TS

Mastering Design Patterns in TypeScript

An Approachable Guide

Adegoke Akintoye

Mastering Design Patterns in TypeScript

An Approachable Guide

Adegoke Akintoye

First edition. March 31, 2024.

Juri Books (email: call.juri@outlook.com tel: +2349012885870)

Disclaimer

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Other Books By The Author:

- Mastering JavaScript Array Methods: A Beginner's Guide to Simplifying Array Manipulation
- Mastering JavaScript String Methods: A Beginner's Guide to Simplifying String Manipulation
- Mastering Coding Test: 50 Problems with Solutions
- Mastering Design Patterns in TypeScript: An Approachable Guide
- Object-Oriented Programming In TypeScript: A Beginner's Guide
- Mastering TypeScript: A Beginner’s Guide
- JavaScript: The Ultimate Guide to Interview Questions
- Integrating HTMX with Laravel: An Approachable Guide
- Object-Oriented Programming in PHP:An Approachable Guide
- Functional Programming in TypeScript: An Approachable Guide
- Laravel Guide
- Mastering API Development with Laravel
- HTMX Guide
- Lumen Illuminated
- Easy, Fast, and Practical PWA

Table of Contents

Preface

In the ever-evolving landscape of software development, the pursuit of writing clean, maintainable, and scalable code is a constant endeavor. Design patterns, which originated from the seminal work of the "Gang of Four," have proven to be invaluable tools in achieving this goal. They provide time-tested solutions to recurring problems faced by developers, offering a common language and a structured approach to addressing complex design challenges.

TypeScript, a superset of JavaScript, has gained tremendous popularity in recent years, particularly in the realm of large-scale application development. Its strong typing, object-oriented features, and seamless integration with JavaScript make it an ideal choice for implementing design patterns effectively.

This book, "Mastering Design Patterns in TypeScript: An Approachable Guide," is a concise yet comprehensive resource that aims to demystify design patterns and make them accessible to developers with a basic understanding of TypeScript and object-oriented programming (OOP) concepts. Weighing in at just slightly over 100 pages, this book avoids unnecessary fluff and aims to provide you with a focused and efficient learning experience, respecting your valuable time.

Whether you are a seasoned TypeScript developer or new to the language but have experience with OOP in other programming languages similar to TypeScript, this book will guide you through the intricacies of 23 essential design patterns, providing a practical and straightforward approach to understanding and applying them.

Each chapter is dedicated to a specific design pattern, offering a clear explanation of its purpose, structure, and applicability. The book goes beyond mere theoretical explanations by providing practical implementations in TypeScript, allowing you to see these patterns in action. Additionally, you'll find insightful discussions on the advantages and disadvantages of each pattern, helping you make informed decisions when incorporating them into your projects.

One of the book's standout features is a dedicated chapter that showcases a real-world project, demonstrating how multiple design patterns can be combined to solve complex problems. This practical example will not only solidify your understanding but also inspire you to explore new ways of applying design patterns in your own projects.

Whether you are working on web applications, server-side applications, or any other TypeScript-based project, this book will equip you with the knowledge and tools to write more robust, flexible, and maintainable code. By mastering design patterns, you'll be able to tackle complex challenges with confidence and create software that stands the test of time.

So, embark on this journey of mastering design patterns in TypeScript, and unlock the power of writing truly exceptional code, regardless of whether you are a TypeScript developer or coming from a similar language with some OOP experience. With its concise yet comprehensive approach, this book aims to be an invaluable resource that won't waste your time.

Yor feedback will be highly appreciated. You can reach me here:
call.juri@outlook.com

Chapter 1: Singleton

The Singleton Concept

The Singleton pattern is a design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system. The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element.

Implementing Singleton in TypeScript

In TypeScript, the Singleton pattern can be implemented using a private constructor and a static method. The private constructor ensures that the class cannot be instantiated from outside the class, while the static method ensures that only one instance of the class is created.

Here's a step-by-step guide to implementing the Singleton pattern in TypeScript:

1. **Create a Class with a Private Constructor**: This prevents the direct construction calls with the `new` operator.

```
class Singleton {
    private static instance: Singleton;

    private constructor() {
        // Initialize your instance
    }
}
```

2. **Add a Static Method for Getting the Instance**: This method checks if an instance of the class already exists. If it does, it returns that instance. If it doesn't, it creates a new instance, stores it, and then returns it.

```
class Singleton {
    private static instance: Singleton;

    private constructor() {
        // Initialization code, such as, setting up event listeners, etc.
```

```
    }

    public static getInstance(): Singleton {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }
        return Singleton.instance;
    }
}
```

3. **Accessing the Singleton Instance**: Since the constructor is private, the only way to get the Singleton instance is through the `getInstance` method.

```
const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();

console.log(instance1 === instance2); // true
```

This code demonstrates that `instance1` and `instance2` are indeed the same instance.

Advantages and Disadvantages

Advantages:

- Controlled access to the sole instance.
- Reduced global variables.
- A single point of access for managing the instance.

Disadvantages:

- The Singleton pattern can make unit testing difficult because it introduces a global state into an application.
- It can be challenging to subclass a Singleton.
- Overuse of the Singleton pattern can lead to issues similar to those caused by global variables, making the codebase harder to debug and understand.

Practical Example: Logger

A common use case for the Singleton pattern is creating a logger class. A logger often needs to be a single instance to avoid conflicts and ensure a central point of logging.

```
class Logger {
    private static instance: Logger;

    private constructor() {}

    public static getInstance(): Logger {
        if (!Logger.instance) {
            Logger.instance = new Logger();
        }
        return Logger.instance;
    }

    log(message: string): void {
        console.log(`Log entry: ${message}. Timestamp: ${new Date().toISOString()}`);
    }
}

// Usage
const logger = Logger.getInstance();
logger.log("This is a log message");
```

This `Logger` class ensures that logging is centralized through a single instance, making it easier to manage outputs and configurations.

Conclusion

The Singleton pattern is a powerful tool for ensuring that a class has only one instance and providing a global point of access to it. While it offers several benefits, such as controlled access and reduced global variables, it should be used judiciously to avoid the pitfalls associated with global state.

Appendix

A. TypeScript Basics for Beginners

TypeScript is a powerful superset of JavaScript that adds static types to your code. It helps catch errors early in the development process and enhances code quality and readability. Here's a quick guide to get you started with TypeScript.

A.1 Installing TypeScript

To use TypeScript, you need to install it globally on your machine using npm (Node Package Manager).

```
npm install -g typescript
```

A.2 Basic Types

TypeScript allows you to specify types for variables and function parameters, ensuring that your code behaves as expected.

```
let isDone: boolean = false;
let decimal: number = 6;
let color: string = "blue";
```

A.3 Interfaces

Interfaces in TypeScript are powerful ways of defining contracts within your code as well as contracts with code outside of your project.

```
interface User {
    name: string;
    age: number;
}

function greet(user: User) {
    console.log("Hello, " + user.name);
}
```

A.4 Classes

TypeScript supports modern JavaScript class syntax with additional features like type annotations and modifiers.

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

A.5 Functions

TypeScript functions can be created with type annotations for parameters and return types.

```
function add(x: number, y: number): number {
    return x + y;
}
```

A.6 Enums

Enums allow you to define a set of named constants, making your code more readable and manageable.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

A.7 Generics

Generics provide a way to create reusable components. A component can be used with a variety of types rather than a single one.

```
function identity<T>(arg: T): T {
    return arg;
}
```

A.8 Modules

TypeScript supports ES6 module syntax, allowing you to organize and reuse your code effectively.

```
// SomeModule.ts
export function someFunction() { /* ... */ }
```

```
// OtherModule.ts
import { someFunction } from "./SomeModule";
```

B. Recommended Reading and Resources

- **TypeScript Documentation**: The official TypeScript documentation is an excellent resource for learning and reference.

C. Glossary of Design Patterns and Terms

- **Creational Patterns**: Design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
- **Structural Patterns**: Design patterns that ease the design by identifying a simple way to realize relationships between entities.
- **Behavioral Patterns**: Design patterns that identify common communication patterns between objects and realize these patterns.
- **Singleton Pattern**: Ensures a class has only one instance and provides a global point of access to it.
- **Factory Method**: Defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.
- **Observer Pattern**: Defines a dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Strategy Pattern**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

D. Index

- **Flyweight**: Chapter 10
- **Iterator**: Chapter 17
- **Mediator**: Chapter 22
- **Memento**: Chapter 20
- **Observer**: Chapter 15
- **Prototype**: Chapter 5
- **Proxy**: Chapter 9
- **Singleton**: Chapter 1
- **State**: Chapter 18
- **Strategy**: Chapter 14
- **Template Method**: Chapter 19
- **Visitor**: Chapter 23

This appendix serves as a quick reference and starting point for readers new to TypeScript or those looking to refresh their knowledge. It also provides a roadmap to further explore the rich ecosystem of TypeScript and design patterns.