TED TALK SCRIPT

Real world use of design patterns in .Net applications

Kjell Coppin
Kjellcoppin@gmail.com

Table of Contents

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 code 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 cantered 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.

Behavioural Patterns: Behavioural 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.

Cases from Bestmix

Since I did my internship at Bestmix i will use their codebase to show you how some of these design patterns are used in real production software.

Bestmix recipe management is a software that allows users to accurately manage recipes and generate product specifications.

Observer pattern: validator

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.

In Bestmix there are many entities such as recipes, products, parameters and much more. These entities can be created and edited, which means they need to be validated as well. An entity with invalid data, such as empty mandatory fields should not be saved to the database. For this reason there is a validator class.

The validator class implements the observer pattern twice:

- Entities can subscribe to a validator to get notified when they have been validated
- Validators can subscribe to an entity, so that they get notified of changes in that entity and validate it.

To do this, the validator class has an event and a delegate. In C#, the Observer pattern is implemented using events and delegates, which are built-in language features. Events allow an object to notify other objects when something of interest occurs. Delegates are used to define the signature of the event handler method. So entities can subscribe to the validator's EntityValidated Event to get notified when they have been validated.

Next the validator also has a Subscribe method so that the validator itself can subscribe to entities.

Entity.PropertyChanged is an event that variables have in C# by default and gets invoked when one of their properties changes.

So when the validator is subscribed to an entity, it will validate this entity automatically with the validate method since the validate method is the event handler for entity.propertyChanged.

And after validating, the EnittyValidated event is invoked, notifying the subscribed entity.

One place where this validator is used is in the ChangeMultipleRecipesViewmodel. This viewmodel shows a grid with multiple entities, which you can edit in the grid itself.

In this viewmodel, a validator gets added to each row (with each row representing an entity)

you can see that each row subscribes to its validator and assigns UpdateValidity() as event handler. Then the validator subscribes to the row's entity using its Subscribe method.

All of this means that When an entity is changed in the ChangeMultipleRecipesViewModel, the validator automatically validates it.

And when it is done validating, the row of this entity will automatically update it's isValidated property and potentially block it from being saved to the Db.

While it's true that a direct call to the validator method could bypass the Observer pattern, doing so would tie the entities more closely to the validator, reducing flexibility and making the system less modular.

The Observer pattern, therefore, provides a robust framework for managing dependencies between entities and validators, enhancing the overall design and maintainability of the system.

Strategy pattern: import translations

The goal of this pattern is to enable an object to dynamically change its behaviour at runtime by encapsulating different algorithms or behaviours into separate classes.

It is based on the principle of composition over inheritance, defining a family of algorithms, encapsulating each one, and making them interchangeable.

Various entities in Bestmix contain multilingual data, necessitating translations for their text properties.

These translations can be conveniently imported and exported as CSV files for editing.

The code we will discuss serves as a backend method executor, responsible for updating entities with new translations received from the frontend and then persisting them in the database.

Initially, all entity translations followed a consistent logic due to their uniform multilingual data structure.

However, accommodating a new entity type with a distinct data structure posed a challenge.

My first approach involved modifying the existing code by segregating imported entities into multiple lists,

applying translations to each list separately,

and then merging them back before database persistence.

While functional, this approach was not ideal, as adding a new entity type would necessitate further modifications, bloating the method.

The Strategy Pattern emerged as an elegant solution, enabling the addition of new translation import strategies without altering existing code.

I modularized the logic into multiple ImportStrategies, all implementing the IImportStrategy interface. This interface defines a blueprint requiring only an Execute method.

To prevent code duplication across strategies, ImportStrategies inherit from the BaseStrategy abstract class, housing shared functionality.

The logic previously applied uniformly to all entities is now divided into separate strategies for libraries and other entities.

This modular structure facilitates the seamless addition of new entity-specific strategies, such as for parameters,

which is a type with another structure of multilingual data which I had to add later on.

the GetImportStrategy method determines the appropriate strategy based on the entity type, abstracting the strategy selection process from the executor.

This implementation allows the method executor to delegate work to the respective strategies without needing to know which strategy was being employed.

So by using the strategy pattern, we managed to transform the function executor from a messy method that tries to handle multiple types of entities with the same logic,

to a small clean method that delegates the work to a set of strategy classes that all work trough one interface.

Conclusion

These are just a few examples, but well written code is full of design patterns, and keeping them in mind while writing an application can save you a lot of time and headaches later on, when it comes to maintaining or extending functionality.