Eric Joseph Kizhakkebhagathu

N11190728@qut.edu.au

Dependency Injection as a Programming Paradigm

CAB402 Assignment 2

Table of Contents

[Dependency Injection as a Programming Paradigm 2](#_Toc199184789)

[Introduction 2](#_Toc199184790)

[History and Adoption of DI 3](#_Toc199184791)

[References 4](#_Toc199184792)

[Appendix 4](#_Toc199184793)

[Appendix 1 : DI Introduction Code 4](#_Toc199184794)

# 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.

# 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.

# References

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. (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

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!");

}

}