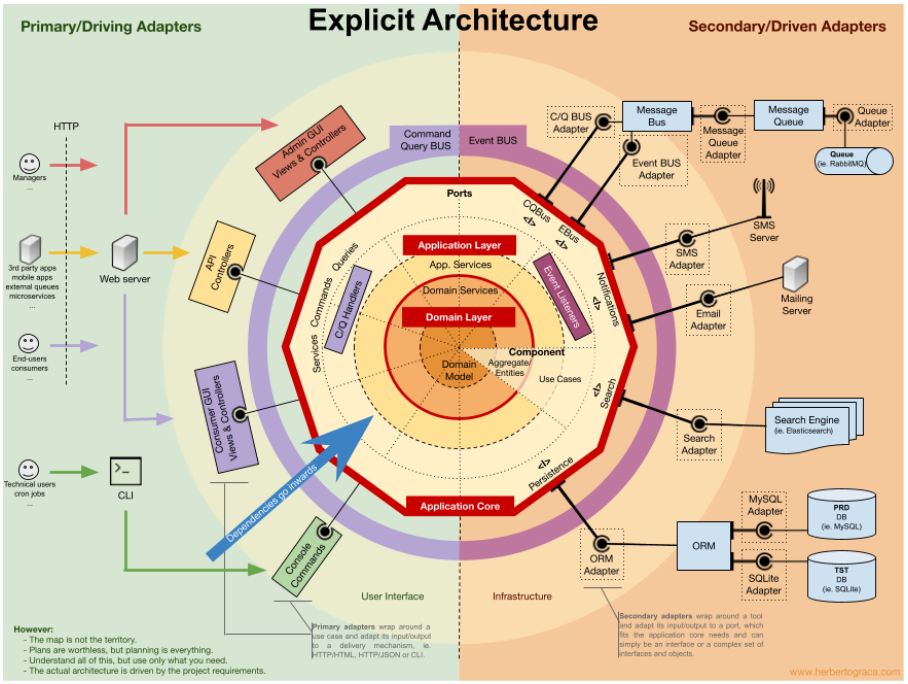
3. Hexagon Architecture

Introduction

Hexagonal architecture, also called **Ports and Adapters** or **Clean Architecture**, was proposed by Alistair Cockburn in 2005. Imagine your business logic as a fortress in the center, completely isolated from the outside world. To communicate with it, you must go through specific gates (ports) and use authorized translators (adapters).

This pattern emerged to solve a critical problem with layered architecture: **business logic becomes contaminated by infrastructure concerns**. In traditional layered systems, your core business rules often depend on databases, web frameworks, or external APIs. Hexagonal architecture flips this around—your business logic depends on **nothing**, and everything depends on it.

The "hexagon" shape is just a visual metaphor suggesting the core has multiple sides for different interactions. The actual number of sides doesn't matter—you might have 4, 6, or 20 ports depending on your needs.



References

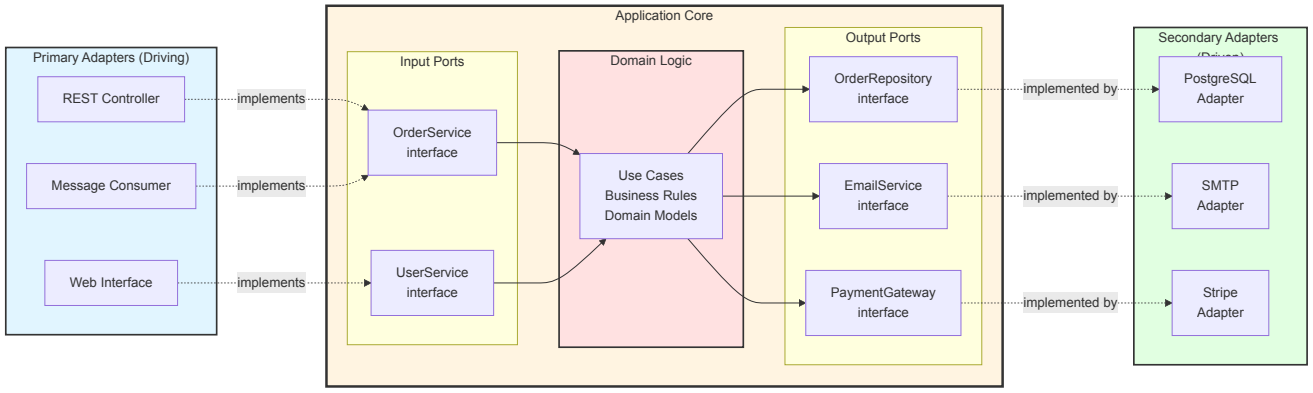
1. Cockburn, A., & Garrido de Paz, J. M. (2024). *Hexagonal architecture explained*
2. Vieira, D. (2022). *Designing hexagonal architecture with Java.* Packt Publishing.
3. Cockburn, A. (2005). *The Hexagonal (Ports & Adapters) architecture*

Explanation

Core Concept

The fundamental idea is **dependency inversion**: your business logic (the core) defines interfaces (ports) that describe what it needs. External components (adapters) implement these interfaces. This way, the core never knows about databases, web frameworks, or UI technologies.

Main Components



1. Application Core (The Hexagon)

- Contains your business logic and domain models
- Has **zero dependencies** on external frameworks or libraries
- Defines what it needs through interfaces (ports)
- Completely testable without any infrastructure
- Examples: Order processing rules, user authentication logic, pricing algorithms

2. Ports (Interfaces) Two types of ports:

Primary/Driving Ports (Input Ports)

- Define how the outside world **uses** your application
- Examples: `OrderService`, `UserRepository`, `PaymentProcessor`
- These are interfaces that your core implements

Secondary/Driven Ports (Output Ports)

- Define what your application **needs** from the outside world
- Examples: `DatabasePort`, `EmailPort`, `PaymentGatewayPort`
- These are interfaces that adapters must implement

3. Adapters Concrete implementations that connect the real world to your ports:

Primary/Driving Adapters (Left side)

- Trigger actions in your application
- Examples: REST controllers, GraphQL resolvers, CLI commands, message queue consumers
- They call your input ports

Secondary/Driven Adapters (Right side)

- Provide services your application needs
- Examples: PostgreSQL repository, SendGrid email adapter, Stripe payment adapter
- They implement your output ports

Key Principles

Dependency Inversion The core doesn't depend on adapters; adapters depend on the core. In traditional layered architecture, your business logic imports database libraries. In hexagonal architecture, your business logic defines an interface, and the database adapter implements it.

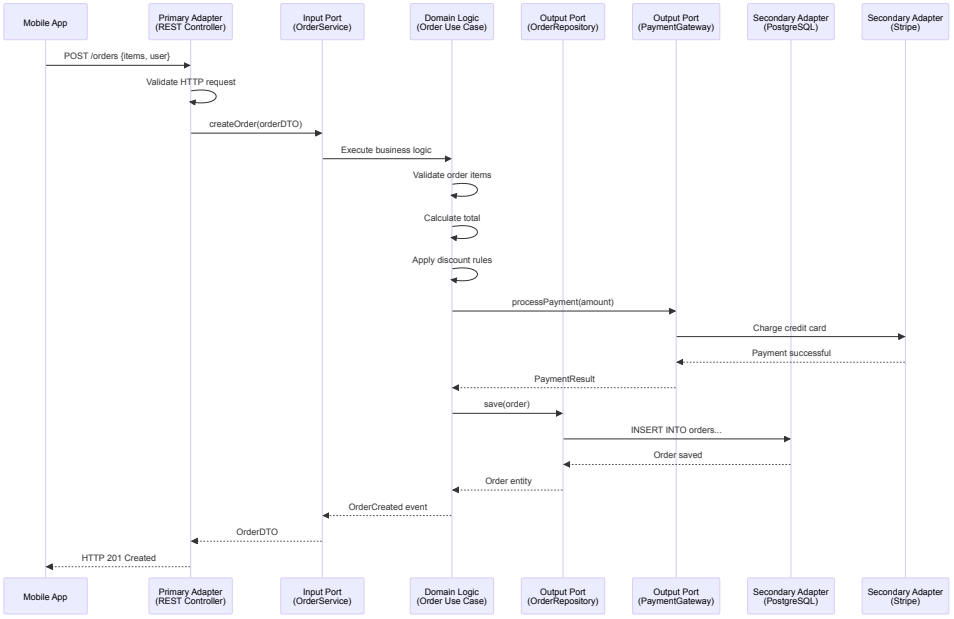
Technology Agnostic Core Your business logic has no idea whether it's running in a web application, a CLI tool, or a serverless function. It doesn't know if data is stored in PostgreSQL, MongoDB, or a file system.

Testability Without Infrastructure You can test your entire business logic with simple mock objects, no database or external services needed.

Symmetry There's no distinction between "top" and "bottom" layers. All external systems (UI, database, external APIs) are treated equally—they're all outside the hexagon.

Dataflow example

Let's see how placing an order flows through hexagonal architecture for an e-commerce order system:



Advantages

Complete Business Logic Isolation

- Your core domain is **pure business logic** with zero technical dependencies
- Can develop and test business rules without any infrastructure
- Business logic becomes truly portable across technologies

Maximum Testability

- Test business logic with simple unit tests (no mocks needed for infrastructure)
- Replace entire databases or external services with test implementations
- Test coverage can approach 100% for core logic

Technology Flexibility

- Swap databases without touching business logic (PostgreSQL → MongoDB)
- Support multiple UIs simultaneously (web + mobile + CLI + API)
- Replace external services easily (SendGrid → AWS SES for email)
- Upgrade or change frameworks without rewriting core logic

Clear Boundaries and Contracts

- Ports define explicit contracts between core and infrastructure
- New team members understand exactly what each component does
- Architectural decisions are explicit, not implicit

Parallel Development

- Frontend and backend teams can work completely independently
- Teams work against interface contracts, not implementations
- Easy to mock dependencies during development

Long-term Maintainability

- Business logic doesn't rot as frameworks change
- Technical debt stays in adapters, not core
- Easier to refactor because dependencies flow one way

Disadvantages

Higher Complexity and Learning Curve

- More abstractions to understand (ports, adapters, dependency inversion)
- Requires understanding of SOLID principles and design patterns
- Can be overwhelming for junior developers
- Over-engineering for simple CRUD applications

More Code to Write

- Every external interaction needs an interface and implementation
- More files and classes to manage
- Simple operations require more boilerplate
- Example: A database query needs: repository interface, implementation, domain models, DTOs

Initial Development Slowdown

- Takes longer to set up initially than layered architecture
- More upfront design decisions
- Have to think about boundaries and abstractions from day one

Potential for Over-Abstraction

- Easy to create too many layers of indirection
- Can lead to "interface explosion" with dozens of small interfaces
- Finding the right granularity for ports is challenging

Performance Overhead

- Additional abstraction layers can impact performance
- Data transformation between layers (Domain → DTO → HTTP response)
- More objects created and garbage collected

Harder to Understand Data Flow

- Following a request through the system requires jumping through many files
- Not as intuitive as top-to-bottom layered flow
- Debugging can be more complex

Team Alignment Required

- Everyone must understand and commit to the pattern
- Inconsistent application breaks the benefits

- Requires discipline to maintain boundaries