as
their interfaces
        var serviceTypes = assembly.GetTypes()
            .Where(t => t.Name.EndsWith("Service")
                && !t.IsInterface
                && !t.IsAbstract);

        foreach (var serviceType in serviceTypes)
        {
            var interfaceType = serviceType.GetInterfaces()
                .FirstOrDefault(i => i.Name ==
$"I{serviceType.Name}");

            if (interfaceType != null)
            {
                services.AddScoped(interfaceType,
serviceType);
            }
        }

        return services;
    }
}
```

Container configuration should be organized by layer or feature to maintain clarity as your application grows. Rather than placing all registrations in a single method, create extension methods that group related registrations together. You might have `AddDomainServices`, `AddInfrastructureServices`, and `AddApplicationServices` methods that each handle their respective layer's registrations. This organization makes it easier to understand what services are available, reduces merge conflicts in team environments, and allows you to conditionally register services based on configuration or environment.

Understanding how containers handle open generic types enables powerful registration patterns. You can register a generic repository implementation once and have the container automatically resolve it for any entity type. For example, registering `IRepository<>` to `Repository<>` allows the container to resolve `IRepository<Order>`, `IRepository<Customer>`, and any other entity type without individual registrations. This pattern reduces duplication and ensures consistency across your repository implementations while maintaining type safety.

```
// Register open generic repository
builder.Services.AddScoped(typeof(IRepository<>),
typeof(Repository<>));

// Container can now resolve any IRepository<T>
public class OrderService
{
    private readonly IRepository<Order> _orderRepository;
    private readonly IRepository<Customer>
_customerRepository;

    public OrderService(
        IRepository<Order> orderRepository,
        IRepository<Customer> customerRepository)
    {
        _orderRepository = orderRepository;
        _customerRepository = customerRepository;
    }
}
```

12.3 Service Lifetimes and Scopes

Service lifetime determines how long an instance lives and when the container creates new instances versus reusing existing ones. The three primary lifetimes in the Microsoft.Extensions.DependencyInjection container are Transient, Scoped, and Singleton. Choosing the correct lifetime for each service is crucial for application performance, memory usage, and correctness. An incorrect lifetime choice can lead to memory leaks, concurrency issues, or unexpected behavior where services share state when they should not.

Transient lifetime creates a new instance every time the service is requested from the container. This lifetime is appropriate for lightweight, stateless services where creating new instances has minimal performance impact. Transient services should not maintain any state between method calls because each consumer receives a different instance. Use transient lifetime for services that perform calculations, transformations, or other operations that do not require maintaining state. The container does not track transient instances, so they are garbage collected when no longer referenced.

Scoped lifetime creates one instance per scope, typically corresponding to a single web request in ASP.NET Core applications. All services resolved within the same scope receive the same instance, but different scopes get different instances. This lifetime is ideal for services that should maintain state throughout a single operation but should not share state across operations. Database contexts, unit of work implementations, and repositories typically use scoped lifetime because they need to track changes within a single transaction but should not persist state across multiple requests.

```
// Transient: New instance every time
builder.Services.AddTransient<IEmailValidator,
EmailValidator>();
builder.Services.AddTransient<IPriceCalculator,
PriceCalculator>();

// Scoped: One instance per request/scope
builder.Services.AddScoped<IOrderRepository,
OrderRepository>();
builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
builder.Services.AddScoped<ApplicationDbContext>();

// Singleton: One instance for application lifetime
builder.Services.AddSingleton<IConfiguration>(configuration
);
builder.Services.AddSingleton<IMemoryCache, MemoryCache>();
builder.Services.AddSingleton<ILogger, Logger>();
```

Singleton lifetime creates a single instance that lives for the entire application lifetime. The container creates the instance on first request and reuses that same instance for all subsequent requests. Singletons are appropriate for expensive-to-create services, services that maintain application-wide state, or truly stateless services where instance creation overhead should be minimized. Configuration objects, caching services, and logging infrastructure commonly use singleton lifetime. However, singletons must be thread-safe because multiple threads will access the same instance concurrently.

Captive dependencies occur when a service with a longer lifetime holds a reference to a service with a shorter lifetime. This situation creates problems because the shorter-lived service outlives its intended scope. For example, if a singleton service depends on a scoped service, that scoped service effectively becomes a singleton because the singleton holds a reference to it for the application's lifetime. This violates the intended lifetime and can cause issues like stale data, memory leaks, or concurrency problems. The container validates these relationships and throws exceptions when it detects captive dependencies.

```
// INCORRECT: Singleton depends on scoped service
public class ReportGenerator // Registered as Singleton
{
    private readonly ApplicationDbContext _context; //
Scoped service

    public ReportGenerator(ApplicationDbContext context)
    {
        _context = context; // Captive dependency!
    }
}

// CORRECT: Use IServiceProvider to resolve scoped services
public class ReportGenerator // Registered as Singleton
{
    private readonly IServiceProvider _serviceProvider;

    public ReportGenerator(IServiceProvider
serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    public async Task<Report> GenerateReportAsync()
    {
        using var scope = _serviceProvider.CreateScope();
        var context = scope.ServiceProvider
            .GetRequiredService<ApplicationDbContext>();

        // Use context within this scope
        return await CreateReportAsync(context);
    }
}
```

Scope creation allows you to manually create dependency injection scopes when needed. While ASP.NET Core automatically creates scopes for each web request, background services, console applications, and long-running operations may need to create their own scopes. You create a scope using `IServiceProvider.CreateScope()`, resolve services from the scope's service provider, and dispose the scope when finished. This pattern ensures that scoped services are properly created and disposed, preventing memory leaks and ensuring correct lifetime management.

Lifetime selection guidelines help you choose the appropriate lifetime for your services. Use transient for lightweight, stateless services with no expensive initialization. Use scoped for services that maintain state within a single operation, particularly database contexts and repositories. Use singleton for expensive-to-create services, truly stateless services, or services that maintain application-wide state. When in doubt, prefer shorter lifetimes because they are safer and easier to reason about. You can always change to a longer lifetime if profiling reveals performance issues.

Disposal of services happens automatically for services that implement `IDisposable` or `IAsyncDisposable`. The container tracks disposable transient and scoped services and disposes them when the scope ends. Singleton services are disposed when the application shuts down. This automatic disposal management ensures that resources like database connections, file handles, and network sockets are properly released. However, you must be careful with transient disposable services because the container tracks them until the scope ends, potentially causing memory issues if you create many instances within a long-lived scope.

12.4 Composition Root Pattern

The Composition Root is the single location in your application where you configure the dependency injection container and wire up all dependencies. This pattern centralizes object composition logic, making it clear where and how your application assembles its components. The composition root should be as close to the application's entry point as possible, typically in the `Main` method or `Startup` class. By keeping composition logic in one place, you avoid scattering service instantiation throughout your codebase and maintain a clear picture of your application's structure.

Proper composition root design keeps your domain and application layers free from dependency injection concerns. These layers should not reference the dependency injection container or know anything about how services are registered. Only the composition root, which lives in your presentation or host project, should have knowledge of the container. This separation ensures that your core business logic remains portable and testable without requiring a specific dependency injection framework. Your domain and application layers depend only on abstractions, while the composition root provides the concrete implementations.

Organizing registrations by layer or feature improves maintainability as your application grows. Rather than placing all service registrations in a single method, create extension methods that group related registrations. Each layer can provide its own registration extension method that the composition root calls. For example, your infrastructure project might expose an `AddInfrastructure` method that registers all repositories and external service implementations. This approach keeps registration logic close to the implementations while maintaining a clean composition root.

```
// Infrastructure layer registration extension
public static class InfrastructureServiceExtensions
{
    public static IServiceCollection AddInfrastructure(
        this IServiceCollection services,
        IConfiguration configuration)
    {
        // Register DbContext
        services.AddDbContext<ApplicationDbContext>(options
=>
            options.UseSqlServer(

configuration.GetConnectionString("DefaultConnection")));

        // Register repositories
        services.AddScoped<IOrderRepository,
OrderRepository>();
        services.AddScoped<ICustomerRepository,
CustomerRepository>();
        services.AddScoped<IProductRepository,
ProductRepository>();

        // Register external services
        services.AddSingleton<IEmailService,
SmtpEmailService>();
        services.AddScoped<IPaymentGateway,
StripePaymentGateway>();

        return services;
    }
}
```

```
// Application layer registration extension
public static class ApplicationServiceExtensions
{
    public static IServiceCollection AddApplication(
        this IServiceCollection services)
    {
        services.AddScoped<IOrderService, OrderService>();
        services.AddScoped<ICustomerService,
CustomerService>();
        services.AddScoped<IProductService,
ProductService>();

        return services;
    }
}

// Composition root in Program.cs
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddApplication();

builder.Services.AddInfrastructure(builder.Configuration);

        var app = builder.Build();
        app.Run();
    }
}
```

The Service Locator anti-pattern occurs when you inject `IServiceProvider` into your classes and use it to resolve dependencies directly. While this approach provides flexibility, it hides dependencies and makes your code harder to test and understand. Classes that use service location do not declare their dependencies explicitly, making it unclear what they need to function. This pattern also couples your code to the dependency injection framework, reducing portability. Reserve service location for specific scenarios like factory implementations or when you need to resolve scoped services from singleton contexts.

Configuration-based registration allows you to change implementations without recompiling your application. You can read configuration values to determine which implementations to register, enabling different behavior in different environments. For example, you might use a real email service in production but a mock email service in development. This flexibility supports the Open-Closed Principle by allowing you to extend behavior through configuration rather than code changes. However, be cautious about making too many decisions based on configuration, as this can make your application's behavior less predictable.

```
public static class InfrastructureServiceExtensions
{
    public static IServiceCollection AddInfrastructure(
        this IServiceCollection services,
        IConfiguration configuration)
    {
        var emailProvider = configuration["EmailProvider"];

        switch (emailProvider)
        {
            case "Smtp":
                services.AddSingleton<IEmailService,
SmtpEmailService>();
                break;
            case "SendGrid":
                services.AddSingleton<IEmailService,
SendGridEmailService>();
                break;
            case "Mock":
                services.AddSingleton<IEmailService,
MockEmailService>();
                break;
            default:
                throw new InvalidOperationException(
                    $"Unknown email provider:
{emailProvider}");
        }

        return services;
    }
}
```

Decorator pattern implementation through dependency injection allows you to add behavior to services without modifying their code. You register multiple implementations of the same interface, with each decorator wrapping the previous implementation. This pattern is useful for cross-cutting concerns like logging, caching, or validation. The container resolves the outermost decorator, which delegates to the next decorator, eventually reaching the core implementation. This approach maintains the Open-Closed Principle and keeps your core services focused on their primary responsibilities.

Module-based composition organizes registrations into cohesive modules that can be included or excluded as needed. Each module represents a feature or subsystem and provides its own registration method. This approach works well for large applications with optional features or for building reusable components that can be shared across multiple applications. Modules encapsulate their dependencies and configuration, making it easy to understand what each feature requires and enabling you to compose applications from well-defined building blocks.

Validation of container configuration helps catch errors early in the application lifecycle. The built-in container provides a `ValidateScopes` option that checks for captive dependencies during development. You can also manually validate that all required services are registered by attempting to resolve them during application startup. This validation ensures that your application fails fast with clear error messages rather than encountering missing dependencies during runtime. Consider creating integration tests that verify your container configuration, especially for complex applications with many services.

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Enable scope validation in development
        if (builder.Environment.IsDevelopment())
        {
            builder.Host.UseDefaultServiceProvider(options
=>
            {
                options.ValidateScopes = true;
                options.ValidateOnBuild = true;
            });
        }

        builder.Services.AddApplication();

builder.Services.AddInfrastructure(builder.Configuration);

        var app = builder.Build();

        // Validate critical services are registered
        using (var scope = app.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var orderService =
services.GetRequiredService<IOrderService>();
                var dbContext =
services.GetRequiredService<ApplicationDbContext>();
                // Validate other critical services
            }
            catch (Exception ex)
            {
                var logger =
services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred
during service validation");
                throw;
            }
        }

        app.Run();
    }
}
```

13 Chapter 12: Presentation Layer with Blazor

The presentation layer represents the outermost ring of your domain-centric architecture, serving as the interface between users and your application's business logic. In modern C# development, Blazor has emerged as a compelling framework for building interactive web applications that can run both on the server and in the browser via WebAssembly. When properly integrated with Clean Architecture principles, Blazor components become thin presentation shells that delegate all business logic to your application layer, maintaining the strict separation of concerns that makes your system maintainable and testable.

Many developers struggle with Blazor integration because they're tempted to place business logic directly in components, creating tight coupling between the UI and domain concerns. This approach leads to components that are difficult to test, hard to maintain, and impossible to reuse across different presentation technologies. The key to successful Blazor integration lies in treating components as pure presentation elements that coordinate between user interactions and application services, never containing business rules or data access logic themselves.

This chapter explores how to structure Blazor applications within a domain-centric architecture, ensuring your components remain focused on presentation while your business logic stays safely isolated in the domain and application layers. You'll learn practical strategies for component design, dependency injection configuration, state management, and communication patterns that preserve architectural boundaries. By the end of this chapter, you'll understand how to build Blazor applications that are both user-friendly and architecturally sound, with clear separation between what users see and how your business operates.

13.1 Blazor in Clean Architecture

Blazor fits naturally into the frameworks and drivers layer of Clean Architecture, sitting at the outermost ring where external interfaces reside. Your Blazor components should never directly reference domain entities or infrastructure concerns like database contexts. Instead, they communicate exclusively with the application layer through well-defined interfaces, typically application services that expose use cases. This arrangement ensures that changes to your UI framework don't ripple through your business logic, and conversely, domain changes don't require modifications to every component that displays related data.

The typical flow in a Blazor-based Clean Architecture application follows a clear pattern. A user interacts with a Blazor component, triggering an event handler. That handler calls an application service method, passing any necessary data as command or query objects. The application service orchestrates domain operations, potentially involving repositories, domain services, and aggregates. Results flow back through DTOs (Data Transfer Objects) to the component, which updates its state and re-renders. This unidirectional flow maintains clear boundaries and makes the system predictable and testable.

Consider a practical example of an order management system. Your Blazor component displays order information and provides a button to approve an order. The component structure should look like this:

- **Component responsibility:** Render order details, capture user approval action, display success or error messages
- **Application service responsibility:** Validate approval permissions, load the order aggregate, invoke the approval method, persist changes
- **Domain responsibility:** Enforce business rules about order approval, update order status, raise domain events
- **Infrastructure responsibility:** Retrieve and save order data from the database

Your Blazor project should be structured as a separate assembly that references only the application layer, never the domain or infrastructure directly. This physical separation enforces the dependency rule at compile time. If a developer tries to inject a repository directly into a component, the code won't compile because the necessary references aren't available. This architectural constraint guides developers toward correct patterns and prevents accidental violations of layer boundaries.

When organizing your solution, create a clear project structure that reflects these boundaries. A typical setup includes:

- `YourApp.Domain` - Contains entities, value objects, domain services, and interfaces
- `YourApp.Application` - Contains application services, DTOs, and use case implementations
- `YourApp.Infrastructure` - Contains repository implementations, EF Core configurations, external service integrations
- `YourApp.Blazor` - Contains Blazor components, pages, and presentation logic
- `YourApp.Blazor.Server` or `YourApp.Blazor.WebAssembly` - Contains hosting configuration and startup logic

The hosting project serves as the composition root where dependency injection is configured. This is where you wire up all your services, repositories, and infrastructure concerns. The Blazor component project remains clean, containing only UI code and references to application service interfaces. This separation makes it possible to switch between Blazor Server and Blazor WebAssembly hosting models, or even to add additional presentation layers like a REST API or desktop application, without duplicating business logic.

One common mistake is creating "view models" in the Blazor project that contain business logic or data access code. This pattern, borrowed from traditional MVC applications, violates Clean Architecture principles when those view models do more than simple presentation formatting. Instead, use DTOs from your application layer for data transfer, and keep any presentation-specific logic (like formatting dates for display or managing

UI state) directly in your components. If you find yourself writing complex logic in a view model, that logic probably belongs in an application service instead.

Authentication and authorization present special considerations in Blazor applications. While you'll implement authorization checks in your application and domain layers to enforce business rules, Blazor components also need to respond to user permissions for UI purposes—hiding buttons users can't use, disabling features they can't access. Handle this by injecting an authorization service from your application layer that provides permission-checking methods. Your components call these methods to determine what to display, but the actual enforcement happens deeper in the architecture when use cases execute.

13.2 Component Design and Separation of Concerns

Effective Blazor component design requires thinking carefully about responsibilities and boundaries. Each component should have a single, well-defined purpose within the presentation layer. Large, monolithic components that handle multiple concerns become difficult to test, reuse, and maintain. Instead, compose your UI from smaller, focused components that each handle one aspect of the user interface. This approach mirrors the Single Responsibility Principle at the component level, making your presentation layer as maintainable as your domain layer.

Consider three categories of components in your Blazor application: page components, feature components, and shared components. Page components correspond to routes and serve as composition roots for a particular view, coordinating between multiple feature components. Feature components implement specific functionality like displaying a product list or handling order submission. Shared components provide reusable UI elements like buttons, modals, or data grids that appear throughout your application. Each category has different responsibilities and different relationships with your application services.

Page components typically inject multiple application services and coordinate complex workflows. They handle routing parameters, manage page-level state, and orchestrate interactions between feature components. Here's an example structure for an order details page:

- **Inject services:** `IOrderQueryService`, `IOrderCommandService`, `IAuthorizationService`
- **Load data:** Retrieve order details using the query service when the page initializes
- **Compose UI:** Render child components for order header, line items, and actions
- **Handle events:** Respond to events from child components by calling appropriate command services
- **Manage state:** Track loading states, error messages, and success notifications

Feature components focus on specific functionality and should be designed for reusability. An `OrderLineItemList` component, for example, receives order line items as a parameter and displays them in a table. It doesn't know how to load that data or what page it's being used on. It might raise events when users interact with line items, but it doesn't handle those events itself—that's the parent component's responsibility. This separation makes the component testable in isolation and reusable across different contexts.

Avoid the temptation to inject application services into every component. Shared components especially should be pure presentation elements that work entirely through parameters and events. If a button component needs to know about your domain model to function, you've coupled your

UI too tightly to your business logic. Instead, the parent component that uses the button should handle the domain interaction and simply tell the button what to display and what event to raise when clicked.

Parameter design significantly impacts component reusability and testability. Use strongly-typed parameters that clearly communicate what data the component needs. Avoid passing entire domain entities to components; instead, create presentation-specific DTOs or use primitive types. For example, instead of passing an `Order` entity to a component, pass an `OrderSummaryDto` that contains only the fields needed for display. This prevents components from accidentally accessing or modifying domain objects inappropriately.

Event handling in components should follow a clear pattern: capture user interaction, validate input at the UI level (format, required fields), call the appropriate application service, handle the result. Never put business logic in event handlers. Consider this example of an order approval handler:

Component lifecycle methods like `OnInitializedAsync` and `OnParametersSetAsync` are appropriate places to call application services for data loading. Keep this logic simple and focused on coordination. If you find yourself writing complex logic in lifecycle methods, extract that logic into an application service. The component should orchestrate, not implement, business functionality. Use these methods to trigger service calls, update component state based on results, and handle errors appropriately.

Error handling in components requires careful consideration. Components should catch exceptions from application service calls and display user-friendly error messages, but they shouldn't try to recover from business rule violations or handle domain-specific errors. Your application services should return result objects that indicate success or failure with appropriate error information. Components then translate these results into UI feedback—error messages, validation summaries, or toast notifications—without needing to understand the underlying business rules that caused the failure.

13.3 Dependency Injection in Blazor

Blazor's built-in dependency injection system integrates seamlessly with Clean Architecture principles, providing the mechanism to inject application services into components while maintaining proper layer separation. The key is configuring your services correctly in the composition root and ensuring components depend only on abstractions from the application layer. This approach keeps your presentation layer loosely coupled and makes components easy to test by allowing you to substitute mock implementations during testing.

Service registration happens in your hosting project's `Program.cs` file (or `Startup.cs` in older templates). This is where you configure the entire dependency graph, registering implementations for all the interfaces your application uses. The registration follows a clear pattern: register application services, register infrastructure services (repositories, external service clients), register domain services if needed, and configure framework-specific services like authentication and authorization. Here's a typical registration structure:

- **Application services:**`builder.Services.AddScoped<IOrderCommandService, OrderCommandService>()`
- **Repositories:**`builder.Services.AddScoped<IOrderRepository, OrderRepository>()`
- **DbContext:**`builder.Services.AddDbContext<ApplicationDbContext>(options => ...)`

- **Domain services:**`builder.Services.AddScoped<IPricingService, PricingService>()`

Service lifetimes matter significantly in Blazor applications, especially when using Blazor Server where components maintain state across multiple user interactions. Use `AddScoped` for services that should be created once per circuit (in Blazor Server) or per request (in Blazor WebAssembly). This includes your application services and repositories. Use `AddSingleton` for services that maintain application-wide state and are thread-safe. Use `AddTransient` sparingly, typically only for lightweight services that have no state and are inexpensive to create.

In Blazor components, inject services using the `@inject` directive at the top of your component file or through constructor injection in code-behind files. The `@inject` directive is more common in Blazor because it's concise and works well with the component model. Always inject interfaces, never concrete implementations. This maintains the dependency inversion principle and keeps your components testable. Here's an example of proper service injection in a component:

```
@page "/orders/{OrderId:int}"
@inject IOrderQueryService OrderQueryService
@inject IOrderCommandService OrderCommandService
@inject NavigationManager Navigation

<h3>Order Details</h3>

@if (order == null)
{
    <p>Loading...</p>
}
else
{
    <OrderDetailsDisplay Order="@order" />
    <button @onclick="ApproveOrder">Approve Order</button>
}

@code {
    [Parameter]
    public int OrderId { get; set; }

    private OrderDetailsDto? order;

    protected override async Task OnInitializedAsync()
    {
        order = await
OrderQueryService.GetOrderDetailsAsync(OrderId);
    }

    private async Task ApproveOrder()
    {
        var result = await
OrderCommandService.ApproveOrderAsync(OrderId);
        if (result.IsSuccess)
        {
            Navigation.NavigateTo("/orders");
        }
    }
}
```

Avoid injecting infrastructure services directly into components. Components should never inject `DbContext`, repository implementations, or external service clients. These are infrastructure concerns that belong behind application service interfaces. If you find yourself wanting to inject a repository into a component, that's a sign you need to create an application service that encapsulates the use case you're trying to implement. The component should interact with that application service instead.

Testing components with dependency injection becomes straightforward when you follow these patterns. Create mock implementations of your application service interfaces that return predictable test data. Register these mocks in your test context instead of the real implementations. Your component tests can then verify that components call the correct service methods with the correct parameters and respond appropriately to different return values. This approach tests component behavior without requiring a database or external services.

Consider organizing your service registrations into extension methods that group related services together. This keeps your `Program.cs` clean and makes it easy to see what services are registered. Create methods like `AddApplicationServices`, `AddInfrastructureServices`, and `AddDomainServices` in their respective projects. Each method encapsulates the registration logic for that layer, promoting separation of concerns even in your composition root.

When working with Blazor WebAssembly, be mindful of service registration differences compared to Blazor Server. WebAssembly runs in the browser, so services like database contexts can't be registered directly. Instead, you'll typically register HTTP clients that communicate with a backend API. Your application service interfaces remain the same, but the implementations differ—in WebAssembly, they make HTTP calls instead of direct database access. This is another benefit of proper abstraction: the same components work in both hosting models because they depend on interfaces, not implementations.

Scoped services in Blazor Server persist for the lifetime of the circuit, which can be much longer than a typical HTTP request. This means you need to be careful about services that hold state or resources. Repositories that inject a `DbContext` should be scoped, ensuring each circuit gets its own context instance. Be cautious with services that cache data—what seems like a performance optimization can lead to stale data problems when the circuit lives for minutes or hours. Consider implementing explicit refresh mechanisms or using shorter-lived caches.

13.4 State Management Strategies

State management in Blazor applications requires careful consideration to maintain architectural boundaries while providing a responsive user experience. The fundamental principle is that components should manage only presentation state—things like whether a modal is open, which tab is selected, or whether a form is in edit mode. Business state—the actual data your application operates on—should live in your domain layer and be accessed through application services. Mixing these concerns leads to components that are difficult to test and maintain.

Component-level state is the simplest form of state management and should be your default choice. Each component maintains its own state in private fields or properties, updating that state in response to user interactions or service calls. This state is local to the component and doesn't need to be shared with other parts of the application. For example, a search component might maintain the current search term, whether results are loading, and any error messages. This state is purely presentational and doesn't need to persist beyond the component's lifetime.

When multiple components need to share state, you have several options that maintain architectural integrity. The simplest is passing state through component parameters and events—parent components manage state and pass it down to children, while children raise events that parents handle by updating state. This works well for closely related components in the same feature area. For example, a product list page might manage the current filter state and pass it to both a filter component and a results component.

For state that needs to be shared across unrelated components or persisted across navigation, consider creating application-level state services. These are scoped services that components inject to access and modify shared state. However, these services should manage only presentation state, not business state. A `NavigationStateService` might track breadcrumb information or remember which page the user came from. A `NotificationService` might manage a queue of toast messages to display. These services coordinate presentation concerns without duplicating business logic.

Implement state services with clear interfaces and event notifications so components can react to state changes. Here's an example of a notification service that maintains architectural boundaries:

- **Service responsibility:** Maintain a collection of notification messages, provide methods to add and remove notifications, raise events when notifications change
- **Component responsibility:** Inject the service, subscribe to change events, display notifications in the UI, call service methods to dismiss notifications
- **Application service responsibility:** After completing business operations, add success or error notifications to the notification service

Avoid creating state services that duplicate your domain model or cache business data. If you find yourself creating a `OrderStateService` that loads and caches orders, you're probably violating architectural boundaries. Instead, components should call application services to retrieve data when needed, and those services coordinate with repositories to access the domain. If performance is a concern, implement caching in the infrastructure layer where it belongs, not in the presentation layer.

Browser storage (localStorage and sessionStorage) can be useful for persisting presentation state across browser sessions or tabs. Use this for things like user preferences, UI settings, or draft form data. Never store sensitive business data or authentication tokens in browser storage without proper encryption. Access browser storage through a service abstraction so your components don't directly depend on JavaScript interop. This makes your code more testable and allows you to change storage mechanisms without modifying components.

The Blazor `CascadingValue` component provides a way to pass state down through the component tree without explicitly passing parameters through every level. Use this sparingly and only for truly cross-cutting presentation concerns like theme settings or user preferences. Overusing cascading values makes component dependencies implicit and harder to understand. If you find yourself cascading business data, that's a sign you should be using application services instead.

State synchronization between Blazor Server and clients happens automatically for component state, but you need to be mindful of timing and race conditions. When a user triggers an action that modifies state, ensure you update the UI optimistically if appropriate, showing loading indicators while operations complete. Handle the case where operations fail and you need to revert optimistic updates. This is purely presentation logic—the actual business operation succeeds or fails based on domain rules, and the component simply reflects that outcome in the UI.

Consider implementing the Command Query Responsibility Segregation (CQRS) pattern in your application layer to support efficient state management. Query services return read-optimized DTOs that components can display directly without transformation. Command services accept command objects and return simple result indicators. This separation makes it clear which operations modify state and which simply retrieve it, helping you design components that interact with your application layer in predictable ways.

When dealing with real-time updates in Blazor Server, use SignalR to push changes from the server to connected clients. However, structure this carefully to maintain architectural boundaries. Don't push domain entities directly to clients. Instead, have your application services raise events when important business operations complete, and use a SignalR hub to translate those events into client notifications. Components subscribe to these notifications and refresh their data by calling query services, maintaining the proper flow of dependencies.

Form state deserves special attention because forms often represent complex user interactions with your domain. Use Blazor's built-in `EditForm` component with model binding to manage form state. Create form models (simple DTOs) that represent the data being edited, separate from your domain entities. When the form is submitted, map the form model to a command object and pass it to an application service. The service validates the command, executes the business logic, and returns a result. The component updates its state based on that result, displaying validation errors or success messages as appropriate.

14 CHAPTER 13: TESTING DOMAIN-CENTRIC APPLICATIONS

Testing domain-centric applications offers distinct advantages over testing traditional layered architectures. When your business logic lives in a pure domain layer with no external dependencies, you can test it in isolation without databases, web servers, or external services. This separation creates fast, reliable tests that verify your core business rules independently. The architecture naturally supports the testing pyramid, where you write many unit tests, fewer integration tests, and minimal end-to-end tests. Each layer has clear boundaries and responsibilities, making it obvious what to test and how to test it.

The dependency rule that governs domain-centric architecture also governs your testing strategy. Since dependencies flow inward toward the domain, your domain layer has no dependencies on infrastructure or frameworks. This means domain tests run in milliseconds without any setup overhead. Application layer tests require slightly more setup as they coordinate between domain and infrastructure, but you can still use test doubles to keep them fast. Infrastructure tests verify actual integrations with databases and external services, accepting slower execution times in exchange for confidence that your adapters work correctly.

Your testing approach should match the architectural layers. Domain layer tests focus on business rules and invariants, ensuring entities and aggregates enforce their constraints correctly. Application layer tests verify use case orchestration, confirming that commands and queries coordinate domain objects and repositories properly. Infrastructure layer tests validate that your implementations of repository interfaces, external service adapters, and database mappings function as expected. Presentation layer tests ensure your UI components interact correctly with application services. This chapter explores each testing level in detail, providing concrete strategies and examples for testing domain-centric C# applications.

14.1 The Testing Pyramid

The testing pyramid represents a balanced testing strategy with many fast unit tests at the base, fewer integration tests in the middle, and minimal end-to-end tests at the top. This distribution optimizes for speed, reliability, and maintenance cost. Unit tests execute in milliseconds and pinpoint exact failures, making them ideal for rapid feedback during development. Integration tests verify that components work together correctly but run slower due to external dependencies. End-to-end tests validate complete user workflows but are brittle, slow, and expensive to maintain. Domain-centric architecture naturally supports this pyramid because the dependency rule creates testable boundaries between layers.

In a domain-centric application, your testing pyramid should follow this approximate distribution:

- **70% unit tests** covering domain entities, value objects, domain services, and application services
- **20% integration tests** verifying repository implementations, database mappings, and external service adapters
- **10% end-to-end tests** validating critical user workflows through the entire application stack

Unit tests form the foundation because they test business logic in isolation. A well-designed domain layer contains most of your application's complexity in the form of business rules, validations, and calculations. These rules should be thoroughly tested without any infrastructure dependencies. For example, testing an `Order` aggregate's ability to add items, apply discounts, and calculate totals requires no database or external services. You instantiate the aggregate, call its methods, and assert the results. These tests run thousands per second, providing immediate feedback when you break something.

Integration tests verify that your infrastructure implementations correctly interact with external systems. These tests use real database connections, actual file systems, or test instances of external APIs. They confirm that your Entity Framework mappings correctly persist and retrieve aggregates, that your email service adapter sends messages properly, and that your repository implementations honor the contracts defined in your domain layer. While slower than unit tests, integration tests catch issues that mocks cannot reveal, such as incorrect SQL queries, serialization problems, or API contract mismatches.

End-to-end tests validate complete user scenarios from the presentation layer through to the database and back. In a Blazor application, this might mean using a headless browser to click buttons, fill forms, and verify displayed results. These tests are valuable for critical workflows like user registration, checkout processes, or report generation. However, they're expensive to write and maintain because they break when any layer changes. They also run slowly and can fail intermittently due to timing issues or environmental factors. Use them sparingly for high-value scenarios that justify their cost.

The testing pyramid inverts when teams neglect unit tests and rely primarily on integration or end-to-end tests. This anti-pattern creates slow test suites that take minutes or hours to run, discouraging developers from running tests frequently. When tests run slowly, developers skip them during development and only run them in CI/CD pipelines, delaying feedback and making debugging harder. The inverted pyramid also makes tests brittle because changes in infrastructure or UI break many tests simultaneously. Domain-centric architecture prevents this anti-pattern by making unit testing natural and integration testing explicit.

Your test suite should execute quickly enough to run on every code change. Aim for your entire unit test suite to complete in under ten seconds, allowing developers to run tests continuously during development. Integration tests might take thirty seconds to a minute, acceptable for running before commits. End-to-end tests can take several minutes but should only run in CI/CD pipelines or before releases. This speed distribution encourages frequent testing and rapid feedback, catching bugs when they're easiest to fix. Domain-centric architecture achieves these speeds by isolating business logic from slow external dependencies.

Consider a practical example: testing an e-commerce order processing system. You might have 200 unit tests covering order creation, item addition, discount application, inventory validation, and price calculation. These tests run in two seconds total. You'd have 40 integration tests verifying that orders persist correctly to the database, inventory updates trigger properly, and payment gateway integration works. These run in thirty seconds. Finally, you'd have five end-to-end tests covering complete checkout workflows for different customer types. These take three minutes. This distribution provides comprehensive coverage while maintaining fast feedback cycles.

14.2 Unit Testing the Domain Layer

Unit testing the domain layer focuses on verifying business rules, invariants, and domain logic without any external dependencies. Your domain entities, value objects, and domain services should contain pure business logic that's easy to test. Each test instantiates domain objects, invokes methods, and asserts expected outcomes. Because domain objects have no dependencies on infrastructure, databases, or frameworks, these tests run extremely fast and require no setup or teardown. This makes them ideal for test-driven development, where you write tests before implementation and run them continuously during coding.

Testing entities involves verifying that they enforce their invariants correctly. An entity should never allow itself to enter an invalid state. For example, an `Order` entity might enforce that it cannot be submitted without at least one item, that quantities must be positive, and that discounts cannot exceed the order total. Your tests should attempt to violate these rules and verify that the entity prevents invalid operations. Here's a concrete example testing an order entity:

```
public class OrderTests
{
    [Fact]
    public void AddItem_WithValidProduct_AddsItemToOrder()
    {
        // Arrange
        var order = new Order(customerId: 123);
        var product = new Product(id: 1, name: "Widget",
price: 29.99m);

        // Act
        order.AddItem(product, quantity: 2);

        // Assert
        Assert.Single(order.Items);
        Assert.Equal(2, order.Items.First().Quantity);
        Assert.Equal(59.98m, order.Total);
    }

    [Fact]
    public void
AddItem_WithZeroQuantity_ThrowsDomainException()
    {
        // Arrange
        var order = new Order(customerId: 123);
        var product = new Product(id: 1, name: "Widget",
price: 29.99m);

        // Act & Assert
        var exception = Assert.Throws<DomainException>(
            () => order.AddItem(product, quantity: 0)
        );
        Assert.Equal("Quantity must be greater than zero",
exception.Message);
    }

    [Fact]
    public void Submit_WithoutItems_ThrowsDomainException()
    {
        // Arrange
        var order = new Order(customerId: 123);

        // Act & Assert
        var exception = Assert.Throws<DomainException>(
            () => order.Submit()
        );
        Assert.Equal("Cannot submit an empty order",
exception.Message);
    }
}
```

Value objects require testing their equality semantics and validation logic. Unlike entities, value objects are identified by their properties rather than an ID. Two value objects with identical properties should be considered equal. Your tests should verify this equality behavior and ensure that value objects reject invalid values. For example, an `EmailAddress` value object should validate email format and implement proper equality comparison. Test both valid and invalid inputs to ensure your validation logic is comprehensive.

Domain services contain business logic that doesn't naturally belong to a single entity. Testing domain services follows the same pattern as testing entities: instantiate the service, call its methods with various inputs, and assert expected outputs. The key difference is that domain services often coordinate multiple entities or perform calculations across aggregates. For example, a `PricingService` might calculate discounts based on customer tier, order volume, and promotional rules. Your tests should cover all calculation paths and edge cases:

```
public class PricingServiceTests
{
    [Theory]
    [InlineData(CustomerTier.Standard, 100, 100)]
    [InlineData(CustomerTier.Premium, 100, 90)]
    [InlineData(CustomerTier.Enterprise, 100, 80)]
    public void CalculatePrice_AppliesTierDiscount(
        CustomerTier tier, decimal basePrice, decimal
expectedPrice)
    {
        // Arrange
        var service = new PricingService();
        var customer = new Customer(id: 1, tier: tier);

        // Act
        var finalPrice = service.CalculatePrice(customer,
basePrice);

        // Assert
        Assert.Equal(expectedPrice, finalPrice);
    }

    [Fact]
    public void
CalculatePrice_WithVolumeDiscount_AppliesBothDiscounts()
    {
        // Arrange
        var service = new PricingService();
        var customer = new Customer(id: 1, tier:
CustomerTier.Premium);
```

```
        // Act - Premium gets 10% off, volume over 1000
gets additional 5%
        var finalPrice = service.CalculatePrice(customer,
basePrice: 1000, quantity: 100);

        // Assert
        Assert.Equal(85500m, finalPrice); // 90% of base *
95% volume discount
    }
}
```

Testing domain events ensures that entities raise appropriate events when significant state changes occur. Domain events enable loose coupling between aggregates and allow other parts of the system to react to domain changes. Your tests should verify that events are raised with correct data at the right times. For example, when an order is submitted, it should raise an `OrderSubmittedEvent` containing the order ID, customer ID, and total amount. Test that events are raised for all significant state transitions and that they contain all necessary information for event handlers.

Aggregate testing focuses on consistency boundaries and invariant enforcement. An aggregate root controls access to all entities within its boundary, ensuring that business rules are never violated. Your tests should attempt to violate aggregate invariants through various paths and verify that the aggregate prevents invalid operations. For example, an `Order` aggregate might prevent adding items after submission, changing quantities to negative values, or applying discounts that exceed the order total. Test all public methods and ensure they maintain aggregate consistency.

Use test data builders to create complex domain objects for testing. As your domain model grows, setting up test scenarios becomes verbose and repetitive. Test data builders provide fluent APIs for constructing domain objects with sensible defaults and easy customization. This pattern makes tests more readable and maintainable. Here's an example builder for creating test orders:

```
public class OrderBuilder
{
    private int _customerId = 1;
    private List<(Product product, int quantity)> _items =
new();
    private OrderStatus _status = OrderStatus.Draft;

    public OrderBuilder WithCustomer(int customerId)
    {
        _customerId = customerId;
        return this;
    }

    public OrderBuilder WithItem(Product product, int
quantity)
    {
        _items.Add((product, quantity));
        return this;
    }

    public OrderBuilder WithStatus(OrderStatus status)
    {
        _status = status;
        return this;
    }

    public Order Build()
    {
        var order = new Order(_customerId);
        foreach (var (product, quantity) in _items)
        {
            order.AddItem(product, quantity);
        }
        if (_status == OrderStatus.Submitted)
        {
            order.Submit();
        }
        return order;
    }
}

// Usage in tests
var order = new OrderBuilder()
    .WithCustomer(123)
    .WithItem(product1, quantity: 2)
    .WithItem(product2, quantity: 1)
    .WithStatus(OrderStatus.Submitted)
    .Build();
```

Parameterized tests reduce duplication when testing similar scenarios with different inputs. Use theories in xUnit or test cases in NUnit to run the same test logic with multiple parameter sets. This approach is particularly useful for testing validation logic, calculations, or state transitions that follow the same pattern but with different values. Parameterized tests make your test suite more maintainable because you add new test cases by adding data rather than duplicating test methods. They also make test intent clearer by showing the relationship between inputs and expected outputs in a tabular format.

14.3 Integration Testing with Repositories

Integration tests verify that your infrastructure implementations correctly interact with external systems like databases, file systems, and APIs.

Unlike unit tests that use mocks and test doubles, integration tests use real implementations to catch issues that only appear when components interact. Repository integration tests are particularly important because they verify that your Entity Framework mappings correctly persist and retrieve domain aggregates. These tests ensure that your domain objects survive the round trip to the database without losing data or violating invariants.

Setting up integration tests requires creating a test database that mirrors your production schema. Use an in-memory database like SQLite for fast tests, or use a containerized database like PostgreSQL or SQL Server for tests that need to verify database-specific behavior. The key is to ensure each test runs in isolation with a clean database state. Use transactions that roll back after each test, or recreate the database between tests. Here's a base class for repository integration tests using Entity Framework Core with SQLite:

```
public abstract class RepositoryTestBase : IDisposable
{
    protected ApplicationDbContext Context { get; }

    protected RepositoryTestBase()
    {
        var options = new
DbContextOptionsBuilder<ApplicationDbContext>()
            .UseSqlite("DataSource=:memory:")
            .Options;

        Context = new ApplicationDbContext(options);
        Context.Database.OpenConnection();
        Context.Database.EnsureCreated();
    }

    public void Dispose()
    {
        Context.Database.CloseConnection();
        Context.Dispose();
    }
}
```

Testing repository implementations involves verifying all CRUD operations and query methods. Each test should add entities to the database, retrieve them, and verify that all properties are correctly persisted and loaded. Pay special attention to complex mappings like value objects, collections, and owned entities. For example, testing an `OrderRepository` should verify that orders persist with all their items, that value objects like addresses are correctly mapped, and that navigation properties load properly:

```
public class OrderRepositoryTests : RepositoryTestBase
{
    [Fact]
    public async Task Add_PersistsOrderWithItems()
    {
        // Arrange
        var repository = new OrderRepository(Context);
        var order = new Order(customerId: 123);
        var product = new Product(id: 1, name: "Widget",
price: 29.99m);
        order.AddItem(product, quantity: 2);

        // Act
        await repository.AddAsync(order);
        await Context.SaveChangesAsync();

        // Assert
        var retrieved = await
repository.GetByIdAsync(order.Id);
        Assert.NotNull(retrieved);
        Assert.Equal(123, retrieved.CustomerId);
        Assert.Single(retrieved.Items);
        Assert.Equal(2, retrieved.Items.First().Quantity);
        Assert.Equal(59.98m, retrieved.Total);
    }

    [Fact]
    public async Task Update_PersistsChanges()
    {
        // Arrange
        var repository = new OrderRepository(Context);
        var order = new Order(customerId: 123);
        var product = new Product(id: 1, name: "Widget",
price: 29.99m);
        order.AddItem(product, quantity: 2);
        await repository.AddAsync(order);
        await Context.SaveChangesAsync();

        // Act
        var retrieved = await
repository.GetByIdAsync(order.Id);
        retrieved.AddItem(product, quantity: 1);
        await repository.UpdateAsync(retrieved);
        await Context.SaveChangesAsync();

        // Assert
        var updated = await
repository.GetByIdAsync(order.Id);
        Assert.Equal(2, updated.Items.Count);
        Assert.Equal(89.97m, updated.Total);
    }
}
```

Query method testing ensures that your repository correctly implements complex queries and filters. Test methods that find entities by various criteria, apply sorting and pagination, and load related entities. Verify that queries return correct results for edge cases like empty result sets, single results, and large result sets. For example, testing a method that finds orders by customer and date range should verify correct filtering, proper handling of boundary dates, and correct ordering of results.

Testing aggregate persistence requires special attention to consistency boundaries. When you persist an aggregate root, all entities within its boundary should be persisted atomically. Your tests should verify that adding items to an order, removing items, and updating quantities all persist correctly. Test that orphaned entities are deleted when removed from the aggregate. For example, if you remove an order item, the corresponding database row should be deleted, not just orphaned with a null foreign key.

Concurrency testing verifies that your repository handles concurrent updates correctly. Entity Framework provides optimistic concurrency through row version columns or timestamp fields. Your tests should simulate concurrent updates and verify that the second update detects the conflict and throws a `DbUpdateConcurrencyException`. This ensures that your application doesn't silently overwrite changes made by other users:

```
[Fact]
public async Task
Update_WithConcurrentModification_ThrowsConcurrencyExceptio
n()
{
    // Arrange
    var repository = new OrderRepository(Context);
    var order = new Order(customerId: 123);
    await repository.AddAsync(order);
    await Context.SaveChangesAsync();

    // Act - Simulate two users loading the same order
    var order1 = await repository.GetByIdAsync(order.Id);
    var order2 = await repository.GetByIdAsync(order.Id);

    // First user updates and saves
    order1.AddItem(new Product(1, "Widget", 29.99m), 1);
    await repository.UpdateAsync(order1);
    await Context.SaveChangesAsync();

    // Second user tries to update the stale version
    order2.AddItem(new Product(2, "Gadget", 39.99m), 1);
    await repository.UpdateAsync(order2);

    // Assert
    await Assert.ThrowsAsync<DbUpdateConcurrencyException>(
        async () => await Context.SaveChangesAsync()
    );
}
```

Performance testing in integration tests helps identify slow queries and N+1 query problems. While not traditional performance tests with load and stress testing, these tests verify that your queries execute efficiently. Use Entity Framework's query logging to inspect generated SQL and ensure queries use appropriate joins and indexes. Test that loading an aggregate with many related entities doesn't generate hundreds of individual queries. Use eager loading with `Include` or split queries when appropriate to optimize performance.

Database migration testing ensures that your schema changes don't break existing functionality. Create tests that apply migrations to a database with existing data and verify that data is preserved and transformed correctly. This is particularly important when renaming columns, splitting tables, or changing data types. Test both upgrade and downgrade migrations to ensure you can roll back changes if needed. These tests catch migration issues before they reach production and corrupt data.

External service integration tests verify that your adapters correctly interact with third-party APIs and services. Use test instances or sandbox environments provided by the service, or use tools like WireMock to simulate API responses. Test both success and failure scenarios, including network timeouts, invalid responses, and rate limiting. For example, testing a payment gateway adapter should verify successful payments, declined cards, network errors, and timeout handling. These tests ensure your application handles external service failures gracefully.

14.4 Mocking and Test Doubles

Test doubles replace real dependencies with simplified implementations that make testing easier and faster. The term encompasses several patterns including mocks, stubs, fakes, and spies, each serving different testing needs. In domain-centric architecture, you primarily use test doubles when testing the application layer, where use cases coordinate between domain objects and infrastructure services. The domain layer itself rarely needs mocks because it has no external dependencies.

Understanding when and how to use each type of test double improves your test quality and maintainability.

Stubs provide predetermined responses to method calls without any behavior verification. Use stubs when you need a dependency to return specific values but don't care how many times it's called or with what arguments. For example, when testing a use case that retrieves a customer, you might stub the repository to return a specific customer object. The test focuses on what the use case does with that customer, not on repository interaction details. Stubs are the simplest test doubles and should be your default choice when you just need to provide data:

```
public class StubCustomerRepository : ICustomerRepository
{
    private readonly Customer _customer;

    public StubCustomerRepository(Customer customer)
    {
        _customer = customer;
    }

    public Task<Customer> GetByIdAsync(int id)
    {
        return Task.FromResult(_customer);
    }

    // Other methods throw NotImplementedException
}
```

Mocks verify that specific interactions occur between the system under test and its dependencies. Use mocks when the interaction itself is important to test, such as verifying that a use case calls a repository's save method or that an event handler publishes a domain event. Mocking frameworks like Moq or NSubstitute make creating mocks easy and provide fluent APIs for setting up expectations and verifying calls. However, overusing mocks leads to brittle tests that break when implementation details change. Reserve mocks for testing critical interactions:

```
public class PlaceOrderCommandHandlerTests
{
    [Fact]
    public async Task Handle_SavesOrderToRepository()
    {
        // Arrange
        var mockRepository = new Mock<IOrderRepository>();
        var handler = new
PlaceOrderCommandHandler(mockRepository.Object);
        var command = new PlaceOrderCommand(customerId:
123, items: new[]
        {
            new OrderItemDto(productId: 1, quantity: 2)
        });

        // Act
        await handler.Handle(command);

        // Assert
        mockRepository.Verify(
            r => r.AddAsync(It.Is<Order>(o => o.CustomerId
== 123)),
            Times.Once
        );
    }
}
```

Fakes are working implementations with simplified behavior suitable for testing but not production. An in-memory repository that stores entities in a dictionary is a common fake. Fakes are more realistic than stubs because they implement actual behavior, making tests more reliable. They're particularly useful for integration tests where you want real behavior without the overhead of external dependencies. For example, an in-memory repository fake allows testing use cases with realistic repository behavior without database setup:

```
public class InMemoryOrderRepository : IOrderRepository
{
    private readonly Dictionary<int, Order> _orders =
new();
    private int _nextId = 1;

    public Task<Order> GetByIdAsync(int id)
    {
        return
Task.FromResult(_orders.GetValueOrDefault(id));
    }

    public Task AddAsync(Order order)
    {
        if (order.Id == 0)
        {
            order.SetId(_nextId++);
        }
        _orders[order.Id] = order;
        return Task.CompletedTask;
    }

    public Task UpdateAsync(Order order)
    {
        _orders[order.Id] = order;
        return Task.CompletedTask;
    }

    public Task<List<Order>> GetByCustomerIdAsync(int
customerId)
    {
        return Task.FromResult(
            _orders.Values.Where(o => o.CustomerId ==
customerId).ToList()
        );
    }
}
```

Spies record information about how they're called, allowing you to verify interactions after the fact. Unlike mocks that set up expectations before execution, spies capture actual calls and let you inspect them afterward. This approach feels more natural and produces clearer test code. You can implement simple spies manually or use mocking frameworks in spy mode. Spies are useful when you want to verify interactions but don't want to couple tests to specific call sequences or argument matchers.

Choosing between test double types depends on what you're testing. Use stubs when you need to provide data to the system under test. Use mocks when verifying that specific interactions occur is critical to the test's purpose. Use fakes when you want realistic behavior without external dependencies. Use spies when you want to verify interactions but prefer a more flexible approach than mocks. In domain-centric architecture, you'll use stubs and fakes most often because tests focus on behavior rather than interaction verification.

Avoid over-mocking, which creates brittle tests that break when implementation details change. Tests that verify every method call with specific arguments couple tests to implementation rather than behavior. When refactoring changes how a use case achieves its goal without changing the outcome, tests shouldn't break. Focus on testing observable behavior and outcomes rather than internal implementation details. For example, test that an order is created with correct items and total, not that the repository's add method is called with specific arguments in a specific order.

Mock setup complexity indicates design problems. If you need dozens of lines to set up mocks for a single test, your class probably has too many dependencies or responsibilities. This is a code smell suggesting that you should refactor to reduce coupling. Domain-centric architecture naturally limits this problem because the dependency rule prevents excessive coupling. Domain objects have no dependencies to mock, and application services typically depend on a few repositories and domain services. If you find yourself creating complex mock setups, reconsider your design.

Test double libraries like Moq, NSubstitute, and FakeItEasy provide powerful features for creating mocks and stubs. Moq is the most popular in the C# ecosystem, offering a fluent API for setting up method returns and verifying calls. NSubstitute provides a more concise syntax that some developers prefer. FakeItEasy offers a different approach with a focus on discoverability. Choose one library and use it consistently across your test suite. Here's a comparison of syntax for the same test using different libraries:

```
// Moq
var mock = new Mock<IOrderRepository>();
mock.Setup(r => r.GetByIdAsync(123))
    .ReturnsAsync(order);
mock.Verify(r => r.AddAsync(It.IsAny<Order>()),
Times.Once);

// NSubstitute
var substitute = Substitute.For<IOrderRepository>();
substitute.GetByIdAsync(123).Returns(order);
substitute.Received(1).AddAsync(Arg.Any<Order>());

// FakeItEasy
var fake = A.Fake<IOrderRepository>();
A.CallTo(() => fake.GetByIdAsync(123)).Returns(order);
A.CallTo(() =>
fake.AddAsync(A<Order>._)).MustHaveHappenedOnceExactly();
```

Consider using the builder pattern for complex test doubles. When a test double needs significant setup or configuration, encapsulate that complexity in a builder class. This makes tests more readable and reusable. For example, a repository builder might provide methods for configuring what entities to return for different queries, making it easy to set up various test scenarios without repeating setup code. Builders also make it easier to maintain tests when interfaces change because you update the builder once rather than every test.

15 Chapter 14: Practical Implementation Guide

Moving from theory to practice represents the most challenging phase of adopting domain-centric architecture. You've learned about layers, dependencies, aggregates, and patterns throughout this book. Now comes the critical question: how do you actually structure your solution, organize your projects, and make these concepts work in real C# applications? This chapter bridges the gap between architectural principles and concrete implementation, providing you with actionable guidance for building maintainable systems.

The transition to domain-centric architecture requires careful planning and deliberate decisions about project organization. Your solution structure directly impacts how easily developers can navigate your codebase, understand dependencies, and maintain the architectural boundaries you've established. A well-organized solution makes the Dependency Rule visible and enforceable, while a poorly structured one invites violations and coupling. The choices you make about project references, namespaces, and folder hierarchies either reinforce or undermine your architectural goals.

Many teams face the challenge of migrating existing applications rather than starting fresh with greenfield projects. Legacy codebases often contain years of accumulated technical debt, tightly coupled components, and architectural patterns that directly conflict with domain-centric principles. Understanding how to incrementally refactor these systems without disrupting business operations requires strategy, patience, and practical techniques. Additionally, even experienced developers encounter common pitfalls when implementing clean architecture for the first time. Recognizing these traps early helps you avoid costly mistakes and maintain architectural integrity as your application grows.

15.1 Organizing Your Solution Structure

Your solution structure serves as the physical manifestation of your architectural layers. Each project in your solution should represent a distinct architectural concern with clear responsibilities and dependencies. A typical domain-centric solution contains four to six projects, each mapping to a specific layer or cross-cutting concern. The domain layer sits at the center with zero external dependencies, while outer layers reference inner ones but never the reverse. This structure makes dependency violations immediately visible through compilation errors.

Start with the core domain project, which contains your entities, value objects, domain services, and aggregate roots. Name this project something like `YourApp.Domain` or `YourApp.Core`. This project should reference only standard .NET libraries and perhaps a few carefully selected packages for domain-specific functionality like validation attributes. Avoid any references to Entity Framework, ASP.NET, or other infrastructure concerns. Your domain project defines interfaces for repositories and external services but never implements them. This keeps your business logic pure and testable without any infrastructure dependencies.

The application layer project, typically named `YourApp.Application`, sits just outside the domain and orchestrates use cases. This project references the domain project and contains your application services, command and query handlers, DTOs, and mapping logic. You might include packages like MediatR for implementing the mediator pattern or FluentValidation for input validation. The application layer defines interfaces for external concerns it needs, such as email services or file storage, but doesn't implement them. This maintains the inward dependency flow while allowing the application layer to coordinate complex operations across multiple aggregates.

Your infrastructure project, named `YourApp.Infrastructure`, implements all the interfaces defined in your domain and application layers. This is where Entity Framework DbContext lives, along with repository implementations, external service integrations, and data access logic. The infrastructure project references both domain and application projects, implementing their contracts with concrete technology choices. You'll include packages like Entity Framework Core, Dapper, or HTTP client libraries here. Keep your infrastructure implementations focused on technical concerns, delegating all business logic to the domain layer.

The presentation layer varies based on your application type. For a Blazor application, create a project named `YourApp.Web` or `YourApp.Blazor`. For an API, use `YourApp.Api`. This project references the application and infrastructure projects, configuring dependency injection and wiring everything together. Your presentation layer contains controllers, Razor components, view models, and presentation-specific logic. It translates user interactions into application commands and queries, then formats responses for display. Keep this layer thin, pushing any complex logic down into the application or domain layers.

Consider adding a shared kernel project for cross-cutting concerns used across multiple layers. Name it `YourApp.SharedKernel` and include base classes, common interfaces, and utility functions that don't belong to any specific layer. This might contain base entity classes, result types, or common value objects. Be cautious about what you place here—the shared kernel should contain only truly universal concepts. Overusing this project can lead to a dumping ground for miscellaneous code that should live in proper layers.

Here's a complete solution structure example with project references:

```
YourApp.sln
├── src/
│   ├── YourApp.Domain/
│   │   ├── Entities/
│   │   ├── ValueObjects/
│   │   ├── Aggregates/
│   │   ├── DomainServices/
│   │   ├── Interfaces/
│   │   └── Events/
│   ├── YourApp.Application/
│   │   ├── Commands/
│   │   ├── Queries/
│   │   ├── DTOs/
│   │   ├── Interfaces/
│   │   ├── Mappings/
│   │   └── Services/
│   ├── YourApp.Infrastructure/
│   │   ├── Persistence/
│   │   ├── Repositories/
│   │   ├── ExternalServices/
│   │   └── Configuration/
│   ├── YourApp.Web/
│   │   ├── Pages/
│   │   ├── Components/
│   │   ├── ViewModels/
│   │   └── Program.cs
│   └── YourApp.SharedKernel/
│       ├── Base/
│       └── Common/
└── tests/
    ├── YourApp.Domain.Tests/
    ├── YourApp.Application.Tests/
    ├── YourApp.Infrastructure.Tests/
    └── YourApp.Integration.Tests/
```

Enforce architectural boundaries through project references and namespace organization. Your domain project should have zero project references except perhaps to the shared kernel. The application project references only the domain. Infrastructure references both domain and application. The presentation layer references application and infrastructure for dependency injection configuration. Use tools like NDepend or custom Roslyn analyzers to detect and prevent dependency violations automatically. Configure your CI/CD pipeline to fail builds that violate these rules, making architectural boundaries enforceable rather than merely aspirational.

Organize folders within each project to reflect architectural patterns and make code discoverable. In your domain project, group entities by aggregate, placing the aggregate root and its related entities together. Create separate folders for domain services, specifications, and domain events. In the application layer, organize by feature or use case rather than by technical pattern. A folder named `Orders` might contain `CreateOrderCommand`, `CreateOrderHandler`, and `OrderDto` together. This feature-based organization makes it easier to understand and modify related functionality without jumping between distant folders.

Consider using vertical slice architecture within your application layer for complex applications. Instead of separating commands, queries, and handlers into different folders, group everything related to a specific feature together. This reduces coupling between features and makes it easier to modify or remove functionality without affecting unrelated code. Each slice becomes a self-contained unit with its own commands, queries, validators, and DTOs. This approach works particularly well with

MediatR and aligns naturally with microservices if you later decide to extract features into separate services.

Document your solution structure and architectural decisions in a README file at the solution root. Explain the purpose of each project, the dependency rules, and any conventions developers should follow. Include diagrams showing the layer relationships and data flow through the application. This documentation helps new team members understand the architecture quickly and serves as a reference when making decisions about where new code belongs. Update this documentation as your architecture evolves, treating it as a living guide rather than a one-time artifact.

15.2 Migrating Existing Applications

Migrating a legacy application to domain-centric architecture requires a pragmatic, incremental approach rather than a complete rewrite. Big-bang rewrites rarely succeed and often introduce more problems than they solve. Instead, adopt the strangler fig pattern, gradually replacing old code with new architecture while keeping the application functional throughout the transition. Start by identifying bounded contexts within your existing application, then migrate one context at a time. This allows you to deliver value continuously while improving the architecture incrementally.

Begin your migration by creating the new solution structure alongside your existing code. Add the domain, application, and infrastructure projects without immediately moving any code. This establishes the target architecture and allows you to start building new features using clean architecture while maintaining the legacy system. Configure your existing application to reference these new projects, creating a hybrid architecture during the transition period. New features go into the new structure, while legacy code remains untouched until you're ready to refactor it.

Identify a small, relatively isolated feature or bounded context to migrate first. Choose something with clear boundaries, moderate complexity, and business value. Avoid starting with the most complex or most critical feature—you need early wins to build momentum and learn the migration process. Extract the business logic from this feature into domain entities and value objects in your new domain project. This often reveals how much business logic has leaked into controllers, views, or data access code. Consolidate this scattered logic into proper domain models, enforcing invariants and encapsulating behavior.

Create an anti-corruption layer to interface between your new domain-centric code and the legacy system. This layer translates between the old data structures and your new domain models, preventing legacy concerns from polluting your clean architecture. Implement adapters that convert legacy DTOs or database entities into your domain objects and vice versa. This isolation allows your new code to follow clean architecture principles while still interacting with the existing system. As you migrate more features, the anti-corruption layer shrinks until it eventually disappears.

Here's an example of an anti-corruption layer adapter:

```
public class LegacyOrderAdapter
{
    private readonly LegacyOrderRepository _legacyRepo;

    public LegacyOrderAdapter(LegacyOrderRepository 
legacyRepo)
    {
        _legacyRepo = legacyRepo;
    }

    public Order ConvertToDomainModel(LegacyOrderDto 
legacyOrder)
    {
        // Translate legacy structure to domain model
        var customerId = new 
CustomerId(legacyOrder.CustomerID);
        var orderItems = legacyOrder.Items
            .Select(item => new OrderItem(
                new ProductId(item.ProductID),
                new Money(item.Price, item.Currency),
                item.Quantity))
            .ToList();

        return Order.Create(
            new OrderId(legacyOrder.OrderID),
            customerId,
            orderItems,
            legacyOrder.OrderDate);
    }

    public LegacyOrderDto ConvertToLegacyModel(Order order)
    {
        // Translate domain model back to legacy structure
        return new LegacyOrderDto
        {
            OrderID = order.Id.Value,
            CustomerID = order.CustomerId.Value,
            OrderDate = order.OrderDate,
            Items = order.Items.Select(item => new
LegacyOrderItemDto
            {
                ProductID = item.ProductId.Value,
                Price = item.Price.Amount,
                Currency = item.Price.Currency,
                Quantity = item.Quantity
            }).ToList()
        };
    }
}
```

Tackle data access migration carefully, as this often represents the most challenging aspect of the transition. Your legacy application likely uses Entity Framework or another ORM directly in controllers or services, with database concerns scattered throughout the codebase. Start by creating repository interfaces in your domain project that express what your domain needs without specifying how data is retrieved. Implement these repositories in your infrastructure project, initially wrapping your existing data access code. This provides a clean interface while reusing existing database logic, allowing you to refactor the implementation later without affecting the domain.

Consider running dual-write scenarios during migration where you write to both the old and new systems temporarily. This allows you to validate that your new implementation produces the same results as the legacy code before fully cutting over. Compare outputs, run both code paths in parallel, and gradually increase confidence in the new implementation. Once you've verified correctness, remove the legacy code path and rely entirely on the new architecture. This technique reduces risk and provides a safety net during critical migrations.

Address testing gaps as you migrate each feature. Legacy code often lacks comprehensive tests, making refactoring risky. Before migrating a feature, add characterization tests that document current behavior, even if that behavior is flawed. These tests act as a safety net, alerting you if your migration changes existing functionality unexpectedly. Once you've migrated the feature to clean architecture, replace characterization tests with proper unit and integration tests that verify correct behavior. Your test coverage should improve with each migration, making future changes safer and easier.

Manage database schema evolution carefully during migration. Your legacy database likely contains denormalized data, missing constraints, and schema designs that don't align with your domain model. Don't try to fix everything at once—use database views, stored procedures, or mapping configurations to bridge the gap between your domain model and the existing schema. Plan schema changes incrementally, coordinating with your code migrations. Use Entity Framework migrations or a database migration tool like DbUp to version and apply schema changes safely. Consider maintaining backward compatibility during the transition, allowing old and new code to coexist.

Communicate the migration strategy clearly to your team and stakeholders. Explain that this is a gradual process that will take months or even years depending on application size. Set realistic expectations about the pace of migration and the benefits that will accrue over time. Celebrate milestones as you complete each bounded context migration. Track metrics like test coverage, code complexity, and bug rates to demonstrate improvement. This helps maintain support for the migration effort and justifies the investment in architectural improvement.

Create migration guidelines and coding standards specific to your transition period. Document patterns for the anti-corruption layer, naming conventions for new projects, and decision criteria for when to refactor versus rewrite. Establish code review practices that ensure new code follows clean architecture principles while legacy code is gradually improved. Consider pair programming or mob programming sessions for complex migrations, spreading knowledge across the team and maintaining consistency. These practices help the entire team contribute to the migration effectively.

15.3 Common Pitfalls and How to Avoid Them

One of the most common pitfalls is creating anemic domain models that contain only properties without behavior. Developers accustomed to transaction script or CRUD-based architectures often create entities that are merely data containers, placing all logic in application services. This defeats the purpose of domain-centric architecture. Your entities should encapsulate business rules and enforce invariants through methods rather than exposing setters. If your domain layer looks like a collection of DTOs with getters and setters, you've missed the point. Push behavior into your domain models, making them rich with business logic.

Over-engineering represents another frequent mistake, especially when developers first encounter clean architecture. The temptation to create abstractions for everything, implement every pattern, and build elaborate frameworks can lead to unnecessary complexity. Not every application needs CQRS, event sourcing, and a full-blown domain event system. Start simple and add complexity only when you have concrete requirements that justify it. A small application might need only basic layers without sophisticated patterns. Let your architecture evolve based on actual needs rather than anticipated future requirements.

Violating the Dependency Rule undermines your entire architecture but happens surprisingly often. Developers might reference the infrastructure project from the domain layer to use a specific ORM feature, or reference the presentation layer from the application layer to access a view model. These shortcuts seem harmless initially but create coupling that makes your code harder to test and maintain. Use dependency injection and interfaces to maintain proper dependency flow. Configure your build system to detect and prevent these violations automatically. Treat dependency rule violations as serious architectural defects that must be fixed immediately.

Misunderstanding aggregate boundaries leads to performance problems and data consistency issues. Developers often create aggregates that are too large, loading excessive data and creating contention. Alternatively, they create aggregates that are too small, requiring complex coordination across multiple aggregates for simple operations. Remember that aggregates define consistency boundaries—everything inside an aggregate must be consistent after each operation. If you find yourself frequently loading multiple aggregates to complete a single business operation, your boundaries might be wrong. Redesign your aggregates based on business invariants and transactional requirements.

Here's an example of an aggregate that's too large and should be split:

Improper use of repositories causes confusion and violates architectural principles. Some developers create generic repositories with methods like `GetAll()`, `Find(Expression<Func<T, bool>>)`, or `Update(T entity)` that expose query capabilities to the application layer. This leaks data access concerns into higher layers and makes it difficult to optimize queries. Instead, create specific repository methods that express domain operations: `GetActiveOrdersForCustomer(CustomerId)`, `FindProductsByCategory(CategoryId)`. These methods encapsulate query logic in the infrastructure layer where it belongs.

Neglecting integration tests while focusing solely on unit tests leaves gaps in your test coverage. Unit tests verify that individual components work correctly in isolation, but integration tests ensure that layers interact properly and that your infrastructure implementations work with real databases and external services. Write integration tests that exercise your repositories against a real database, test your API endpoints end-to-end, and verify that dependency injection configuration is correct. Use tools like Testcontainers to run integration tests against real database instances in Docker containers, ensuring your tests reflect production behavior.

Mixing concerns across layers happens when developers don't clearly understand each layer's responsibilities. Application services might contain data access code, domain entities might reference DTOs, or controllers might contain business logic. Establish clear guidelines for what belongs in each layer:

- **Domain Layer:** Business rules, invariants, domain logic, entity behavior
- **Application Layer:** Use case orchestration, transaction management, DTO mapping
- **Infrastructure Layer:** Data access, external service integration, technical implementations
- **Presentation Layer:** User interface, input validation, response formatting

Ignoring performance implications of architectural patterns can lead to slow applications. Loading entire aggregates when you only need a few fields, making multiple database round trips for simple queries, or using inefficient ORM configurations all impact performance. Domain-centric architecture doesn't require sacrificing performance. Use read models and query objects for complex queries, implement caching strategically, and optimize your Entity Framework configurations. Consider CQRS for read-heavy scenarios, using different models for commands and queries. Profile your application regularly and optimize hot paths while maintaining architectural integrity.

Failing to establish team conventions and coding standards leads to inconsistent implementations across your codebase. One developer might place validation in the domain layer while another puts it in the application layer. Some might use rich domain models while others create anemic entities. Document your architectural decisions, create code templates, and conduct regular architecture reviews. Use tools like ArchUnit or custom Roslyn analyzers to enforce conventions automatically. Pair programming and code reviews help spread knowledge and maintain consistency. Your architecture is only as good as your team's understanding and adherence to its principles.

Premature optimization of the architecture itself wastes time and adds complexity without clear benefits. You might spend weeks designing the perfect aggregate structure or implementing sophisticated patterns before writing any actual business logic. Start with the simplest implementation that follows clean architecture principles, then refactor as you learn more about the domain. Your understanding of the business domain will evolve as you build the application. Allow your architecture to evolve with that understanding rather than trying to design everything perfectly upfront. Refactoring is a normal and healthy part of software development.

Finally, treating architecture as a one-time decision rather than an ongoing practice leads to architectural decay. Your initial structure might be clean, but without continuous attention, dependencies creep in, abstractions leak, and the architecture degrades. Establish regular architecture reviews, refactor proactively, and address technical debt before it accumulates. Make architectural quality a part of your definition of done. Educate new team members about architectural principles and the reasoning behind your design decisions. Architecture is not something you do once at the beginning—it's a continuous practice that requires vigilance and commitment throughout the application's lifetime.

16 CONCLUSION

You've journeyed through the principles, patterns, and practices that define domain-centric architecture in C#. From understanding the fundamental problems with traditional layered approaches to implementing sophisticated patterns like aggregates and repositories, you now possess the knowledge to build applications that place business logic at their core. This architectural approach isn't just about organizing code—it's about creating systems that reflect how your business actually works, making them easier to understand, maintain, and evolve over time.

The transition from database-centric or framework-centric thinking to domain-centric architecture represents a significant shift in perspective. Rather than letting technical concerns drive your design decisions, you've learned to let your domain model guide the structure of your application. This inversion of priorities leads to code that speaks the language of your business, where developers and domain experts can collaborate effectively because the code mirrors the concepts they discuss daily. The technical infrastructure becomes what it should be: a supporting player that enables your domain logic to shine.

Throughout this book, you've seen how Clean Architecture and Onion Architecture provide complementary approaches to achieving the same goal—protecting your domain from external concerns. You've explored SOLID principles not as abstract academic concepts but as practical tools that make your code more flexible and testable. You've learned to design aggregates that enforce business rules, implement repositories that abstract persistence, and leverage dependency injection to achieve loose coupling. These aren't isolated techniques but interconnected practices that work together to create maintainable systems.

As you move forward, remember that architecture is a journey, not a destination. The patterns and principles you've learned provide a foundation, but every application presents unique challenges that require thoughtful adaptation. The goal isn't to apply these patterns dogmatically but to understand the problems they solve and use them judiciously. Sometimes a simpler approach is better; other times, the full power of domain-centric architecture is necessary. Your growing experience will help you make these decisions with confidence.

16.1 Key Takeaways

The **Dependency Rule** stands as the cornerstone of domain-centric architecture. Dependencies must always point inward toward the domain, never outward toward infrastructure or presentation concerns. This rule ensures that your business logic remains independent of databases, frameworks, and UI technologies. When you respect this rule, you can change your persistence mechanism from SQL Server to MongoDB, swap your web framework from MVC to Blazor, or replace external services without touching your domain code. This independence is what makes your applications truly maintainable over the long term.

Your domain layer should contain rich, behavior-focused entities rather than anemic data containers. The difference between these approaches is profound:

- **Rich domain models** encapsulate business logic within entities, ensuring rules are enforced consistently
- **Anemic models** push logic into service layers, scattering business rules across multiple classes
- **Value objects** represent concepts without identity, making your model more expressive and type-safe

- **Domain services** handle operations that don't naturally belong to a single entity
- **Domain events** enable loose coupling between aggregates while maintaining consistency

Aggregates define consistency boundaries within your domain model, and designing them correctly is crucial for maintaining data integrity. An aggregate is a cluster of objects treated as a single unit for data changes, with one entity serving as the aggregate root. All external access to the aggregate must go through this root, which enforces invariants and business rules. The key insight is that aggregates should be as small as possible while still maintaining consistency—typically, this means one aggregate per transaction boundary. Oversized aggregates lead to concurrency issues and performance problems, while undersized aggregates fail to protect business rules.

The Repository Pattern provides a collection-like interface for accessing aggregates, abstracting away persistence details from your domain layer. Repositories should work with aggregate roots, not individual entities within aggregates. This means you don't create repositories for every entity—only for aggregates that need to be retrieved independently. Your repository interfaces belong in the domain layer, expressing what operations are needed in domain terms, while implementations live in the infrastructure layer, handling the technical details of data access. This separation allows you to test domain logic without touching a database.

SOLID principles aren't just theoretical guidelines—they're practical tools that directly impact your code's maintainability:

- **Single Responsibility** keeps classes focused and easier to understand and modify
- **Open-Closed** allows extension without modification, reducing the risk of breaking existing functionality

- **Liskov Substitution** ensures derived classes can replace base classes without surprises
- **Interface Segregation** prevents clients from depending on methods they don't use
- **Dependency Inversion** makes high-level modules independent of low-level implementation details

Dependency Injection inverts the flow of control, allowing your classes to declare their dependencies rather than creating them. This inversion enables loose coupling, making your code testable and modular. In C#, the built-in dependency injection container handles object creation and lifetime management, but the real power comes from designing your classes to depend on abstractions rather than concrete implementations. When you inject `IOrderRepository` instead of `SqlOrderRepository`, you can easily substitute implementations for testing or when requirements change. The composition root—typically your application's startup class—is where you wire up these dependencies.

Entity Framework Core can be effectively used in domain-centric architectures when properly isolated in the infrastructure layer. The key is to prevent EF concerns from leaking into your domain. Use the Fluent API in separate configuration classes rather than attributes on domain entities. Map your rich domain models to database tables without compromising their behavior. Implement repositories that translate between EF's `DbSet` operations and your domain's collection-like interfaces. When done correctly, your domain layer remains completely unaware that EF exists, and you could swap it for Dapper or another ORM without changing domain code.

Testing becomes significantly easier with domain-centric architecture because dependencies flow inward and are expressed through interfaces. Your domain layer can be unit tested without any infrastructure—no databases, no file systems, no external services. Application services can be tested with mocked repositories. Integration tests can verify that your infrastructure implementations work correctly with real databases. This testing pyramid—many unit tests, fewer integration tests, even fewer end-to-end tests—becomes natural when your architecture supports it. The investment in proper architecture pays dividends every time you write a test that runs in milliseconds instead of seconds.

Modern frameworks like Blazor benefit tremendously from clean architecture principles. By separating your presentation logic from business logic, you create components that are focused on rendering and user interaction while delegating business operations to application services. This separation means your business logic can be reused across different presentation technologies—the same application services can support both Blazor WebAssembly and Blazor Server, or even a completely different UI framework. State management becomes clearer when you distinguish between UI state (which component is expanded) and domain state (what data exists in your system).

16.2 Your Next Steps

Start by applying these principles to a small, well-defined project rather than attempting to refactor your entire enterprise application at once. Choose a bounded context—a specific area of your business domain—and implement it using the patterns you've learned. This might be an order processing system, a user management module, or an inventory tracking feature. Keep the scope manageable so you can complete it and learn from the experience. You'll encounter challenges and make mistakes, but that's how you develop the intuition needed to apply these patterns effectively in larger contexts.

Create a reference implementation that serves as your team's architectural template. This implementation should include:

- A properly structured solution with separate projects for domain, application, infrastructure, and presentation layers
- Example entities, value objects, and aggregates that demonstrate rich domain modeling
- Repository interfaces in the domain layer with EF Core implementations in infrastructure
- Application services that orchestrate use cases without containing business logic
- Dependency injection configuration showing how to wire up the layers
- Unit tests for domain logic and integration tests for repositories
- Documentation explaining the architectural decisions and patterns used

If you're working with an existing codebase, resist the urge to rewrite everything immediately. Instead, adopt a strangler fig pattern where you gradually introduce domain-centric architecture alongside existing code. When you need to add new features, implement them using clean architecture principles in new projects or namespaces. When you need to modify existing features, consider refactoring them to align with domain-centric patterns if the change is substantial enough to justify the effort. Over time, the new architecture will grow and eventually replace the old, but you'll deliver value continuously rather than embarking on a risky big-bang rewrite.

Invest time in understanding your business domain deeply. Domain-centric architecture is only as good as your domain model, and your domain model is only as good as your understanding of the business. Schedule regular conversations with domain experts—the people who understand the business rules, workflows, and terminology. Use techniques like Event Storming to discover domain events, commands, and aggregates collaboratively. Read business documentation, observe users working with current systems, and ask questions about edge cases and business rules. The technical patterns are important, but they're in service of modeling the domain accurately.

Practice test-driven development to reinforce the architectural principles you've learned. Writing tests first forces you to think about dependencies and interfaces before implementation details. When you write a test for a domain entity, you'll naturally focus on behavior rather than data. When you write a test for an application service, you'll naturally inject repository interfaces rather than concrete implementations. TDD and domain-centric architecture complement each other beautifully—TDD encourages the same loose coupling and separation of concerns that clean architecture requires. Start with simple domain logic tests and gradually expand to more complex scenarios.

Build a personal library of reusable components that support domain-centric architecture. This might include:

- Base classes for entities and value objects with common functionality
- Generic repository interfaces and implementations
- Result types for handling success and failure without exceptions
- Domain event infrastructure for publishing and handling events
- Specification pattern implementations for complex queries
- Validation frameworks that work with your domain model

Join communities focused on domain-driven design and clean architecture. Online forums, local meetups, and conferences provide opportunities to learn from others' experiences and share your own. The DDD community is particularly active and welcoming, with practitioners eager to discuss aggregate design, bounded contexts, and domain modeling challenges. You'll encounter different perspectives and approaches that will refine your understanding. Some practitioners favor CQRS and event sourcing, others prefer simpler approaches—exposure to these variations helps you understand when each approach is appropriate.

Measure the impact of your architectural decisions on concrete metrics that matter to your organization. Track how long it takes to implement new features before and after adopting domain-centric architecture. Monitor bug rates in domain logic versus infrastructure code. Measure test coverage and test execution time. Document how architectural decisions affected your ability to respond to changing requirements. These metrics help you demonstrate the value of good architecture to stakeholders who may not understand technical details but care deeply about delivery speed, quality, and maintainability. Architecture isn't free—it requires investment—so being able to show returns on that investment is important.

Experiment with complementary patterns and practices that enhance domain-centric architecture. CQRS (Command Query Responsibility Segregation) separates read and write models, which can simplify complex domains. Event sourcing stores state changes as events rather than current state, providing audit trails and temporal queries. Specification pattern encapsulates query logic in reusable, composable objects. Mediator pattern decouples request senders from handlers, reducing direct dependencies. Not every project needs these patterns, but understanding them expands your architectural toolkit. Try implementing them in side projects or proof-of-concept applications to understand their benefits and costs.

16.3 Continuing Your Architectural Journey

Architecture is a discipline that requires continuous learning because technology, business needs, and best practices evolve constantly. The principles you've learned—dependency inversion, separation of concerns, domain modeling—remain relevant across decades, but their application changes with new frameworks, languages, and paradigms. Stay current by reading books, following thought leaders, and experimenting with emerging patterns. Eric Evans' "Domain-Driven Design" remains essential reading for understanding strategic design and bounded contexts. Robert Martin's "Clean Architecture" provides deeper insights into the dependency rule and architectural boundaries. Vernon Vaughn's

"Implementing Domain-Driven Design" offers practical guidance on tactical patterns.

Develop your ability to recognize when domain-centric architecture is appropriate and when simpler approaches suffice. Not every application needs the full ceremony of Clean Architecture with separate projects for each layer. A simple CRUD application with minimal business logic might be over-engineered with aggregates and repositories. A prototype or proof-of-concept might prioritize speed over maintainability. However, applications with complex business rules, multiple developers, long expected lifespans, or frequent requirement changes benefit enormously from domain-centric architecture. Learning to assess these factors and make informed architectural decisions is a skill that develops with experience.

Understand that architecture exists on a spectrum, not as a binary choice between "clean" and "messy." You might start with a simple layered architecture and introduce domain-centric patterns gradually as complexity grows. You might apply clean architecture rigorously in your core domain while using simpler approaches in supporting subdomains. You might use full aggregates for some parts of your model while accepting simpler entities elsewhere. Pragmatism matters—the goal is maintainable software that delivers business value, not perfect adherence to patterns. As you gain experience, you'll develop intuition about where to invest in architectural sophistication and where to keep things simple.

Mentor others in domain-centric architecture principles, as teaching reinforces your own understanding and spreads good practices throughout your organization. Conduct code reviews that focus on architectural concerns—are dependencies pointing in the right direction? Is business logic in the domain layer or scattered in services? Are aggregates properly designed? Pair program with less experienced developers on architectural decisions, explaining your reasoning as you work. Create internal documentation and presentations that explain your team's architectural standards. When you help others understand these principles, you create a culture where good architecture becomes the norm rather than the exception.

Balance architectural purity with practical delivery constraints. In the real world, you'll face deadlines, budget limitations, and technical debt from previous decisions. Sometimes you'll need to make compromises—perhaps putting a small piece of business logic in an application service because refactoring the aggregate would take too long, or coupling to a specific framework because the abstraction cost isn't justified. Make these compromises consciously, document them, and create technical debt items to address them later. The key is understanding when you're deviating from principles and why, rather than accidentally creating problems through ignorance.

Explore how domain-centric architecture applies to different types of applications beyond traditional web applications. Microservices architectures benefit from clean architecture within each service, with bounded contexts mapping to service boundaries. Desktop applications can use these patterns to separate business logic from UI frameworks like WPF or WinForms. Mobile applications built with Xamarin or MAUI can share domain and application layers across platforms while maintaining platform-specific presentation layers. Background services and console applications can leverage the same architectural principles. The patterns are technology-agnostic, which is precisely what makes them valuable.

Consider the organizational and team dynamics that support or hinder good architecture. Architecture isn't just a technical concern—it's influenced by team structure, communication patterns, and organizational culture. Conway's Law suggests that systems mirror the communication structures of the organizations that build them[6]. If your teams are organized around technical layers (database team, UI team, API team), you'll struggle to implement domain-centric architecture effectively. Teams organized around business capabilities or bounded contexts naturally produce better domain models. Advocate for team structures that align with your architectural goals.

Recognize that architecture is ultimately about managing complexity and enabling change. Every pattern, principle, and practice you've learned serves these goals. The Dependency Rule manages complexity by preventing tangled dependencies. SOLID principles enable change by making code flexible and extensible. Aggregates manage complexity by defining clear consistency boundaries. Repositories enable change by abstracting persistence mechanisms. When you make architectural decisions, ask yourself: does this reduce complexity? Does this make future changes easier? If the answer is yes, you're probably on the right track. If the answer is no, reconsider your approach.

Your journey with domain-centric architecture doesn't end with this book—it's just beginning. You now have the knowledge and tools to build applications that place business logic at their core, creating systems that are testable, maintainable, and aligned with business needs. You understand how to organize code into layers with clear dependencies, design rich domain models that enforce business rules, and integrate with infrastructure concerns without compromising your domain. Most importantly, you understand why these practices matter and the problems they solve. Take this knowledge, apply it thoughtfully, learn from your experiences, and continue growing as a software architect. The applications you build will be better for it, and the developers who maintain them—including your future self—will thank you.

[6] Conway, Melvin E. (1968). How Do Committees Invent? Datamation, 14(4), 28-31.