SOLID PROGRAMMING

Michael Feathers coined the acronym, **SOLID**, to help us remember the first five object-oriented principles of class design, collected and published by Robert C. Martin[1].

These principles offer flexibility and maintainability. Not only do they help us write code that is easier to understand, piece by piece, they also help us achieve a more holistic comprehension of our applications, and how they behave. Given the years that these principles have had to mature, we benefit from a wealth of tools, and patterns that ease our path to following them.

(1): Robert C. Martin published multiple books and articles discussing SOLID and other principles: http://www.objectmentor.com/PPP/

THE SOLID PRINCIPLES

SRP: The Single Responsibility Principle

OCP: The Open-Closed Principle

LSP: The Liskov Substitution Principle

ISP: The Interface Segregation Principle

DIP: The Dependency Inversion Principle

SINGLE RESPONSIBILITY PRINCIPLE

TOO MANY OPTIONS LEAD TO A BAD DEVELOPER EXPERIENCE

The Single Responsibility Principle: a class should have one, and only one, reason to change[1]

Separation of Concerns: a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability[2]

The Single Responsibility Principle (SRP) has a lot in common with Separation of Concerns. The key differentiator is that the Single Responsibility Principle is focused on class design, where Separation of Concerns is a broader concept that we apply in micro and macro design. The Single Responsibility Principle is a refinement of Separation of Concerns.

A word we use often when developing software that helps us consider SRP, is **scope**. **What is the scope of responsibility for the class I am about to write**? It also helps to envision what a telescope does: it reduces our point of view to a single, focused object.

After reading this article, you should be able to:

- Break a problem into concerns/functionality grouped by category and behavior
- Clearly communicate the responsibility of classes and methods through naming conventions

Start Reading SOLID Part 1: SRP

(1) Robert C. Martin, in Principles of OOD
(2) Bedir Tekinerdogan, in Separation of Concerns

OPEN/CLOSED PRINCIPLE

FAVOR AUGMENTATION OVER OPEN CHEST SURGERY

Software entities [[classes, modules, functions, etc.]] should be open for extension, but closed for modification[1]

The original definition of the **Open/Closed Principal** (OCP) is a bit abstract, but the meaning is rather simple. In the same way that we use scope to clearly communicate and define responsibility, we can also design our code such that it rarely needs to change.

Pay special attention to Solution 2, in this reading. The author notes that OCP complements SRP. When you move on to Interface Segregation, you will find even more complimentary methodology is at play in the SOLID principles.

Ignore the authors rant about interface naming conventions. No one is familiar with all IDE's and color schemes that developers may choose. In fact, many IDEs and syntax themes do not differentiate interfaces from classes with colors.

After reading this article, you should be able to:

- Describe the **Open-Closed Principle** through example
- Indicate whether a given set of code satisfies the **Open-Closed Principle**

Start Reading SOLID Part 2: OCP

(1) Bertrand Mayer, in Object-Oriented Software Construction

LISKOV SUBSTITUTION PRINCIPLE

IF IT LOOKS LIKE A DUCK, BUT DOESN'T QUACK LIKE ONE, THEN THE ABSTRACTION IS NOT A RELIABLE IMPLEMENTATION

Child classes should never break the parent class' type definitions[1]

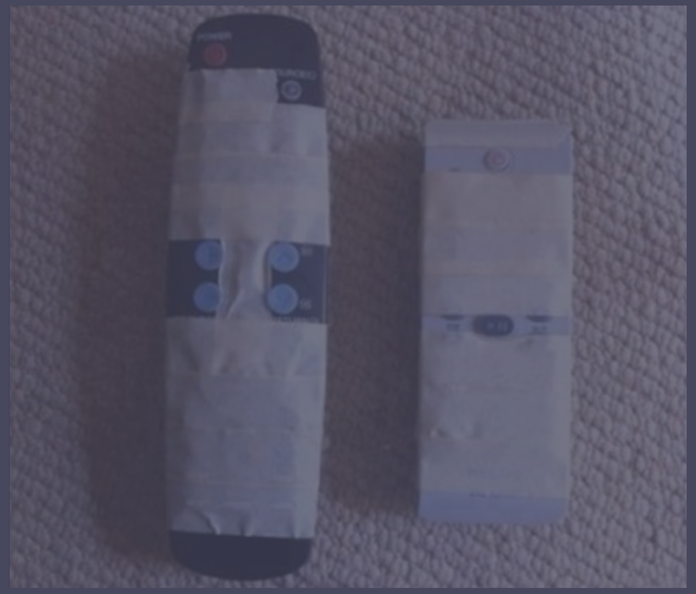
The Liskov Substitution Principle (LSP) proposes that derived classes should not modify the behavior of the classes/interfaces from which they are derived, such that the base class'/interface's output is free of tacit knowledge. In other words, the behavior or scope of responsibility of a given method is always the same, and may not be changed by derived (child) classes.

We'll get to SOLID Part 3 in the next reading.

(1) Patkos Csaba, in SOLID Part 3

INTERFACE SEGREGATION PRINCIPLE

WHAT ARE ALL THESE BUTTONS!?!?



No client should be forced to depend on methods it does not use[1]

The Interface Segregation Principle (ISP) is a refinement of the Single Responsibility principle. We should strive to scope our interfaces such that a caller need only implement the interfaces it needs to perform a unit of work (UOW). Given the right modality, following ISP should also reduce the impacts of change.

After reading this article, you should be able to:

- Describe the **Liskov Substitution Principle**, and what it has in common with the **Single Responsibility** and **Open/Closed** principles
- Describe **Interface Segregation** and what it has in common with the **Single Responsibility** and **Open/Closed** principles
- Break a set of behaviors into multiple interfaces through segregating scope

Start Reading SOLID Part 3: LSP and ISP

(1) Robert C. Martin, in Agile Software Development: Principles, Patterns and Practices

DEPENDENCY INVERSION PRINCIPLE

WOULD YOU PERMANENTLY CONNECT YOUR PHONE TO YOUR COMPUTER?

A. High-level modules should not depend on low-level modules. Both should depend on abstractions[1]
B. Abstractions should not depend on details. Details should depend on abstractions[1]

In industry, you will also hear the **Dependency Inversion Principle** (DIP) referred to as, or in the context of, “Dependency Injection”, and, “Inversion of Control”.

Loose coupling is desirable, whether we are writing small consumer apps, large enterprise apps, middle-ware or third-party toolsets. Loose coupling can be the difference between an organizations ability to maintain and improve their offering(s) or failing to keep pace with the market due to rigidity and lack of flexibility. It can (and should) be a deciding factor in client adoption.

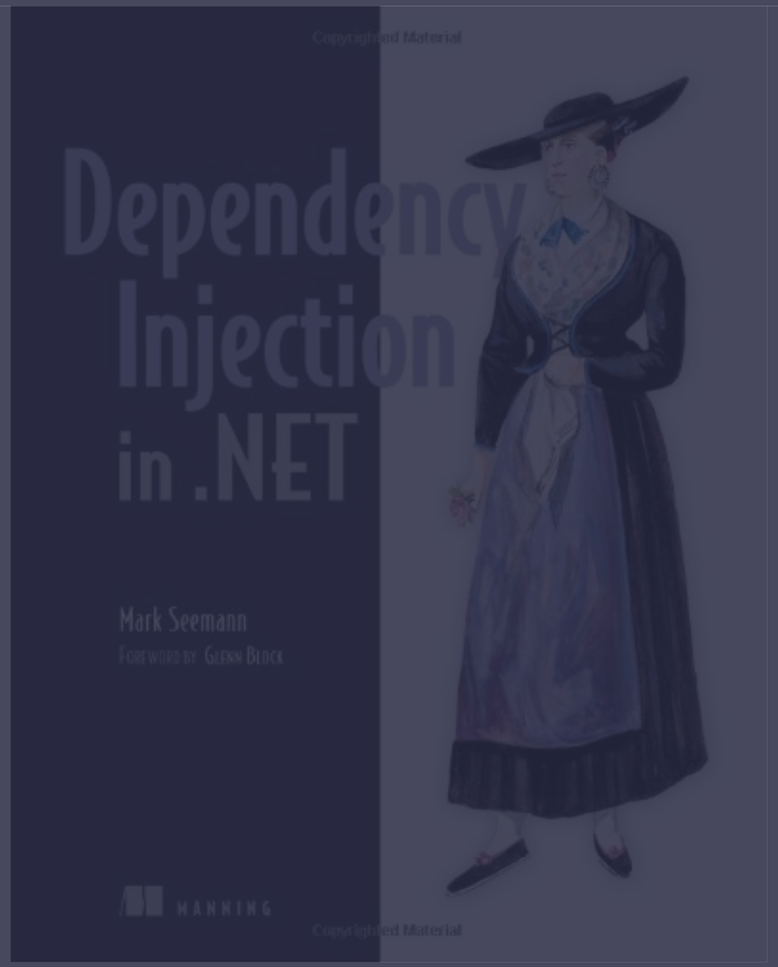
After reading this article, you should be able to:

- Describe **Dependency Inversion**
- Explain how **Dependency Inversion** can lead to loose coupling

Start Reading SOLID Part 4: DIP

(1) Robert C. Martin, in Agile Software Development: Principles, Patterns and Practices

COMPOSITION ROOT



A Composition Root is a (preferably) unique location in an application where modules are composed together[1]

The composition root presents a single place in an application where we can compose the object graphs. What does that mean? The Composition Root, is the place where we will satisfy the dependencies that one object may have on another. It is a single, easy to find location in our app, where we can redefine these dependencies as/if they change in the future.

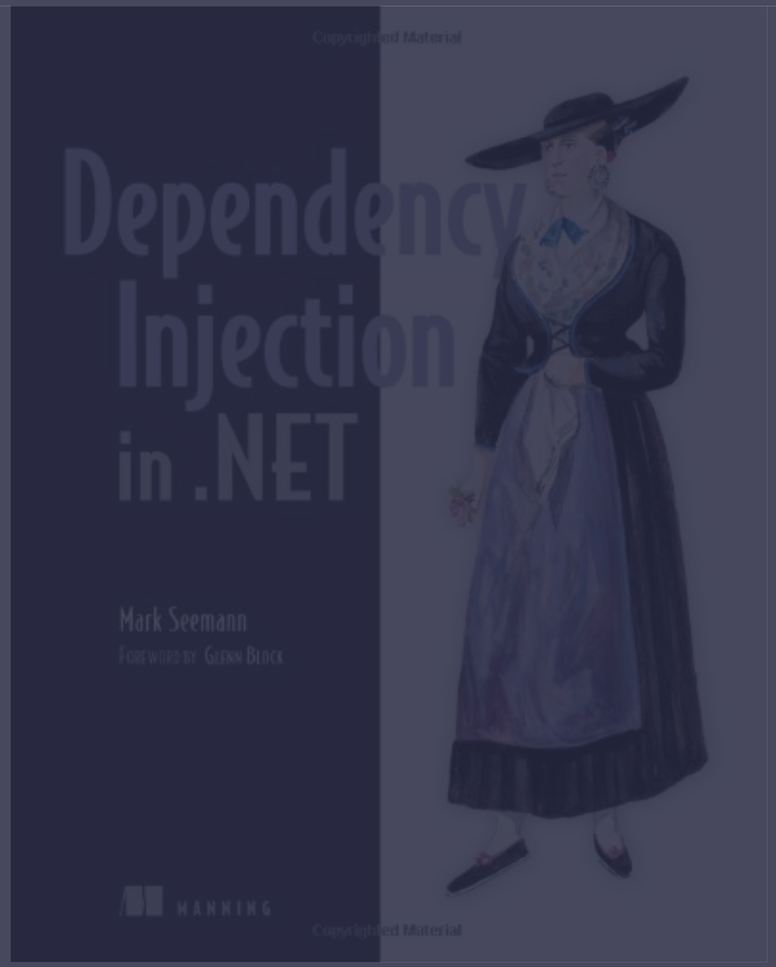
After reading this article, you should be able to:

- Describe Composition Root and where it should be implemented

Start Reading about the Composition Root

(1) Mark Seemann, in Dependency Injection in .NET
(2) Image © Amazon

SERVICE LOCATION



Service location is the opposite of dependency injection. Using service location, a class or module requests a dependency, rather than being injected with it. Read on to learn why this is an **anti-pattern**.

After reading this article, you should be able to:

- Explain the difference between dependency injection and service location

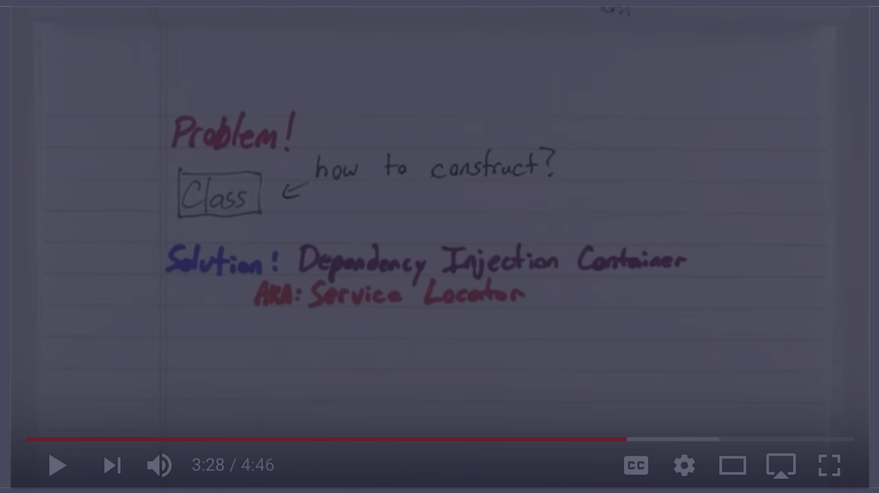
Why should I read this?

- Many developers mistake service location for dependency injection
- Node.js is built heavily on the concepts of service location

Start Reading about Service Location

(1) Mark Seemann, in Dependency Injection in .NET
(2) Image © Amazon

DEPENDENCY INJECTION OR SERVICE LOCATION?



In this video, Anthony Ferrera illustrates dependency injection in a fairly clear way... until he starts explaining the Inversion of Control container.

When describing the Inversion of Control container, is he describing:

(a) Dependency Inversion
(b) Service Location (the opposite of Dependency Inversion)

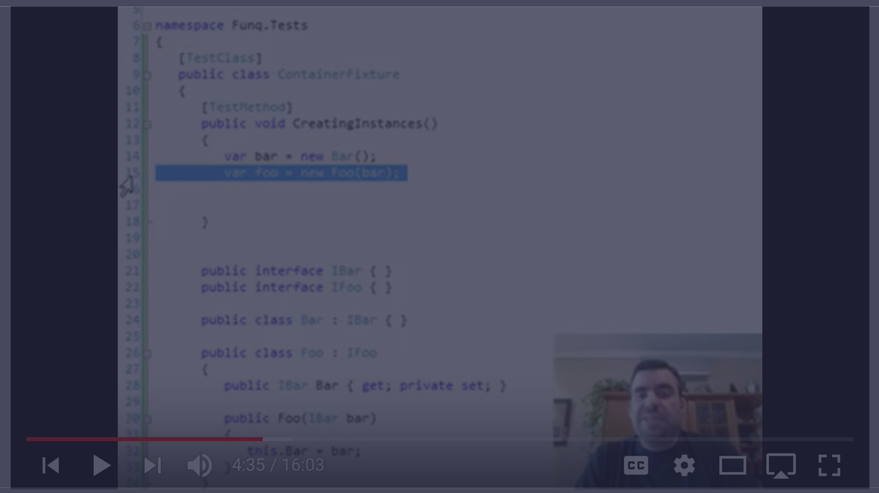
Why should I watch this?

- It's a simple explanation of dependency injection
- It illustrates that it's common for developers to mistake service location for dependency injection

Watch the Video

(1) Mark Seemann, in Dependency Injection in .NET
(2) Image © Amazon

BUILDING AN IOC CONTAINER



In 2009, Daniel Cazzulino recorded himself building the dependency injection container, Funq, using Test Driven Development (TDD). Funq takes a minimalistic approach to dependency injection; one which helps show the magic behind the curtain. This series of videos also demonstrates TDD quite well.

After watching these videos, you should be able to:

- Demonstrate TDD
- Describe a simple Inversion of Control Container / Dependency Injector

Why should I watch these?

- If IoC / DI still seems ambiguous to you, there is a good chance these videos will clear things up for you - they pull back the curtain, so you can see how IoC Containers work.

Watch the Videos

(1) Mark Seemann, in Dependency Injection in .NET
(2) Image © Amazon