slim shady

Table of Contents

[Introduction 2](#_Toc165812385)

[Goal 2](#_Toc165812386)

[Gang of four 2](#_Toc165812387)

[Explain a few in detail with example code 3](#_Toc165812388)

[Singleton pattern (Creational example) 3](#_Toc165812389)

[Observer pattern (Behavioral example) 3](#_Toc165812390)

[Proxy pattern (Structural example) 4](#_Toc165812391)

[Cases from Bestmix 4](#_Toc165812392)

# Introduction

Ladies and gentlemen, imagine a world where every architectural marvel, from the towering skyscrapers of New York to the serene temples of Kyoto, follows a set of universal principles. These principles, known as design patterns, provide architects with a language of solutions to recurring challenges in their craft. But what if I told you that this concept didn't just stay within the realm of buildings and bridges? What if it transcended into the very codes and algorithms that power our digital world?

In 1977, Christopher Alexander, an architect with a keen eye for patterns, introduced a revolutionary idea in his book 'A Pattern Language.' He proposed a language centered around entities called patterns—timeless solutions to architectural problems found across different cultures. These patterns weren't just bricks and mortar; they were the building blocks of universal design thinking.

Fast forward to 1995, a landmark year for software engineering. Four visionaries—Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Also known as ‘the Gang of Four’ —crafted a masterpiece: 'Design Patterns: Elements of Reusable Object-Oriented Software.' This book brought the concept of design patterns into the digital age. Drawing inspiration from Alexander's work in architecture, they observed how software development teams faced similar challenges. Just as a blueprint guides an architect, these design patterns became the blueprint for software engineers, guiding them through the complexities of code.

But what exactly is a design pattern? Imagine it as a roadmap, not for physical structures, but for the structures of code itself. These are tried-and-tested solutions to common programming problems, offering a clear path through the maze of software development. From object composition to class structure, design patterns in languages like C# provide a framework for developers to create more readable, maintainable, and efficient code.

# Goal

Hello, I'm Kjell Coppin, a passionate software engineer with a focus on .NET development. Today, in this TED talk, I aim to empower fellow developers by sharing insights on writing cleaner, more efficient code.

# Gang of four

In the famous book written by the Gang of four, 3 types of design patterns are talked about.

**Creational Patterns**: These patterns focus on the process of object creation, providing mechanisms for creating objects in a manner suitable for a given situation. Examples include Factory Method, Abstract Factory, Singleton, Builder, and Prototype patterns.

**Behavioral Patterns**: Behavioral patterns are concerned with communication between objects, focusing on how objects distribute responsibilities and duties among themselves. Examples include Observer, Strategy, Command, Iterator, and State patterns.

**Structural Patterns**: Structural patterns deal with the composition of classes or objects to form larger structures. They help ensure that if one part of a system changes, the entire system doesn't need to do so. Examples include Adapter, Decorator, Proxy, Composite, and Facade patterns.

# Explain a few in detail with example code

What I’ve explained until now may all seem very abstract, so let me give you a few examples of design patterns.

## Observer pattern (Behavioural example)

An interesting behavioural pattern is the Observer pattern.

Observer lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

**Show notification server example**

As you can see in this C# example, we have a notification server and a phone. In c# the observer pattern is implemented using events and delegates. The event is defined in the notification server, and phones can subscribe to it using += and say what method should run when the event occurs. The delegate notificationhanlder in the notification server serves as a signature for what the handler method in a phone, the ReceiveNotification method in this case, should look like.

All this combined means that when the notification server runs the SendNotification method, all subscribed phones will run their ReceiveNotification method.

You should consider implementing the Observer pattern when you encounter scenarios where changes in one object's state need to trigger actions in other parts of the system, but you want to keep these components loosely coupled.

## Proxy pattern (Structural example)

The proxy pattern is an example of a structural pattern. It lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

The following example has a database service, which has a proxy that does pre and post processing when writing data.

**Show database example**

Here we have our Database service, a simple class containing a method to write to the database. It follows the IDatabaseService interface.

Now this is our proxy class. It follows the same interface as the database service, so the proxy service and actual database service can be used interchangeably. The proxy contains an instance of the database service, and when the proxy’s WriteData method is called, it just calls the database service’s WriteData method. However now you can add whatever logic, pre and post processing in this case around the WriteData method without the caller or database service needing to worry about it.

# Cases from Bestmix

Talk about some cases of design pattern use in the Bestmix codebase (idk yet how many, got to see how big they are).

For every case:

* Give some context about code I am talking about
* Show snippets
* Explain what design pattern we are dealing with
* Why is this a good/bad choice
* What are possible improvements?
* Show improved snippets

## Observer pattern: validator

A

Observer lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.