Comparison between Ports in Hexagonal Architecture and Interfaces in Clean Architecture: A Conceptual and Practical Analysis

Nagib Sabbag Filho
FIAP (Faculty of Informatics and Administration Paulista) Avenida Paulista, 1106 - 7º andar - Bela Vista, São Paulo, Brazil.
e-mail: profnagib.filho@fiap.com.br
PermaLink: https://leaders.tec.br/article/8793e4
ago 26 2024

Abstract:
The article explores the similarities and differences between two central concepts in popular software architectures: Ports in Hexagonal Architecture and Interfaces in Clean Architecture. The analysis addresses the theoretical foundations of each approach, highlighting how these concepts shape the structure and interaction of software components.

Key words:
hexagonal architecture, ports, interfaces, clean architecture, differentiation, software design, dependencies, communication, adaptation, testing, abstraction, components, systems, flexibility, decoupling, layers, interactions, implementations, design patterns.

Fundamental Concepts of Software Architecture

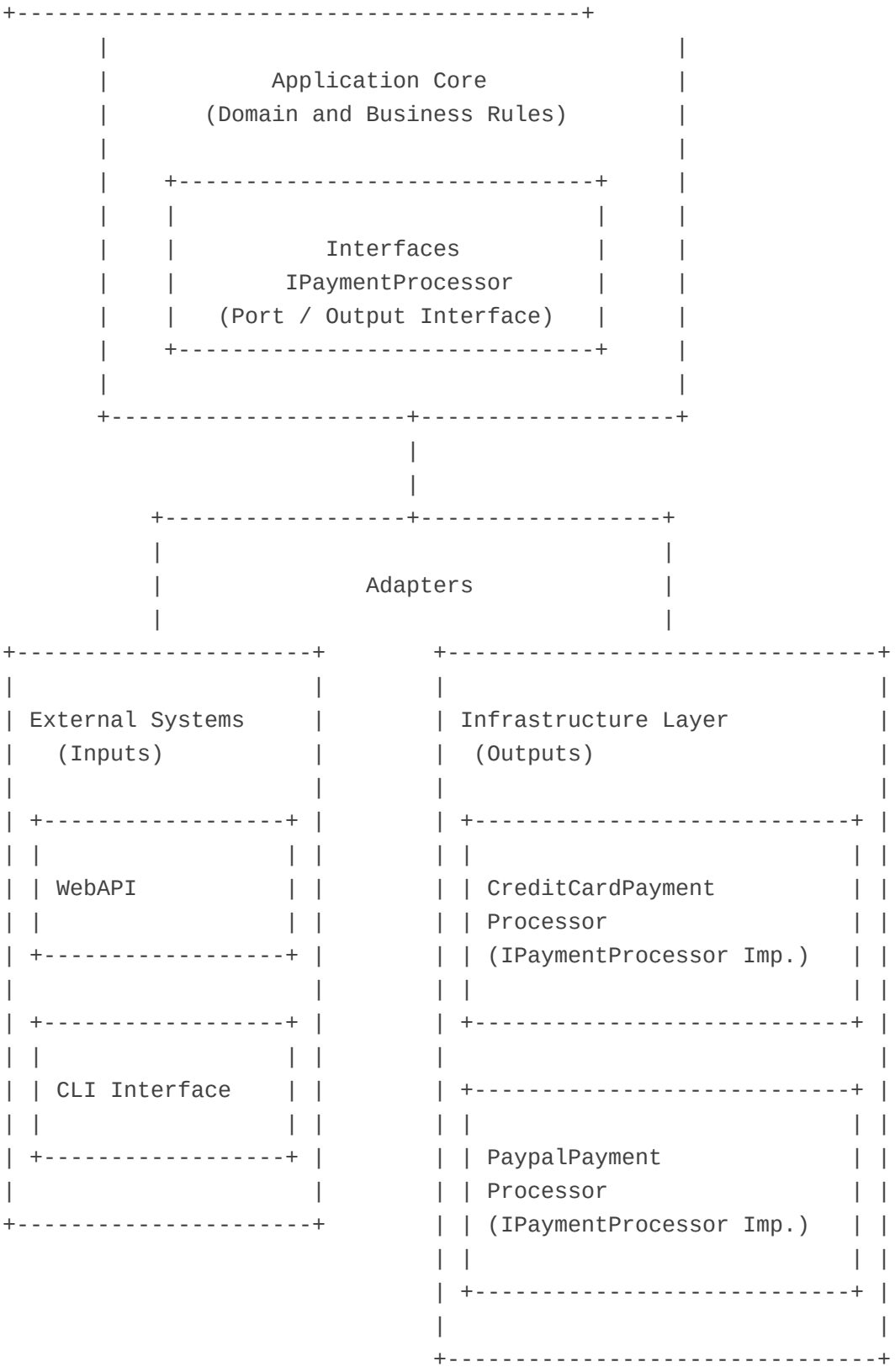
Software architecture plays an essential role in building complex and durable systems. It defines guidelines for the organization of code and the interaction between system components, directly influencing its scalability, flexibility, and maintenance. With the advancement of technologies and the growth of demands for more robust systems, various architectural approaches have emerged to address common problems in software engineering. Among these approaches, Hexagonal Architecture and Clean Architecture stand out, both aiming to create systems that are independent of technical details, allowing for easy adaptation to technological changes. However, each of these architectures adopts specific principles and patterns that differentiate them in terms of organization and handling of external dependencies (MARTIN, 2017).

Hexagonal Architecture: Structure and Function

Hexagonal Architecture, also known as Ports and Adapters Architecture, was introduced by Alistair Cockburn as a solution to isolate the core of the application from external interactions. Its main objective is to protect the business logic from changes in external technologies, thereby promoting an architecture that is both flexible and robust. In Hexagonal Architecture, the core of the application interacts with the outside world through "ports," which are well-defined interfaces. These ports allow different adapters to be connected to the core so that the business logic can be executed independently of external technologies, such as databases, user interfaces, or third-party services (RICHARDS, 2015).

An illustrative example of this architecture can be found in e-commerce systems. In this context, an application may define a port that represents the interface for processing payments. This port can be implemented by various adapters, such as an adapter for credit card payments and another for PayPal payments. This modular structure allows the central logic of the system to remain unchanged, even as new payment methods are added. Thus, Hexagonal Architecture promotes the continuous evolution of the system without compromising the stability and integrity of the application.

Design Example - Hexagonal Architecture



Application Core - Interfaces

The interface IPaymentProcessor defines the contract that both payment processors must follow:

```
namespace ApplicationCore.Interfaces
{
    public interface IPaymentProcessor
    {
```

```
        bool ProcessPayment(decimal amount, string paymentDetails);
    }
}
```

Adapters - External Systems - WebAPI - Controllers

The Web API PaymentController uses the implementations of IPaymentProcessor to process payments both with credit card and PayPal:

```
using ApplicationCore.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace Adapters.ExternalSystems.WebAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class PaymentController : ControllerBase
    {
        private readonly IPaymentProcessor _creditCardPaymentProcessor;
        private readonly IPaymentProcessor _paypalPaymentProcessor;

        public PaymentController(IPaymentProcessor creditCardPaymentProcessor,
IPaymentProcessor paypalPaymentProcessor)
        {
            _creditCardPaymentProcessor = creditCardPaymentProcessor;
            _paypalPaymentProcessor = paypalPaymentProcessor;
        }

        [HttpPost("credit-card")]
        public IActionResult ProcessCreditCardPayment([FromBody] PaymentRequestModel
request)
        {
            var result = _creditCardPaymentProcessor.ProcessPayment(request.Amount,
request.PaymentDetails);

            if (result)
                return Ok("Credit card payment processed successfully.");

            return BadRequest("Credit card payment processing failed.");
        }

        [HttpPost("paypal")]
        public IActionResult ProcessPayPalPayment([FromBody] PaymentRequestModel
request)
        {
            var result = _paypalPaymentProcessor.ProcessPayment(request.Amount,
request.PaymentDetails);

            if (result)
                return Ok("PayPal payment processed successfully.");

            return BadRequest("PayPal payment processing failed.");
```

```
        }
    }
}
```

Adapters - External Systems - WebAPI - Models

```
namespace Adapters.ExternalSystems.WebAPI.Models
{
    public class PaymentRequestModel
    {
        public decimal Amount { get; set; }
        public string PaymentDetails { get; set; }
    }
}
```

Adapters - Infrastructure Layer - CreditCardPaymentProcessor

The implementation CreditCardPaymentProcessor is responsible for performing the actual payment processing, simulating integrations with external payment systems:

```
using ApplicationCore.Interfaces;
using System;

namespace Adapters.InfrastructureLayer.PaymentProcessors
{
    public class CreditCardPaymentProcessor : IPaymentProcessor
    {
        public bool ProcessPayment(decimal amount, string paymentDetails)
        {
            // Logic to process the credit card payment
            // Example: Integration with an external payment gateway

            Console.WriteLine($"Processing credit card payment of {amount} with details
{paymentDetails}.");
            return true; // Simulation of success
        }
    }
}
```

Adapters - Infrastructure Layer - PayPalPaymentProcessor

The implementation PayPalPaymentProcessor is responsible for processing payments via PayPal:

```
using ApplicationCore.Interfaces;
using System;

namespace Adapters.InfrastructureLayer.PaymentProcessors
{
    public class PayPalPaymentProcessor : IPaymentProcessor
    {
        public bool ProcessPayment(decimal amount, string paymentDetails)
        {
            // Logic to process the payment via PayPal
            // Example: Integration with the PayPal API
```

```
            Console.WriteLine($"Processing PayPal payment of {amount} with details
{paymentDetails}.");
            return true; // Simulation of success
        }
    }
}
```

Adapters - External Systems - WebAPI - Services.AddScoped

Dependency Injection configuration:

```
services.AddControllers();

// Dependency injection for payment processors
services.AddScoped < IPaymentProcessor, CreditCardPaymentProcessor > (provider =>
            new CreditCardPaymentProcessor());

services.AddScoped < IPaymentProcessor, PayPalPaymentProcessor > (provider =>
            new PayPalPaymentProcessor());
```

This ports and adapters pattern also promotes the testability of the application, as the business logic can be tested in isolation using mock implementations of the ports. Additionally, the use of ports as entry and exit points allows for a clear definition of responsibilities within the system, resulting in a more cohesive and modular architecture. Thus, Hexagonal Architecture not only facilitates the maintenance and evolution of the system but also promotes a cleaner and more organized design, where each component has a well-defined responsibility and is easily replaceable.

Clean Architecture: Interfaces and Organizations

Clean Architecture, proposed by Robert C. Martin (also known as Uncle Bob), follows similar principles but with a different approach in terms of organization. Clean Architecture emphasizes the separation of concerns and independence from frameworks and libraries, so that the core of the application can evolve independently of external changes. This approach organizes the system into concentric layers, where the business core (domain) is at the center, surrounded by layers of use cases, and finally, by the interface and infrastructure layer. This way, the business logic is completely isolated from external concerns, ensuring that changes in technologies or frameworks do not affect the core of the application (MARTIN, 2017).

A typical example of applying Clean Architecture can be found in task management systems. In this scenario, the core of the system consists of entities and use cases that define the rules and business flows. For example, a Task entity may contain properties such as Id, Name, and Completion Status. A use case, such as Task Management, may be responsible for marking a task as completed. The interfaces for user interaction and data persistence are placed in the outer layers, so that the business logic can be tested and developed independently of the user interface or database technology used.

Design Example - Clean Architecture

```
+-------------------------------------------------------------+
|                       Presentation Layer                    |
|-------------------------------------------------------------|
|                                                             |
| User Interfaces (Views, Controllers, APIs)                  |
| [TasksController] : [ControllerBase]                        |
|                                                             |
+-------------------------------------------------------------+
```

```
|                       Application Layer                     |
|-------------------------------------------------------------|
|                                                             |
| Use Cases                                                   |
| Specific application rules                                  |
| [TaskService] : [ITaskService]                              |
|                                                             |
+-------------------------------------------------------------+
|                       Infrastructure Layer                  |
|-------------------------------------------------------------|
|                                                             |
| Infra Implementations                                       |
| Access to databases, external systems                       |
| [TaskRepository] : [ITaskRepository]                        |
|                                                             |
+-------------------------------------------------------------+
|                       Domain Layer                          |
|-------------------------------------------------------------|
|                                                             |
| Entities                                                    |
| Core business rules                                         |
| [TaskEntity]                                                |
| Interfaces - [ITaskService], [ITaskRepository]              |
|                                                             |
+-------------------------------------------------------------+
```

Presentation Layer

The presentation layer deals with user interaction. It can be an MVC controller or API in ASP.NET Core:

```
// Controller in ASP.NET Core
[ApiController]
[Route("api/[controller]")]
public class TasksController : ControllerBase
{
    private readonly ITaskService _taskService;

    public TasksController(ITaskService taskService)
    {
        _taskService = taskService;
    }

    [HttpPost("create")]
    public IActionResult CreateTask([FromBody] CreateTaskRequest request)
    {
        var result = _taskService.CreateTask(request);
        return Ok(result);
    }

    [HttpGet("list")]
    public IActionResult ListTasks()
    {
```

```
        var tasks = _taskService.GetTasks();
        return Ok(tasks);
    }
}
```

Application Layer

The application layer deals with application-specific logic and use rules:

```
// Application Service
public class TaskService : ITaskService
{
    private readonly ITaskRepository _taskRepository;

    public TaskService(ITaskRepository taskRepository)
    {
        _taskRepository = taskRepository;
    }

    public TaskResult CreateTask(CreateTaskRequest request)
    {
        // Apply specific application rules
        if (string.IsNullOrEmpty(request.Title))
        {
            throw new ArgumentException("Title cannot be empty.");
        }

        // Create a new task
        var task = new TaskEntity
        {
            Title = request.Title,
            Description = request.Description,
            CreatedAt = DateTime.UtcNow
        };

        _taskRepository.AddTask(task);

        return new TaskResult
        {
            Success = true,
            TaskId = task.Id
        };
    }

    public IEnumerable < TaskDto > GetTasks()
    {
        var tasks = _taskRepository.GetAllTasks();
        return tasks.Select(task => new TaskDto
        {
            Id = task.Id,
            Title = task.Title,
            Description = task.Description
```

```
        });
    }
}
```

Domain Layer

The domain layer contains the entities and core business rules:

```
// Domain Entity
public class TaskEntity
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime CreatedAt { get; set; }
}

// Domain Repository Interface
public interface ITaskRepository
{
    void AddTask(TaskEntity task);
    IEnumerable < TaskEntity > GetAllTasks();
}

// Domain Service Interface
public interface ITaskService
{
    TaskResult CreateTask(CreateTaskRequest request);
    IEnumerable < TaskDto > GetTasks();
}
```

Infrastructure Layer

The infrastructure layer provides concrete implementations and access to external systems, such as databases:

```
// Infrastructure Implementation
public class TaskRepository : ITaskRepository
{
    private readonly ApplicationDbContext _context;

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

    public void AddTask(TaskEntity task)
    {
        _context.Tasks.Add(task);
        _context.SaveChanges();
    }

    public IEnumerable < TaskEntity > GetAllTasks()
    {
```

```
        return _context.Tasks.ToList();
    }
}

// ApplicationDbContext
public class ApplicationDbContext : DbContext
{
    public DbSet < TaskEntity > Tasks { get; set; }

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

Connecting the layers

Dependency injection configuration:

```
// Startup.cs or Program.cs
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext < ApplicationDbContext > (options =>
            options.UseSqlServer("YourConnectionString"));

        services.AddScoped < ITaskRepository, TaskRepository > ();
        services.AddScoped < ITaskService, TaskService > ();
        services.AddControllers();
    }
}
```

Clean Architecture, by rigorously separating concerns into layers, facilitates the maintenance and evolution of the system, ensuring that changes in one layer do not affect the others. This also promotes code reuse, as the business logic can be utilized in different contexts, such as a web application, a REST API, or a console application, without the need for changes. Furthermore, this architecture allows the business logic to be tested in isolation, without the need for external dependencies, resulting in faster and more reliable tests.

Another important aspect of Clean Architecture is its ability to handle changes. In an agile development environment, where changes are constant, this architecture allows the system to evolve incrementally, without the need to rewrite large parts of the code. By keeping the business logic isolated from other layers, Clean Architecture promotes greater flexibility and adaptability of the system, allowing it to keep pace with changes in business needs and in the technologies used.

Differences between Ports and Interfaces

The main difference between the ports of Hexagonal Architecture and the interfaces of Clean Architecture lies in the context and the purpose of each. In Hexagonal Architecture, the ports are the entry and exit points of the application core, allowing different adapters to be connected as needed. On the other hand, in Clean Architecture, the interfaces serve as contracts that define the interactions between the layers of the application. Although both approaches utilize interfaces to promote the flexibility and modularity of the code, the way these interfaces are employed and the structure of the application as a whole differ between the two architectures.

While Hexagonal Architecture focuses on bidirectional communication between the core of the application and the

outside world, Clean Architecture emphasizes a layered structure that completely isolates the business logic from any external dependency. This fundamental difference influences how systems are designed and maintained over time. In Hexagonal Architecture, the emphasis is on allowing the core of the application to communicate flexibly with the outside world, while in Clean Architecture, the focus is on ensuring that the core of the application is completely independent of external details.

Final Considerations

Both Hexagonal Architecture and Clean Architecture offer valuable approaches for building flexible, modular, and easily maintainable systems. The choice between one or the other depends on the specific needs of the project and the preferences of the development team. In some cases, it may be interesting to combine aspects of both approaches to leverage their respective advantages. Regardless of the choice, it is essential that the software architecture is designed to support the evolution of the system, allowing it to adapt to technological changes and new business demands continuously and efficiently.

References

For more information on architectures, consult the official documentation and other available resources:

Richard, M. Hexagonal Architecture. In: Richards, Mark. Software Architecture Patterns. Sebastopol: O'Reilly Media, 2015. Available at: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437. Accessed on: Aug 19, 2024.
Martin, R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall. Accessed on: Aug 21, 2024.

Nagib is a University Professor and Tech Manager. He has a history of achievements in technical and agile certifications, including MCSD, MCSA, and PSM1. He holds a postgraduate degree in IT Management from SENAC and an MBA in Software Technology from USP. Nagib has completed extension programs at MIT and the University of Chicago. Other achievements include the authorship of a peer-reviewed article on chatbots, presented at the University of Barcelona.