Ports And Adapters
(formerly Hexagonal Architecture)

Daniel Hinojosa

Designing Software with Intent

In this Presentation

- Defining Ports and Adapters Architecture
- Starting Lightweight
- Domain Driven Design
- Behavior Driven Development
- Domain Driven Design Components
- Designing With Modularity
- Folder Structure
- Test Driven Development
- End To End Testing
- Is it the same as the Onion Architecture?

Code and Slides are available at https://github.com/dhinojosa/gids-ports-adapters

Ports and Adapters

Ports and Adapters

- Formerly called **Hexagonal Architecture**
  - Focuses on separating domain logic from infrastructure
  - Places the domain at the center
  - Allows external systems (UIs, DBs, APIs, and even Ai) to plug in via adapters
  - Designed for testability, flexibility, and maintainability
- Introduced by Alistair Cockburn in **2005**
- There have been many refinements until recently

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

The Arrows Go In!

- “Source code dependencies must point only inward, toward higher-level policies” - Clean Architecture
- The Domain, discussed next, **doesn’t depend on anything**
- This is beneficial:
  - Loosely Coupled Architecture
  - Easy to swap out infrastructure
  - Promotes Testing
  - Clear Separation of Concerns

Domain Layer

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

What’s in the Domain?

- Core business logic - The rules and calculations that define how your system behaves
- Key business models - The central objects and data structures your business works with (not tied to database or UI)
- Pure functions or classes - Logic with no side effects, no I/O, just behavior
- Business decisions and policies- Logic that determines how the domain reacts to situations (e.g., should a discount apply?)
- No frameworks, no technical concerns - No HTTP, no SQL, no infrastructural annotations, no Spring — just plain logic

Application Layer

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

Application Layer

- Coordinates use cases of the system
- Defines input ports (e.g., `PlaceOrder`, `SendInvoice`). They may use the term use-cases
- Delegates to the domain layer to perform actual business logic
- Defines ports for external dependencies
- Contains no UI, database, or framework code explicitly only through the ports
- Focused on workflow, orchestration, and policies

Infrastructure Layer

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

Infrastructure Layer

- Contains technical details and external integrations
- Implements ports (e.g., database access, REST clients, message brokers)
- Connects the application to the outside world: DBs, APIs, files, queues
- Often uses frameworks like Spring, Quarkus, or your own hard wiring
- No business logic — purely infrastructure details, although using the domain objects
- Swappable without affecting core logic

What’s a Port?

What's a Port?

- A Port is an `interface` that defines how the system communicates with the outside world
  - It represents an application use case (input port) or a required dependency (output port)
  - Ports live in the application layer
  - They are technology-agnostic — no frameworks, annotations, or implementation details
  - Ports enable inversion of control: the domain declares what it needs, not how it's fulfilled
  - They allow adapters to plug in without coupling them to the core logic
- Ports define stable contracts for input/output, making the system modular and testable

The “In” Ports

The “In” Ports (Driving)

- An In Port represents an entry point into the application’s logic
- It defines a **use case** or operation the system exposes to the outside world
- Implemented in the application layer (e.g., a service or interactor)
- Called by primary (driving) adapters, such as:
  - Web controllers (REST endpoints)
  - CLI handlers
  - Schedulers

The “In” Ports (Driving)

- In Ports are usually interfaces declared in the application, implemented by a service that coordinates domain logic
- They enable the domain to be invoked without depending on the UI or framework
- They help express user-facing intentions, or **use-cases**, and not low-level operations

The “In” Ports (Driving)

- In this case the CLI is the adapter, and it driving the port, which in this case **is in the Application Layer**

The “Out” Ports

The “Out” Ports (Driven)

- An interface the core defines to express external dependencies
- Used for things like databases, APIs, messaging, file systems
- Declares what is needed, not how it’s done
- Keeps core logic decoupled from infrastructure
- Implemented by secondary (driven) adapters

The “Out” Ports (Driven)

- In this case the CLI is the adapter, and it driving the port, which in this case is in the **Infrastructure Layer.**

Adapters

What is an Adapter?

- Bridge between the core and the outside world
- Implement ports (input or output)
- Translate between external formats (e.g., HTTP, SQL, JSON) and core models
- Examples: Web controllers, database gateways, REST clients, CLI handlers
- Belongs to the **Infrastructure Layer**

The “In” Adapters

What is an Adapter?

- Bridge between the core and the outside world
- Implement ports (input or output)
- Translate between external formats (e.g., HTTP, SQL, JSON) and core models
- Examples: Web controllers, database gateways, REST clients, CLI handlers
- Belongs to the **Infrastructure Layer**

The “In” Adapter (Driving)

- In this case the CLI is the adapter, and it driving the port, which in this case **is in the Application Layer**

The “Out” Adapters

The Out Adapters (Driven)

- Implement output ports defined by the core
- Handle infrastructure concerns (DB, APIs, messaging, file systems) concretely
- Translate core requests into technical operations
- Examples: `JpaOrderRepository`, `KafkaEventPublisher`, `RestProductClient`
- Plug into the system without touching domain logic

The “Out” Adapter (Driven)

- In this case the CLI is the adapter, and it driving the port, which in this case is in the **Infrastructure Layer.**

Naming “Conventions” for Output Ports and Adapters

Responsibility	Port Interface (in domain)	Adapter Implementation (in infrastructure)
Persistence / DB	OrderRepository	JpaOrderRepository, MongoOrderRepository
Messaging / Events	OrderEventPublisher	KafkaOrderEventPublisher, RabbitOrderEventPublisher
External API (e.g., REST)	ProductCatalogClient or ProductCatalog	RestProductCatalogClient, FeignProductCatalogClient
Email / Notification	NotificationSender	SmtpNotificationSender, SesNotificationSender
File Storage	FileStorageService	S3FileStorageService, LocalFileStorageService
LLM / AI Service	AIClient or LLMClient	OpenAIClient, VertexLLMClient, HuggingFaceLLMClient
Payment Gateway	PaymentProcessor	StripePaymentProcessor, MockPaymentProcessor
Search	SearchService or SearchGateway	ElasticSearchGateway, MeilisearchSearchService

Demo: Folder Structure

- Let’s demonstrate the folder structure of a Ports and Adapter Architecture
- You will notice that everything will have its place

Demo: Let’s Create a Minimal Application

- Let’s create a minimal application with the components in place
- We will build atop the folder structure:
  - The domain
  - The ports
  - The adapters

Data Transfer Objects

Data Transfer Objects

- Simple objects used to transfer data across layers or systems
- Contain no business logic
- Used by adapters to communicate with external systems (e.g., HTTP, DB, JSON)
- Help decouple domain models from infrastructure concerns
- Often mapped to/from domain objects in application or adapter layers
- Meant for organizing the data for the benefit of the adapters

Integrating Testing

Test Driven Development

- Core business logic lives in the domain layer, free of infrastructure
- TDD is easy because the domain uses pure functions and value objects
- Input ports can be tested with mocks of output ports
- Enables fast, isolated, unit tests without databases or HTTP

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

End-to-End Testing

- With adapters plugged in, ports allow for full system flow testing
- Tests go through UI or API → input port → domain → output port → DB
- Architecture keeps dependencies clear — fewer surprises
- Great for testing real-world workflows with full stack
- Another great idea is you can also swap out the DB with an in-memory database

Infrastructure
Application
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

Behavior Driven Development

- Input ports represent use cases — perfect for scenario-driven specs
- Use cases can be exercised directly, without UI or DB
- Tests express expected business behavior, not internal steps
- Matches well with `Given-When-Then` frameworks like Cucumber or JUnit 5's `@DisplayName`

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

Data Driven Testing

- Output ports let you swap in real infrastructure for adapter testing
- Use **TestContainers** to run real DBs, APIs, brokers in test environments
- Contracts are enforced by testing the adapter separately
- Great for adapter-level integration tests with reproducible environments

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Adapter
Port
Adapter

Demo: Let’s Demonstrate a Testing Harness

- Let’s ensure that testing harnesses are in place
- This includes:
  - Test Driven Development
  - End to End Testing
  - Behavior Driven Development
  - Data Driven Testing

What about LLMs?

LLMs, Ready for the Modern Era

- The joy of Ports and Adapters, is you need a:
  - An `interface` with the methods that you require to connect to an LLM, MCP, or Agentic Server.
  - An Adapter of that `interface` that does the communication
  - Give `interfaces` a consistent naming like `LLMClient` (e.g. then create an adapter called `OpenAILLMClient`)

Domain Driven Design

How does DDD relate?

- “Domain-Driven Design (DDD) and Ports & Adapters are different concepts. They exist independently and evolved separately.”
- “While not related as architectural patterns and not explicitly tied to one another, the two are compatible.”

Bounded Contexts

Learning Domain Driven Design - Vlad Khononov

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

Domain Model Pattern

- The domain model pattern is intended to cope with cases of complex business logic.
- Instead of CRUD, we deal with complex state transitions, business rules, and invariants

Implementation of the Domain Model Pattern

- A domain model is an object model of the domain that incorporates both behavior and data
- DDD's tactical patterns — aggregates, value objects, domain events, and domain services— are the building blocks of such an object model.
- All of these patterns share a common theme: they put the business logic first
- They will all be implemented in plain objects
  - No Infrastructure and no technology concerns like frameworks
- Adhere to all the correct terms: **Ubiquitous Language**

What goes in the Domain Model Pattern?

- Developers will develop the following software artifacts:
  - Value Objects
  - Entities
  - Aggregates
  - Domain Events
  - Domain Services
  - Application Services

Value Objects

- Object that can be identified by the composition of its values
- No explicit identification is required
- Changing the attributes of any of the fields yields a different object
- Value Objects can be used to counter the “Primitive Obsession” code smell
- Typically immutable

```
class Color {
    int red;
    int green;
    int blue;
}
```

Entities

- Opposite of a value object and requires explicit identification
- Explicit identification is required
- For example, an `Employee`, just identified by an employee's first name and last name
- Entities are subject to change

```
class Employee {
   private Name name;
   //Constructors,
   // equals, hashCode, etc.
}
```

```
class Employee {
   private EmployeeId employeeId;
   private Name name;
   //Constructors,
   // equals, hashCode, etc.
}
```

Aggregates

- An entity that manages a cluster of domain objects as a single unit
- Consider an `Order` to `OrderLineItem` relationship
- The root entity ensures the integrity of its parts
- Transactions should not cross aggregate boundaries
- They are domain objects(order, playlist), they are not collections, like `List`, `Set`, `Map`

```
class Album {
   private Name name;
   private List<Track> tracks;

   public void addTrack(Track track) {
      this.tracks.add(track);
   }

   public List<Track> getTracks() {
      return Collections.copy(tracks)
   }
}
```

Domain Events

- The identified domain events directly become domain events in the implementation.
- These are immutable objects that capture significant occurrences in the business domain, reflecting changes in state or triggering further actions within the system.
- The events can then emitted from the aggregates and perform other functions

```
public class Ticket {
    private List<DomainEvent> domainEvents;
    public void execute(RequestEscalation cmd) {
        if (!this.isEscalated &&
            this.remainingTimePercentage <= 0) {
               this.isEscalated = true;
               var escalatedEvent =
                   new TicketEscalated(id, cmd.Reason);
            domainEvents.append(escalatedEvent);
        }
    }
}
```

Aggregate
Domain event
Domain event
Domain event
Subscriber
Subscriber
Subscriber

Aggregate as a Consistency Boundary

Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

Keep Aggregates Small

- Keep the aggregates as small as possible and include only objects that are required to be in a strongly consistent state by the aggregate's business logic

- In the following code example, notice that we are not interested in the entire `Product` nor the entire `User`, just their identifiers (entities)

```
public class Ticket {
    private UserId          customer;
    private List<ProductId> products;
    private UserId          assignedAgent;
    private List<Message>   messages;
}
```

- Reasoning behind referencing external aggregates by ID is to reify that these objects do not belong to the aggregate's boundary, and to ensure that each aggregate has its own transactional boundary.

Domain Services

- Stateless object that implements the business logic
- It naturally doesn't belong to any of the domain model's aggregates or value objects

```
class OrderTax {
   //no state
   private void calculateTaxWithRate(Order
         order, TaxRate taxRate) {
       //...
   }
}
```

Infrastructure
Application
Domain
Adapter
Port
Port
Adapter
Port
Port
Adapter
Adapter

Application Services

- Makes use of Repositories (Storage Abstractions)
- Aggregate instances and then sent for transforming to a transformed object
- The transformed object will typically be routed to the UI

```
class OrderService {
   private OrderRepository orderRepository;
   private void persistOrder(Order) {
       //...
   }
}
```

Transactions

- Transactions should be managed in the Application Layer
- **Application layer** coordinates use cases and knows what needs to be atomic
- **The Domain layer should be unaware of transactions**
- Infrastructure (e.g., JPA, JDBC) provides the mechanism, but not the boundary
- Helps maintain separation of concerns: domain models stay pure, infra stays swappable
- If using DDD, this Application Service Layer is where Transactions should be assigned

Transactions (Spring & Quarkus)

```
@Override
@Transactional
public void placeOrder(PlaceOrderCommand command) {
    Customer customer = customerRepository
                .findById(command.getCustomerId())
                .orElseGet(() -> new Customer(command.getCustomerId(),
                                              command.getCustomerName()));
    Order order = Order.create(customer, command.getLineItems());
    customerRepository.save(customer);
    orderRepository.save(order);
}
```

What if I don’t have `@Transactional`

- There are always strategies to run services transactionally:
  - Unit of Work Pattern
  - Scoped Values (New Alternative to `ThreadLocal`)
  - For Scala, Effect Systems

Unit of Work

```
public interface UnitOfWork {
    void begin();
    void commit();
    void rollback();
    Connection getConnection();
}
```

Transactions at Service (Unit of Work)

```
public void placeOrder(PlaceOrderCommand command) {
    unitOfWork.begin();
    try {
        Customer customer = customerRepository
            .findById(command.getCustomerId())
            .orElseGet(() -> new Customer(
                command.getCustomerId(), command.getCustomerName()
            ));
        Order order = Order.create(customer, command.getLineItems());
        customerRepository.save(customer);
        orderRepository.save(order);
        unitOfWork.commit();
    } catch (Exception e) {
        unitOfWork.rollback();
        throw new RuntimeException("Failed to place order", e);
    }}
```

ScopedValue

```
try (Connection conn = dataSource.getConnection()) {
    conn.setAutoCommit(false);

    ScopedValue.where(ConnectionScope.CURRENT, conn).run(() -> {
        Customer customer = customerRepository
            .findById(command.getCustomerId())
            .orElseGet(() -> new Customer(
                command.getCustomerId(), command.getCustomerName()
            ));
        Order order = Order.create(customer, command.getLineItems());
        customerRepository.save(customer);
        orderRepository.save(order);
        conn.commit();
    });
}
```

Effect Systems (Scala)

```
class PlaceOrderService[F[_]: Monad](
      customerRepo: CustomerRepository[F], orderRepo: OrderRepository[F]) {

  def placeOrder[G[_]: Monad](command: PlaceOrderCommand)
                              (using fk: FunctionK[F, G]): G[(Long, Long)] = {
    val program: F[(Long, Long)] = for {
      maybeCustomer <- customerRepo.findById(command.customerId)
      customer = maybeCustomer.getOrElse(Customer(command.customerId, command.customerName))
      order = Order(customer.id, command.lineItems)
      customerId <- customerRepo.save(customer)
      orderId    <- orderRepo.save(order)
    } yield (customerId, orderId)
    fk(program)
  }
}
```

What is the Win?

DDD compliments nicely
It helps identify terms for business, and gives a set of design patterns that can be pluggable into our domains.

Use Ports And Adapters Without DDD?
Use Ports And Adapters With DDD?
https://vinnylingham.com/a-fork-in-the-road-70288fd3c046

Demo: Final Review of a Full Application

- Let’s go ahead and fill in the rest of the application.
- Here we will see the components lined up and we will see each layer tested well
- We will also see how we integrated Domain Driven Design components like aggregates, entities, and value objects

Microservices with Ports and Adapters

Microservice with Ports and Adapters

- Each microservice can be designed as its own hexagonal architecture
- Domain logic is at the core, with adapters for APIs, messaging, databases
- Ports define clear boundaries for communication and infrastructure
- Encourages autonomous, testable services that evolve independently

Onion & Clean Architecture

Onion Architecture

- Introduced by Jeffrey Palermo
- Concentric rings around the domain, each layer only depends inward
- No explicit notion of “ports,” but same effect via interfaces
- Emphasizes dependency rule: outer layers can depend on inner, never the reverse

https://medium.com/expedia-group-tech/onion-architecture-deed8a554423

Clean Architecture

- Popularized by Robert C. Martin
- Synthesizes:
  - Ports & Adapters
  - Onion
  - SOLID (SRP, OCP, LSP, Interface Segregation, Dependency Inversion)
- Adds a few layers of clarity, especially Use Cases and Entities
- Very testable and strict about boundaries

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Layered Architecture

Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

Why Layered Architecture Falls Short

- Tight coupling between layers — domain often depends on frameworks (e.g., JPA, Spring)
- Hard to test core logic in isolation — requires UI or database setup
- Layers often become leaky: business logic drifts into controllers or persistence
- Changes in infrastructure ripple inward through the stack
- Encourages “big ball of mud” if boundaries aren’t strictly enforced

““A layered architecture has you separate code by concerns and arrange them from “higher” and “lower,” such that higher-level items call or have a dependency upon lower-level ones. More abstract concerns are placed higher in the architecture, while hardware and drivers sit on the bottom while policy-oriented items rise to the top. The policy items have dependency on the drivers and hardware.”

Hexagonal Architecture Explained: How the Ports & Adapters Architecture Simplifies Your Life, and How to Implement It (Series on Object-Oriented Design)

Refactoring to Ports and Adapters

Refactoring to Ports and Adapters

- Know your IDE: Use your refactoring tools to move, extract, and rename safely.
- Watch for code smells:
  - **Feature Envy:** Logic living in the wrong class.
  - **Primitive Obsession:** Overusing raw types instead of value objects.
- Apply refactoring patterns:
  - Move Method, Extract Method, Introduce Delegate
- Rely on Tests: Good tests give you confidence to refactor safely.
- Have courage: Refactoring changes structure, not behavior—trust your tests and design instincts.

Demo: Refactoring to Port and Adapters

- Let’s use Ted Young’s Blackjack and do some Refactoring.
- Find more about Ted Young’s project at https://moretestable.com

Hexagonal Architecture Explained

How the Ports & Adapters architecture simplifies your life, and how to implement it

Alistair Cockburn
Juan Manuel Garrido de Paz

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext