Dependency Injection as a Programming Paradigm

CAB402 Assignment 2

Eric Joseph Kizhakkebhagathu
N11190728@qut.edu.au

Table of Contents

Dependency Injection as a Programming Paradigm

Introduction

Dependency Injection (DI) is an architectural design pattern that changes how programmers think about how to organise programs and how to assign tasks. Although most statically-typed languages offer DI through frameworks or libraries, it is not a built-in language feature or simple syntax shortcut. Rather, DI is a high-level design approach or a “programming paradigm” that teams can adopt to build more modular, loosely coupled systems.

DI is a technique in which objects get the dependencies they need from external sources (“an injector”) instead of creating those dependencies internally. (Spring, 2025). This means that a class no longer manages the construction of its collaborators. Instead, another component, usually an IoC (Inversion of Control) container or factory assembles the required objects and supplies them to the class. By removing such hard-coded dependencies, DI makes an application’s code easier to maintain and test (Microsoft, 2025).

See Appendix 1, very basic example of constructor DI, we observe how DI cleanly separates dependency assembly from class logic. A traditional approach, on the other hand, embeds the dependency directly into the Notification Manager like this:

```
public class NotificationManager
{
    private readonly IMessageService _messageService;


    public NotificationManager()
    {
        _messageService = new EmailService();
    }

    public void Notify(string message)
    {
        _messageService.Send(message);
    }
}
```

At first look, this hard-coded instantiation may not seem like a big deal, but it breaks the Dependency Inversion Principle by tightly linking Notification Manager class to the concrete Email Service class.

Such coupling makes it difficult to substitute alternative implementations whether for testing (e.g. mocks) or to extend functionality and scatters configuration logic throughout the codebase. In the following sections, we will examine these pain points rigid coupling, compromised testability, and limited extensibility and demonstrate how the DI paradigm remedies them through loose coupling, modular composition, and

inversion of control. All code that is used in this report can be found under the following GitHub repository *https://github.com/Theericjoseph/DependencyInjection*.

History and Adoption of DI

Dependency Injection really grew out of the broader idea of Inversion of Control (IoC), in object-oriented design instead of a class looking up or instantiating its dependencies, something else wires them up for you. You have probably seen IoC in GUI toolkits, where the framework calls your event handlers rather than the other way around (inverting the usual flow of control) (Fowler, 2004). In 2004, Martin Fowler gave this wiring-up pattern its name "**Dependency Injection**" and laid out the main flavours (constructor, setter, interface) while contrasting it with the Service Locator approach (Fowler, 2004). Almost at the same time, Rod Johnson bundled those ideas into the first Spring Framework for Java (described in his 2002 book and released as Spring 1.0 in March 2004), showing how a lightweight container could bring IoC and DI to real projects (Victor, 2023) (WIKIPEDIA, 2025).

These achievements established Dependency Injection (DI) as a distinct programming paradigm, and now, .NET has robust support for DI, with a built-in service container for dependency provision (Microsoft, 2024). The same ideas apply to dynamic languages, including Python, where modules like Dependency Injector are available to enable dependency injection using the same design pattern (Mogylatov, 2025). This universality underlines Dependency Injection's importance as a programming paradigm it is a cross-cutting architectural strategy aimed at building more loosely coupled, modular, and maintainable software systems.

Dependency Inversion Principle

We introduced the Dependency Inversion Principle (DIP) earlier not be confused with Dependency Injection: it states that high-level modules should not depend on low-level modules, but both should depend on abstractions. Abstractions should not depend on details. (DevIQ, 2025). In our **DIIntro** console app (see Appendix 1), the `NotificationManager` constructor asks for an `IMessageService` rather than instantiating `EmailService` directly. Because `NotificationManager` only knows about the `IMessageService` interface, you can inject any implementation `EmailService`, an `SmsService`, or even a mocked test service without changing its code, which exemplifies true inversion of dependencies. In the upcoming sections, we will explore various DI patterns and anti-patterns and evaluate how well they honour this principle.

Types of DI

There are three different ways in which a programmer can exercise dependency injection. These include constructor injection, property injection and interface injection (Fowler, 2004). In Appendix 2 you can see them all wired up in our payment-processing example.

1. Constructor Injection

Constructor injection is where you supply the object with all the dependencies it requires on creation, guaranteeing they are always available.

```
public class PaymentService
{
        private readonly IPaymentProcessor _processor;
        private readonly ILogger<PaymentService> _logger;

        public PaymentService(IPaymentProcessor processor,
        ILoggerFactory  loggerFactory)
        {
            _processor = processor;   // injected here
            _logger = loggerFactory.CreateLogger<PaymentService>();
        }

}
```

In this snippet, `PaymentService` declares its dependency on `IPaymentProcessor` via its constructor. At runtime, you simply pass in the concrete processor without ever changing the service itself. This makes the dependency explicit, keeps the class loosely coupled, and simplifies unit testing, achieving Dependency Inversion.

2. Property (Setter) Injection

Setter injection gives dependencies to an object after it has been built by giving them to public properties. Setter injection is different from constructor injection in that you do not have to provide all the necessary collaborators at creation time. This flexibility can be handy for optional features, but it also means you need extra null checks to guard against uninitialized dependencies.

```
public class PaymentService
{
        public IPaymentProcessor Processor { get; set; }

        private readonly ILogger<PaymentService> _logger;

        public PaymentService(ILoggerFactory  loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<PaymentService>();
        }

}
```

Here `PaymentService` can be instantiated without providing a `IPaymentProcessor` element on creation. This essentially defers the assignment of a payment processor till a later stage of execution till required or allows for swapping out different processors when required.

3. Method (Interface) Injection

Interface injection, which is also called method injection, gives the dependence straight to the method that needs it instead of storing it in the object. In other words, you send the collaborator in as a parameter every time you use the method. Your service never keeps conditional dependencies after the call ends. For our full implementation, see Appendix 2—PaymentService.cs.

```
public bool Pay(decimal amount, IPaymentDetails details)
{
    var result = false;
    if (amount <= 0)
    {
        _logger.LogError("Payment amount must be greater than zero.");
        return false;
    }
    if (IsPaymentDetailsValid(details) == false)
    {
       _logger.LogError("Payment details are invalid.");
       return false;
    }
    try
    {
      _logger.LogInformation("Processing payment of ${Amount}",amount);
      result = _processor.ProcessPayment(amount, details);
      // The processor tries to process the payment
    }
    catch (Exception ex)
    {
       _logger.LogError(ex, "Error logging payment details");
    }
    return result;
}
```

In our payment case, this method provides two important architectural benefits:

- After we process a payment, we do not **store** the customer's payment information in memory. `PaymentService` never keeps the objects because it simply passes `IPaymentDetails` to Pay(...).
- Object B (our service) does not have to keep or manage a dependence; rather it **validates** the details with an API and passes it onto the `IPaymentProcessor` to process it.

Dependency Injection Containers

Most major platforms these days come with built-in support for DI through different frameworks that allows programmers to setup Dependency Injection Containers. DI containers usually lie in your “composition root” usually called Main. DI containers do the work of wiring up all the services (Dependencies) in one spot. DI container then automatically constructs and injects the dependencies wherever they are needed. This helps centralising all the dependencies in the code base in one a single maintainable location.

In .NET, this functionality comes from the **Microsoft Extensions Dependency Injection** NuGet package. We used it in our payment-processor demo (see Appendix 2-Main.cs) to wire up configuration, logging, our factory, and the top-level App class. Dependency injection containers perform two core tasks:

- Registration: Map abstractions to concrete types (or factory functions) by calling methods like

```
services.AddTransient<IInterface, Implementation>();
services.AddSingleton(sp => new SomeType(...));
```

- Resolution: When you ask for a service at runtime the container builds the object graph by recursively resolving and injecting constructor parameters. For e.g.

```
var logger = provider.GetRequiredService<ILogger>();
```

In a standard console application like ours, Program.Main configures `IServiceCollection`, builds the `ServiceProvider` and then hands control to your top-level service (for us App.cs). From there on, every class simply declares its constructor dependencies and trusts the container to supply them. Bellow is a snippet of how we do it.

```
static void Main(string[] args)
{
    // 1) Build IConfiguration
    var config = new ConfigurationBuilder()
        .SetBasePath(AppContext.BaseDirectory)
        .AddJsonFile("appsettings.json", optional: false, reloadOnChange:
                     true)
        .Build();

    // 2) Setup DI container
    var services = new ServiceCollection()
        .AddSingleton<IConfiguration>(config)
        .AddLogging(builder => {
            builder.ClearProviders();
            builder.AddConsole();
            builder.AddDebug();
            builder.AddFilter<ConsoleLoggerProvider>(level => level >=
                                                              LogLevel.Error);
```

```
            builder.AddFilter<DebugLoggerProvider>(level => level >=
                                                   LogLevel.Information);
        })
        // bind PaymentConfig from JSON
        .Configure<Configuration.PaymentConfig>(
            config.GetSection("PaymentSettings"))
        .AddSingleton(sp =>
            sp.GetRequiredService<
                IOptions<Configuration.PaymentConfig>>()
                .Value)
        .AddSingleton(typeof(EnvironmentType), sp =>
            Enum.Parse<EnvironmentType>(
                sp.GetRequiredService<IConfiguration>()["Environment"]))
        .AddSingleton<Factories.PaymentProcessorFactory>()
        .AddTransient<App>()   // our entry-point wrapper
        // done
        .BuildServiceProvider();

    // 3) Resolve and run
    var app = services.GetRequiredService<App>();
    app.Run();
}
```

In the above snippet we register the following services:

- Configuration from appsettings.json
- Logging with console and debug providers
- EnvironmentType, parsed from configuration
- PaymentProcessorFactory, which itself depends on the config and environment
- App, our top-level service

Calling `.BuildServiceProvider()` then spins up the DI container with all these registrations.

When we do:

```
    var app = services.GetRequiredService<App>();
    app.Run();
```

the container automatically:

- **Constructs** an App instance
- **Resolves** its constructor parameters (`PaymentProcessorFactory` and `ILoggerFactory`)
- **Injects** them into the App constructor behind the scenes (see Appendix 2 - App.cs).

Because we registered App with the container, we never have to new it up manually or pass its dependencies by hand the DI framework handles it all for us.

Why should I DI ?

So far, we have covered the theory of DI, how it promotes loose coupling, modular composition, and dependency inversion, but what is in it for you as a developer? In this section, we will look at the real-world benefits of DI, such as how it makes unit testing easier, how it makes code safer and easier to maintain, and how it makes a system ready for change.

1. Unit Testing with Mocks and Fakes

Our PaymentService.Pay (see Appendix 2-PaymentService.cs) is a fairly complicated method and as any good developer you would want unit tests around this method. The method verifies the payment details from the user, validates the amount entered as well as call `ProcessPayment` API in the `IPaymentProcessor` class, but because each of the collaborators is injected as an interface, we can write clean, focused unit tests by Mocking the dependencies not being tested. For e.g.

- Mocking `IPaymentDetails` so that we can test is we are actually checking if the entered payment details or amount are valid before calling the API.
- Mocking `IPaymentProcessor` to return true or false, letting us verify that Pay relays the processor's result when all preconditions pass. (see Appendix 3)

```
[DataTestMethod]
[DataRow(0)]
[DataRow(-10)]
public void Pay_InvalidAmount_DoesNotCallProcessor(Int32 badAmount)
{
    // Arrange
    var processorMock = new Mock<IPaymentProcessor>();
    var detailsMock = new Mock<IPaymentDetails>();
    detailsMock.Setup(d => d.Verify());  // no exception

    var service = new PaymentService(
        processorMock.Object,
        _loggerFactoryMock.Object);

    // Act
    var result = service.Pay(badAmount, detailsMock.Object);

    // Assert
    Assert.IsFalse(result, "Pay should return false for non-positive
                    amounts.");
    processorMock.Verify(
        p => p.ProcessPayment(
            It.IsAny<decimal>(),
            It.IsAny<IPaymentDetails>()),
            Times.Never, "Processor must not be called on invalid amount.");
}
```

Appendix 3, shows the test suite for our Payment Service class, the setup for each test is minimal and we are able unit test small blocks of the code. Contrast that with testing a concrete processor we would have to load real configuration from JSON (or spin up a fake config file and bind it), manage environment settings, and potentially invoke external SDK code turning your “unit” test into a massive integration test. By programming against interfaces and letting DI supply mocks, our tests remain fast, isolated, and expressive. We can cover every branch of the Pay method (invalid details, bad amounts, successful and failed processing) with a handful of lines of setup, without ever touching the file system or network.

2. Scalable Strategy

DI enables programmers to design a highly scalable software architecture. When services are designed against abstraction, it becomes fairly easy to swap out existing services or scale up and implement newer services. We separate `PaymentService` from any specific payment method by programming to the `IPaymentProcessor` interface and putting all the instantiation in our `PaymentProcessorFactory` (Figure 1).

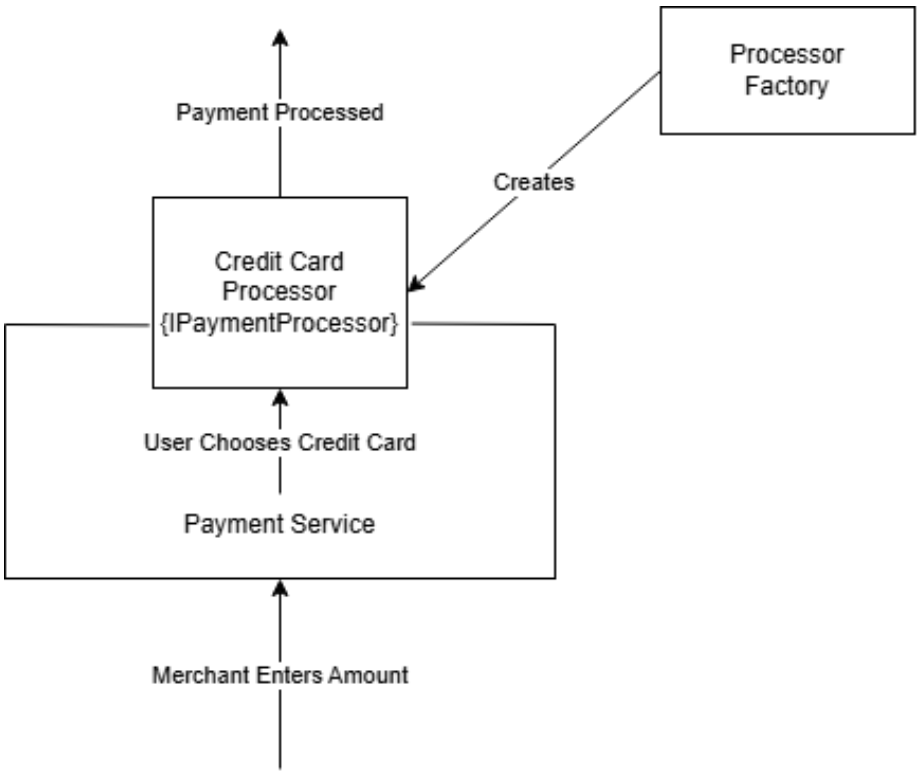

Figure 1 - Business logic of payment processor app

When we need to add PayPal support, we simply implement a new `PayPalProcessor` that conforms to `IPaymentProcessor` and register it in the `PaymentProcessorFactory` (Figure 2). This “plug-and-play” approach keeps business logic untouched, centralizes all wiring in one place, and makes adding or swapping services as easy as adding a class and updating a single mapping ensuring your system can grow and adapt without major refactoring or development time.

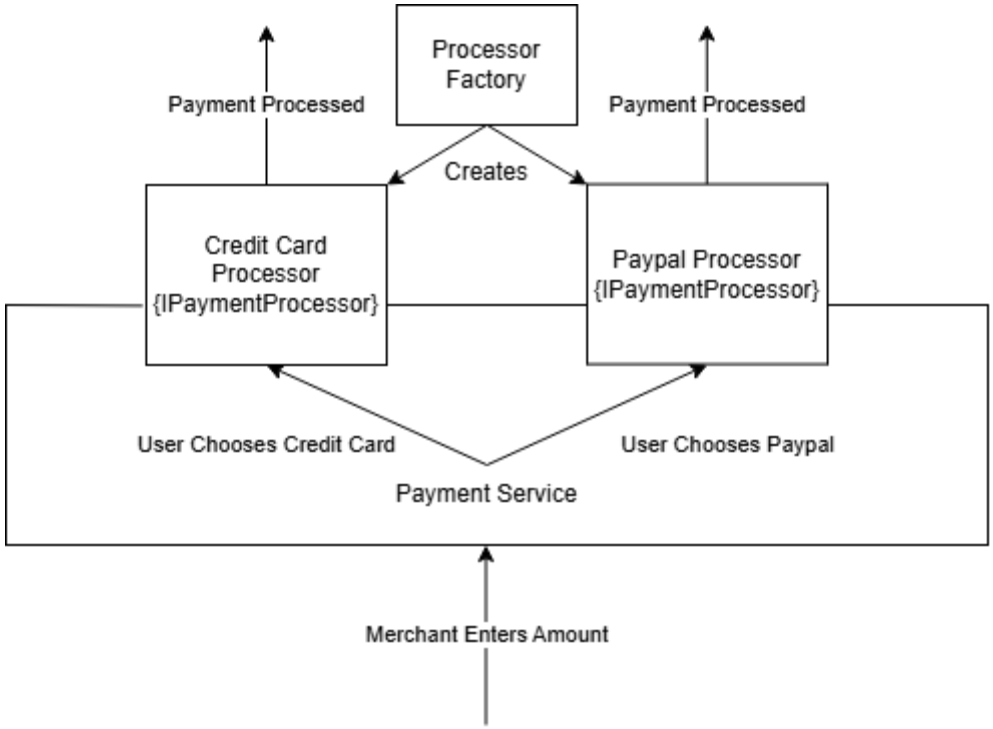

Figure 2 - Shows our business logic remains same when adding a new payment method

Anti-Patterns

1. Overusing Service Locator

A common anti-pattern, is using GetRequiredService<T>() (or injecting IServiceProvider) from inside your business classes. For e.g.

```
public class NotificationManager
{
    private readonly IMessageService _svc
        = ServiceLocator.Get<IMessageService>();  // hidden lookup

    public void Notify(string msg) => _svc.Send(msg);
}
```

This might look like "just DI" at first glance, but it actually goes against the Dependency Inversion Principle because `NotificationManager` now has a hidden dependency on `ServiceLocator` (Fowler, 2004). It is harder to do unit tests when dependencies are hidden behind a global lookup because each test must set up that location. The main thing to remember is that it's okay to use the container to start up your app in Program.Main (for example, a single `GetRequiredService()`), but you should never put service-locating calls all over your code. Instead, use constructor (or setter/method) injection to make every dependency clear. This way, the needs for each class will stay clear and can be tested.That way you enjoy the benefits of DI without slipping into the Service Locator trap.

2. Over-Injection / Constructor Bloat

This is a common scenario that can be seen in big projects where there ends up being one super class that does more that it should. Usually such classes require five, six, or more dependencies. This usually means that the class is doing too much or that the behaviours that go together have not been grouped correctly. When a constructor requires a long list of services, those dependencies get hidden behind abstractions, making debugging and maintenance much harder. Instead of injecting a dozen services into one class, refactor the class into smaller, single-responsibility components. This makes the code easier to read and much easier to maintain and test.

Conclusion

In this report, we introduce Dependency Injection as more than a technique, but as a programming paradigm that helps programmers design highly modular, loosely coupled software architectures which can be scale with increasing demand. DI favours abstraction instead of concreate implementation by shifting the responsibility for object creation and wiring out of business classes into a centralized composition root and injecting them as dependencies wherever required. This method makes it easy to change implementations, mock dependencies for unit tests, and build object graphs that are easy to maintain and understand.

DI has become a common term in many programming languages, from Java's Spring IoC container to Python's dependency-injector and TypeScript's InversifyJS. This shows that developers are changing the way they think about how to structure systems. Adopting DI as a paradigm has immediate benefits, such as faster testing, the ability to scale up, and a centralised configuration approach. However, it takes discipline to prevent problems like over-injecting or exploiting the container as a Service Locator. When done right, DI turns codebases into flexible, testable systems that can adapt to new requirements. This shows a completely different way of thinking about software architecture.

References

DevIQ. (2025, 05 26). *Dependency Inversion Principle*. Retrieved from deviq.com: https://deviq.com/principles/dependency-inversion-principle

Fowler, M. (2004, January 23). *Inversion of Control Containers and the Dependency Injection pattern*. Retrieved from martinFowler.com: https://www.martinfowler.com/articles/injection.html#:~:text=The%20question%20is%3A%20%E2%80%9Cwhat%20aspect,from%20you%20to%20the%20framework

Microsoft. (2024, 18 07). *.NET dependency injection*. Retrieved from learn.microsoft.com: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

Microsoft. (2025, 22 1). *Understand dependency injection basics in .NET*. Retrieved from learn.microsoft: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-basics

Mogylatov, R. (2025, 05 26). *Dependency Injector — Dependency injection framework for Python*. Retrieved from python-dependency-injector.ets-labs.org: https://python-dependency-injector.ets-labs.org/

Spring. (2025, May 25). *Dependency Injection*. Retrieved from docs.spring.io: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html#:~:text=Dependency%20injection%20,or%20the%20Service%20Locator%20pattern

Victor, J. (2023, Sept 19). *Spring Framework, History, and Its Structure*. Retrieved from dev.to: https://dev.to/jeanv0/spring-framework-history-and-its-structure-361

WIKIPEDIA. (2025). *Spring Framework*. Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Spring_Framework

Appendix

Appendix 1 : DI Introduction Code

```
// 1. Define an abstraction
public interface IMessageService
{
    void Send(string message);
}

// 2. Provide a concrete implementation
public class EmailService : IMessageService
{
    public void Send(string message)
    {
        Console.WriteLine($"[Email] {message}");
    }
}

// 3. Provide a concrete implementation
public class SMSService : IMessageService
{
    public void Send(string message)
    {
        Console.WriteLine($"[SMSService] {message}");
    }
}


// 3. Consume the service via constructor injection
public class NotificationManager
{
    private readonly IMessageService _messageService;

// DI happens here: we ask for IMessageService rather than creating a       //
new one ourselves
    public NotificationManager(IMessageService messageService)
    {
        _messageService = messageService;
    }

    public void Notify(string message)
    {
        _messageService.Send(message);
    }
}
```

```
// 4. Composition root: wire up dependencies and run

class Program
{
    static void Main()
    {
        // Manually create the dependency
        IMessageService emailSvc = new EmailService();

        // Inject it into the consumer
        var notifier = new NotificationManager(emailSvc);

        notifier.Notify("Dependency Injection in action!");
    }
}
```

Appendix 2 : DI Payment processor

1. PaymentService.cs

```
public class PaymentService
{
    private readonly IPaymentProcessor _processor;
    private readonly ILogger<PaymentService> _logger;

    public PaymentService(IPaymentProcessor processor, ILoggerFactory
loggerFactory)
    {
        _processor = processor;   // injected here
        _logger = loggerFactory.CreateLogger<PaymentService>();
    }

    public bool Pay(decimal amount, IPaymentDetails details)
    {
        bool result = false; // Initialize result to false
        try
        {
            details.Verify(); // Verify payment details, eg This verifies if
the card number is of the correct format
        }
        catch (ArgumentNullException ex)
        {
            _logger.LogError(ex, "Payment details cannot be null. Please
provide valid payment details.");
            return false; // Return false if details are null
        }
```

```
        if (amount <= 0)
        {
            _logger.LogError("Payment amount must be greater than zero.");
            return result;
        }
        if (IsPaymentDetailsValid(details) == false)
        {
            _logger.LogError("Payment details are invalid.");
            return result;
        }
        try
        {
            _logger.LogInformation("Processing payment of ${Amount}", amount);
            result = _processor.ProcessPayment(amount, details); // The
processor tryes to process the payment
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error logging payment details");
            return false;
        }

        return result; // Return the result of the payment processing

    }

    public bool IsPaymentDetailsValid(IPaymentDetails details)
    {
        // Validate payment details, Program would validate the details based
on the type of payment processor
        // for example, checking if credit card number is valid, by checking
with Visa or Mastercard API, etc.
        // This is a placeholder for actual validation logic.
        return true;
    }

}
```

2. Main.cs

```
static void Main(string[] args)
{
    // 1) Build IConfiguration
    var config = new ConfigurationBuilder()
        .SetBasePath(AppContext.BaseDirectory)
        .AddJsonFile("appsettings.json", optional: false, reloadOnChange:
                                                                    true)
        .Build();
```

```
    // 2) Setup DI container
    var services = new ServiceCollection()
        .AddSingleton<IConfiguration>(config)
        .AddLogging(builder => {
            builder.ClearProviders();
            builder.AddConsole();
            builder.AddDebug();
            builder.AddFilter<ConsoleLoggerProvider>(level => level >=
                                                    LogLevel.Error);
            builder.AddFilter<DebugLoggerProvider>(level => level >=
                                                   LogLevel.Information);
        })
        // bind PaymentConfig from JSON
        .Configure<Configuration.PaymentConfig>(
            config.GetSection("PaymentSettings"))
        .AddSingleton(sp =>
            sp.GetRequiredService<
                IOptions<Configuration.PaymentConfig>>()
                .Value)
        .AddSingleton(typeof(EnvironmentType), sp =>
            Enum.Parse<EnvironmentType>(
                sp.GetRequiredService<IConfiguration>()["Environment"]))
        .AddSingleton<Factories.PaymentProcessorFactory>()
        .AddTransient<App>()   // our entry-point wrapper
        // done
        .BuildServiceProvider();

    // 3) Resolve and run
    var app = services.GetRequiredService<App>();
    app.Run();
}
```

3. App.cs

```
public class App
{
    private readonly PaymentProcessorFactory _factory;
    private readonly ILogger<App> _logger;
    private readonly ILoggerFactory _loggerFactory;

    public App(PaymentProcessorFactory factory,
               ILoggerFactory loggerfactory)
    {
        _factory = factory;
        _loggerFactory = loggerfactory;
        _logger = _loggerFactory.CreateLogger<App>();
    }
```

```
    public void Run()
    {
        _logger.LogInformation("Starting DI Demo…");

        // 1) Select method
        var method = ConsolePrompts.PromptForMethod();
        _logger.LogInformation("User selected {Method}", method);

        // 2) Create processor
        var processor = _factory.Create(method);

        // 3) Read amount
        var amount = ConsolePrompts.PromptForAmount();

        // 4) Read details
        var details = ConsolePrompts.PromptForDetails(method);

        // 5) Inject into service and execute
        var service = new PaymentService(processor, _loggerFactory);
        try
        {
            service.Pay(amount, (IPaymentDetails)details);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An error occurred while processing the
payment.");
        }
    }
}
```

Appendix 3: Payment Service Tests

```
[TestClass]
public class PaymentServiceTests
{
    private Mock<ILoggerFactory> _loggerFactoryMock = null!;
    private Mock<ILogger<PaymentService>> _loggerMock = null!;

    [TestInitialize]
    public void Setup()
    {
        _loggerMock = new Mock<ILogger<PaymentService>>();
        _loggerFactoryMock = new Mock<ILoggerFactory>();
        _loggerFactoryMock
                .Setup(f => f.CreateLogger(typeof(PaymentService).FullName!))
            .Returns(_loggerMock.Object);
    }

    [DataTestMethod]
    [DataRow(0)]
    [DataRow(-10)]
    public void Pay_InvalidAmount_DoesNotCallProcessor(Int32 badAmount)
    {
        // Arrange
        var processorMock = new Mock<IPaymentProcessor>();
        var detailsMock = new Mock<IPaymentDetails>();
        detailsMock.Setup(d => d.Verify());  // no exception

        var service = new PaymentService(
            processorMock.Object,
            _loggerFactoryMock.Object);

        // Act
        var result = service.Pay(badAmount, detailsMock.Object);

        // Assert
        Assert.IsFalse(result, "Pay should return false for non-positive
amounts.");
        processorMock.Verify(
            p => p.ProcessPayment(
                It.IsAny<decimal>(),
                It.IsAny<IPaymentDetails>()),
            Times.Never, "Processor must not be called on invalid amount.");
    }
```

```
    [TestMethod]
    public void Pay_InvalidDetails_DoesNotCallProcessor()
    {
        // Arrange
        var processorMock = new Mock<IPaymentProcessor>();
        var detailsMock = new Mock<IPaymentDetails>();
        // Simulate Verify() throwing for null/invalid details
        detailsMock
            .Setup(d => d.Verify())
            .Throws<ArgumentNullException>();

        var service = new PaymentService(
            processorMock.Object,
            _loggerFactoryMock.Object);

        // Act
        var result = service.Pay(100m, detailsMock.Object);

        // Assert
        Assert.IsFalse(result, "Pay should return false when Verify()
throws.");
        processorMock.Verify(
            p => p.ProcessPayment(
                It.IsAny<decimal>(),
                It.IsAny<IPaymentDetails>()),
            Times.Never, "Processor must not be called on invalid details.");
    }
```

```
    [DataTestMethod]
    [DataRow(true)]
    [DataRow(false)]
    public void Pay_RelaysProcessorResult(bool processorResult)
    {
        // Arrange
        var processorMock = new Mock<IPaymentProcessor>();
        processorMock
            .Setup(p => p.ProcessPayment(
                It.IsAny<decimal>(),
                It.IsAny<IPaymentDetails>()))
            .Returns(processorResult);

        var detailsMock = new Mock<IPaymentDetails>();
        detailsMock.Setup(d => d.Verify());  // valid details

        var service = new PaymentService(
            processorMock.Object,
            _loggerFactoryMock.Object);

        // Act
        var result = service.Pay(50m, detailsMock.Object);

        // Assert
        Assert.AreEqual(
            processorResult,
            result,
            "Pay should return whatever the processor returns when all
preconditions pass.");

        processorMock.Verify(
            p => p.ProcessPayment(50m, detailsMock.Object),
            Times.Once,
            "Processor must be called exactly once with the provided amount
and details.");
    }
}
```