DOCUMENTATION REPORT
Part of the BACHELOR DISSERTATION

Design patterns in .Net
Real world use of design patterns in .Net applications

Bachelor	Applied Computer Science
Elective track Graduation	Software Engineer
Academic year	2023 - 2024
Student	Kjell Coppin
Internal coach	Guy Van Eeckhout (Howest)
External promoter	Sander Wallaert (Bestmix)

howest.be

Declaration of confidentiality (only if applicable)

If this documentation report contains confidential information, the following text is included for confidentiality. If not, delete this page.

Confidential up to and including dd/mm/20yy
Important

This documentation report (part of the bachelor dissertation) contains confidential information and/or confidential research results proprietary to Howest or third parties. It is strictly forbidden to publish, cite or make public in any way this report or any part thereof without the express written permission of Howest. Under no circumstance may this report be communicated to or put at the disposal of third parties. Photocopying or duplicating it in any other way is strictly prohibited. Disregarding the confidential nature of this report may cause irremediable damage to Howest. The stipulations mentioned above are in force until the embargo date.

Availability for consultation

12/05/2024

Foreword

To put this paper together I received help from some people, so I would like to thank:

Sander Wallaert, software engineer, team lead and my internship mentor at Bestmix.

Jasper François, software engineer at Bestmix.

Jonas Lampaert, software engineer at Bestmix.

Kjell Coppin, Gent, 11/05/2024

Table of contents

1 Introduction

1.1 General

In software development, knowing design patterns is essential for making strong code. This thesis looks at how to use design patterns in .Net. Design patterns are like recipes for solving common problems in software, making it easier to build flexible and scalable systems.

This thesis is for programmers and students who already know the basics of programming. It's a practical guide to understanding and using design patterns in real-world projects.

Goal of This Paper The goal of this paper is to get developers to think more about design patterns. It's all about showing them that by spending a bit more time on picking the right pattern and building their code around it, their code will be easier to keep up with and add to in the future.

I worked on this thesis during an internship at BestMix Software, where I was a junior software engineer. I used what I learned about design patterns in my work there and looked at how they were used in BestMix's code. This internship gave me real-world context for my research.

1.2 Limiting scope

This thesis focusses specifically on a specific subset of design patterns, the 23 patterns discussed in the book “Design Patterns: Elements of Reusable Object-Oriented Software”. This paper sheds light on the use and practical implementation of these patterns.

1.3 Research question

What are the most important design patterns in programming, and how are they applied in C#?

Throughout this dissertation, we will seek to provide insights into the selection and application of design patterns in C# programming to address common software design challenges and enhance code maintainability, scalability, and flexibility.

1.4 Experiment

To answer the research question, the primary methodological approach involves a comprehensive review and analysis of existing literature on design patterns, particularly focusing on the seminal work "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Additionally, various reputable online sources, such as technical articles, documentation and tutorials.

While analyzing design patterns in the Bestmix codebase, a lot of my information also came from my coworkers.

2 Experiment

2.1 Introduction to design patterns

2.1.1 Origin

The inception of design patterns finds its origins in architecture, notably credited to Christopher Alexander. In his book 'A Pattern Language' published in 1977, Alexander introduced a novel language centred around entities known as patterns. These patterns are defined as solutions to recurring problems, offering universal applicability. Through cross-cultural studies, Alexander observed a finite set of reusable solutions employed across various cultures for similar architectural challenges. [11]

In 1995, software design patterns saw widespread recognition with the publication of 'Design Patterns: Elements of Reusable Object-Oriented Software' by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This book, often cited as a cornerstone in software engineering, is commonly referred to as the 'Gang of Four' (GoF). [11]

To write this book they looked at several software development teams and drew similar conclusions for writing software as Alexander did for architecture. [11]

2.1.2 What is a design pattern

Design patterns are design-level solutions to recurring problems encountered by software engineers, serving as a blueprint for tackling these issues rather than a specific code implementation. They are considered good practice due to their tried-and-tested designs, which enhance the readability and maintainability of the final code. Design patterns are predominantly used in object-oriented programming languages, such as c#. These patterns address different aspects of software design, from class instantiation and object composition to the structure of classes and their interactions. By leveraging design patterns, developers can avoid reinventing the wheel, saving time and effort, and ensuring a more robust and maintainable codebase [1][2].

2.1.3 Anti-patterns

Anti-patterns in software development are solutions that may seem effective initially but lead to more problems over time due to their inability to scale or evolve. They are often mistaken for good practices because they seem logical or straightforward but can increase complexity, reduce maintainability, and decrease reliability. [13]

Examples of anti-patterns include:

- **Spaghetti Code**: Unstructured and tangled code with complex control flow and interdependencies, making it hard to understand, debug, and maintain.
- **God Object**: A single class that performs all tasks in a system, leading to tight coupling, low cohesion, and difficulty in maintenance and scalability.
- **Copy-Paste Programming:** Repeatedly copying and pasting code snippets instead of creating reusable components or functions, leading to code duplication and maintenance issues.
- **The Blob:** A single class or module with too many responsibilities and dependencies, violating the Single Responsibility Principle (SRP) and leading to code that is hard to understand and maintain.

- **The Swiss Army Knife:** Creating overly generic or multipurpose classes or functions that try to do too much, leading to complexity, poor performance, and difficulty in testing and debugging.

Avoiding anti-patterns involves being aware of common pitfalls, using proven design patterns and best practices, breaking down complex problems into smaller parts, and continuously reviewing code. Recognizing and avoiding anti-patterns can lead to more effective, efficient, and maintainable solutions.

2.2 Gang of Four patterns

2.2.1 What is the Gang of Four?

The Gang of Four (GoF) Design Patterns, introduced in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are a collection of 23 design patterns that address common software design problems. These patterns, categorized into Creational, Structural, and Behavioural patterns, provide solutions to recurring issues in software development, promoting code reusability, modularity, and flexibility. The GoF patterns have become a foundational resource for software engineers, offering a catalogue of proven solutions that enhance the quality and maintainability of software systems. Because this book has been so influential, it will be the main source for the research in this paper. [1] [3]

The following section means to give you an idea of the gang of four patterns, what their intent is and how they work. [1][2][5]

2.2.2 Creational patterns

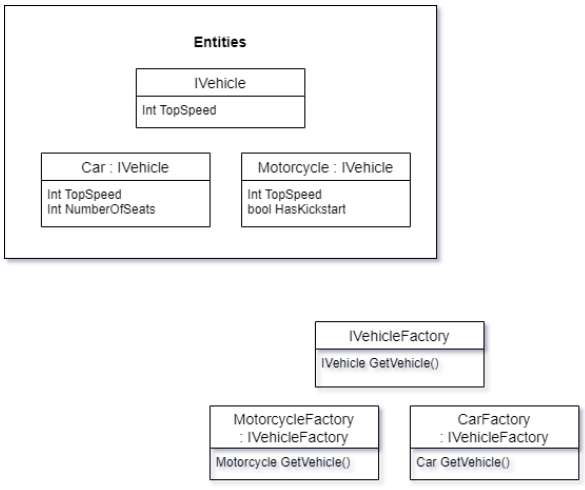
Abstract Factory

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Summary: Abstract factory interface is used by concrete factories to create certain concrete types of an abstract type

Own example:

In this example we have a car and a motorcycle, both implement the IVehicle interface. To create these vehicles we have a separate CarFactory and MotorcycleFactory, but since they both implement IVehicleFactory, We can use the IVehicleFactory and don't need to think about what kind of vehicle we are dealing with. This is useful for example in a class with dependency injection. A vehicle factory is received in the constructor and we can create vehicles regardless of what vehicle. Another benefit is scalability. If tomorrow I want to add a boat which is also an IVehicle, as long as my boat factory implements the IVehicleFactory, The rest of my code will keep working.



Builder

Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Summary: A class can be so large and complex, that using a constructor to create it is very messy. For example the class has a lot of values that might or might not be implemented but creating sub classes for all of these values would be overkill. A builder allows you to build objects step by step and adding what is needed only.

Factory Method

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Summary: The Factory Method design pattern provides an interface for creating objects in a superclass, allowing subclasses to alter the type of objects that will be created. It allows for the creation of objects without specifying the exact class of object that will be created, providing flexibility in object creation. This pattern is especially useful when a class can't anticipate the class of objects it must create.

Prototype

Intent: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Summary: Creational design patterns enable you to duplicate existing objects without tying your code to specific classes. However, directly copying an object without knowledge of its exact type or private fields can be challenging. To ensure objects can be copied, they are designed to inherit an interface with a clone method, allowing them to return a duplicate of themselves.

Singleton

Intent: Ensure a class only has one instance, and provide a global point of access to it.

Summary: Having only one instance of a class can be advantageous, such as ensuring a database is accessed through a single point. This is commonly achieved by having a static instance of the class within the class itself, and then accessing this instance through a static method.

2.2.3 Structural patterns

Adapter

Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Summary: The Adapter Pattern allows incompatible interfaces to work together by creating a bridge between them. It involves a wrapper class, known as the "adapter," that converts the interface of one class into another interface that a client expects. This pattern enables classes

with incompatible interfaces to work together without modifying their source code, promoting code reusability and flexibility in integrating different systems.

Bridge

Intent: Decouple an abstraction from its implementation so that the two can vary independently.

Summary: Sometimes certain fields or logic within a class can be considered a separate entity, in which case it might be better to separate it into its own class and have the original class contain a reference to it. This way, the two classes can evolve separately while remaining connected. The Bridge Pattern achieves this by creating an abstraction (often an abstract class or interface) that contains a reference to the implementor (another class or interface), allowing them to vary independently.

Composite

Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Summary: To create a tree of objects, ensure they all share a common interface for their basic functionality. By allowing objects to "own" other objects, you can form a tree structure where compositions of objects and individual objects can be handled uniformly. This approach enables the manipulation of complex structures as if they were individual objects, simplifying interactions within the tree.

Decorator

Also known as Wrapper

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Summary: The Decorator or Wrapper Pattern works by dynamically adding new behaviors or responsibilities to objects without altering their existing structure. It involves creating a set of decorator classes that are used to wrap concrete components. These decorators implement the same interface as the components they wrap, allowing them to seamlessly substitute for the original objects. Each decorator can add new functionality before or after calling the original component's methods, creating a chain of decorators that can modify the behavior of the base component. This pattern promotes flexibility and extensibility, as new behaviors can be added by creating new decorators rather than modifying the original classes.

Façade

Intent: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use

Summary: The Facade Pattern provides a simplified interface to a complex system, acting as a single entry point for client interaction. It involves creating a facade class that delegates client requests to various components of the system, hiding their complexities. This pattern encapsulates the system's complexities behind a unified interface, making it easier for clients to use

and reducing dependencies between the client code and subsystems. It promotes loose coupling and improves code readability by providing a high-level interface that shields clients from the system's implementation details.

Flyweight

Intent: Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Summary: When dealing with large numbers of objects, you need to minimise RAM usage. This is possible by separating extrinsic (unique) and intrinsic (shared) properties into different objects. This way the intrinsic objects can reuse the extrinsic ones and there is no need to save duplicates.

Proxy

Intent: This pattern 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.

Summary: The Proxy Pattern provides a surrogate or placeholder for another object to control access to it, this placeholder implements the same interface as the original object so the code using it can stay the same. It is used when you want to add a level of indirection to control access to an object, such as for lazy initialization, access control, logging, or caching.

2.2.4 Behavioural patterns

Chain of responsibility

Intent: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Summary: Classes can often be owned by a parent class, and own a child class themselves, creating a chain of classes. With the Chain of Responsibility pattern, you create a linked list of handlers where each handler has a reference to the next handler in the chain. When a request is made, it is passed through the chain until a handler is found that can process it. This allows for dynamic and flexible handling of requests as any handler can either handle the request or pass it to the next handler in the chain, promoting loose coupling and easy addition of new handlers.

Command

Also known as Action or Transaction

Intent: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Summary: The Command Pattern is usually implemented by creating a Command interface or abstract class that defines a method like execute(). Concrete command classes then implement this interface, each encapsulating a specific action. A client object holds a reference to a

command, and when it needs the action to be executed, it calls the execute() method on the command. This decouples the sender of a request from the receiver, allowing for parameterization of clients with different commands, undoable operations, and queuing of requests.

Interpreter

Will not discuss this pattern because it is too specific to certain applications.

Iterator

Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Summary: In more complicated structures like a tree, the traversal behavior of the Iterator Pattern is separated into an iterator object by defining an iterator interface. This interface typically includes methods like hasNext() and next(). Each node in the tree implements a method to create an iterator specific to its traversal logic (e.g., in-order, pre-order). The concrete iterator classes then implement these methods to navigate the tree according to the specified traversal logic. This abstraction allows the client to traverse complex tree structures without knowing the details of traversal, promoting flexibility and code reusability.

Mediator

Intent: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Summary: When using the Mediator Pattern, requests or method calls between classes happen through a mediator class, which will encapsulate the communication logic. This pattern eliminates mutual dependencies between classes that use each other's functionality. Each class only needs to know about the mediator, reducing direct dependencies and promoting loose coupling. The mediator coordinates the interactions between classes, acting as a central hub where objects can communicate without being directly aware of each other.

Momento

Also known as snapshot

Intent: Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Summary: This pattern is implemented by creating a snapshot object which saves the state of your originator object, with methods to get and set the state. Usually a caretaker object is also created, which has methods to create the snapshot and to restore the originator object back to the snapshot. This caretaker often implements the command pattern.

Observer

Also known as publish-subscribe

Intent: Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Summary: The Observer Pattern works by establishing a one-to-many relationship between objects. When the state of one object (the subject) changes, all its dependents (observers) are notified and updated automatically. The pattern involves two main components: the Subject, which maintains a list of observers and notifies them of any state changes, and the Observer, which defines an interface that concrete observers implement to receive updates from the subject.

State

Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Summary: This pattern involves extracting the logic of a class into separate state objects. Each state object contains the original object and assumes its functionality, allowing for different state objects to modify the behavior of the original object. By delegating functionality to different state objects, the original object's behavior can change dynamically based on its current state.

Strategy

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Summary: In a class that performs an algorithm or logic that can be isolated, applying the Strategy Pattern involves isolating this logic into separate strategy objects. By doing so, the class can easily switch between different algorithms or logics by storing a selected strategy as a field and executing it when needed. This approach promotes flexibility and allows the class to adapt to varying requirements without altering its core implementation.

Template method

Intent: Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Summary: This pattern is implemented by splitting an algorithm into multiple methods or steps that are contained in an object. This object now can be extended by different implementations of the algorithms that can overwrite certain steps. This way none of the steps need to be duplicated within the code.

Visitor

Intent: Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

Summary: When you have an algorithm that needs to operate on different types of objects, you can separate the algorithm from the objects by creating a visitor object. This visitor object contains the algorithm's implementations for each type of object. Next, each type of object will have a method, usually called accept, that takes the visitor object as a parameter. This accept method allows the object to receive the visitor and lets the visitor perform its operations on the object.

2.3 A Closer look

This chapter takes an in depth look at a few design patterns. Specifically talking about their implementation in C# and when to use them.

2.3.1 Builder

The goal of the Builder pattern is to separate the construction of a complex object from its representation, allowing the same construction process to create different representations.

When to use

This pattern should be applied in scenarios where objects need to be constructed in various, predefined configurations. It's particularly useful for complex objects with multiple setup options or when the construction process involves multiple steps.

Example in C#

In this example we build motorcycles, these motorcycles can be dirtbikes or sports bikes. These types of motorcycles are built in a different ways so it is very fitting for the builder pattern.

```
public interface IMotorcycleBuilder
{
    IMotorcycleBuilder SetEngine();
    IMotorcycleBuilder SetWheels();
    IMotorcycleBuilder SetHorn();
    Motorcycle Build();
}

// Concrete Builder for DirtBike
public class DirtBikeBuilder : IMotorcycleBuilder
{
    private Motorcycle _motorcycle = new Motorcycle();

    public IMotorcycleBuilder SetEngine()
    {
        _motorcycle.Engine = "single";
        return this;
    }
    public IMotorcycleBuilder SetWheels()
    {
        _motorcycle.Wheels = 2;
        return this;
    }
    public IMotorcycleBuilder SetHorn()
    {
        _motorcycle.Horn = "tuut";
        return this;
    }
    public Motorcycle Build()
    {
        return _motorcycle;
    }
}
```

```
// Concrete Builder for SportsBike
public class SportsBikeBuilder : IMotorcycleBuilder
{
    private Motorcycle _motorcycle = new Motorcycle();

    public IMotorcycleBuilder SetEngine()
    {
        _motorcycle.Engine = "inline-4";
        return this;
    }
    public IMotorcycleBuilder SetWheels()
    {
        _motorcycle.Wheels = 2;
        return this;
    }
    public IMotorcycleBuilder SetHorn()
    {
        _motorcycle.Horn = "bip";
        return this;
    }
    public Motorcycle Build()
    {
        return _motorcycle;
    }
}


// Motorcycle Class
public class Motorcycle
{
    public string Engine { get; set; }
    public int Wheels { get; set; }
    public string Horn { get; set; }
}
```

```
// Usage
IMotorcycleBuilder dirtBikeBuilder = new DirtBikeBuilder();
Motorcycle dirtbike = dirtBikeBuilder
    .SetEngine()
    .SetWheels()
    .SetHorn()
    .Build();
```

A benefit of this pattern is that we can leave out or add steps to the build process as we see fit. For example if our motorcycle doesn't have a horn we can just build it like this.

```
Motorcycle dirtbike = dirtBikeBuilder
    .SetEngine()
    .SetWheels()
    .Build();
```

2.3.2 Singleton

The goal of this pattern is to ensure a class has only one instance and provides a global point of access to it, primarily to control access to shared resources like a database or a file, and to provide a global access point to that instance, offering a safer alternative to global variables. [5]

When to use

When a resource is used throughout your application, and making it available everywhere you need it is challenging, then the singleton pattern can simplify your code. Another reason to use this pattern is when initialization of a resource is expansive, so you want to avoid it being initialized multiple times.

Example in C#

Double Thread locking

For this example, we want to make sure their is only one instance of the dbContext, so that only one connection with the database is made. As in most programming languages, we will create a static getter method that checks if an instance of our class already exists. But to prevent that multiple threads access this method simultaneously and cause it to create multiple instances anyways, we will use double check locking. [6]

```
public class DbContext
{
    private static DbContext? _instance;
    private static readonly object _instanceLock = new object();

    public static DbContext Instance {
        get
        {
            if (_instance == null)
            {
                lock (_instanceLock)
                {
                    if (_instance == null)
                    {
                        _instance = new DbContext();
                    }
                }
            }
            return _instance;
        }
     }
}
```

To optimize performance and resource usage, the double-checked locking pattern first checks if an instance exists without acquiring a lock, and if not, it locks the thread to ensure the instance is created only once, thereby avoiding unnecessary locking when the instance already exists. [7]

Lazy<T>

Another approach to implementing a thread-safe singleton in C# is using the Lazy<T> type, which was introduced in .NET 4.0. This type provides support for lazy initialization in a thread-safe manner, making it a simpler and more modern alternative to the double-checked locking pattern:

```
public class DbContext
{
     private static readonly Lazy<DbContext> _lazyInstance = new
               Lazy<DbContext>(() => new DbContext());

    public DbContext Instance
    {
        get
        {
            return _lazyInstance.Value;
        }
    }
}
```

The Lazy<T> type handles the thread safety for you, ensuring that the singleton instance is created only once in a thread-safe manner [8]

With asp.net core

If you are using the asp.net core framework, we can use the framework's service registration method AddSingleton to add a singleton that can be used for automated dependency injection. [9]

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

          ...

        builder.Services.AddSingleton<DbContext>();

        var app = builder.Build();

        ...

        app.Run();
    }
}
```

Now this singleton can be accessed from anywhere in the project.

2.3.3 Observer

The Observer pattern is a behavioral design pattern that establishes a one-to-many dependency between objects so that when one object (the subject) changes its state, all its dependents (observers) are notified and updated automatically.

This pattern is particularly useful in event-driven architectures and GUI components, allowing objects to communicate changes without being tightly coupled to each other. [10].

When to use

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. Signs that suggest the use of the Observer pattern include situations where multiple objects need to react to changes in a single object, or when you find yourself implementing custom event handling mechanisms to manage communication between objects.

Example in C#

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. This combination makes the Observer pattern both straightforward and idiomatic in C#. [10]

When you declare an event using the event keyword, the .NET runtime automatically manages a list of subscribers for that event. This list is updated whenever a method is subscribed to the event using the += operator and cleared when a method is unsubscribed using the -= operator.

In this example, phones are subscribers that subscribe to a notification server. The sendNotification function is implemented so that every subscribed phone gets the notification.

```
public class NotificationServer
{
    public delegate void NotificationHandler(string notificationText);
    public event NotificationHandler OnNotification;

    public void SendNotification(string notificationText)
    {
        OnNotification?.Invoke(notificationText);
    }
}

public class Phone
{
    public void Subscribe(NotificationServer server)
    {
        server.OnNotification += ReceiveNotification;
    }
    private void ReceiveNotification(string notificationText)
    {
        Console.WriteLine($"New notification: ${notificationText}");
    }
}
```

2.3.4 Strategy

The goal of this pattern is to enable an object to dynamically change its behavior at runtime by encapsulating different algorithms or behaviors 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. [12]

When to use

This pattern should be applied in scenarios where you need to dynamically change the behaviour of an object based on its state. Or when you anticipate new behaviours to be added in the future.
When your code tries to handle multiple states or types with the same code trough complicated if statements, this pattern could be useful.

Example in C#

In this example, we have multiple compression algorithms, but we want to be able to use them interchangeably. To do this we make a strategy interface, which is implemented by both of the concrete interfaces.

```
// Strategy interface
public interface ICompression
{
    void CompressFolder(string folderName);
}
```

```
// Concrete Strategy
public class RarCompression : ICompression
{
    public void CompressFolder(string folderName)
    {
        Console.WriteLine($"{folderName} compressed using RAR.");
    }
}

// Concrete Strategy
public class ZipCompression : ICompression
{
    public void CompressFolder(string folderName)
    {
        Console.WriteLine($"{folderName} compressed using ZIP.");
    }
}

// Context class
public class CompressionContext
{
    private ICompression _compression;

    public CompressionContext(ICompression compression)
    {
        _compression = compression;
    }

    public void SetCompression(ICompression compression)
    {
        _compression = compression;
    }

    public void Compress(string folderName)
    {
        _compression.CompressFolder(folderName);
    }
}
```

This way our strategy context can use any of these strategies without knowing which one it's given.

```
// Usage
CompressionContext context = new CompressionContext(new RarCompression());
context.Compress("Folder1");

context.SetCompression(new ZipCompression());
context.Compress("Folder2");
```

2.3.5 Proxy

The Proxy pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. This pattern is particularly useful for controlling access to complex objects, adding a layer of security, or providing a simplified interface to a complex subsystem. [13][14]

When to use

You should consider implementing the Proxy pattern when:

- You need to add a level of indirection to access to an object, such as logging, caching, or access control.
- You want to provide a simplified interface to a complex subsystem.
- You need to protect an object from misuse or unauthorized access.

Signs that suggest the use of the Proxy pattern include situations where direct access to an object would expose it to risks or complexities that you wish to manage or hide.

Example in C#

In C#, the Proxy pattern can be implemented using interfaces and classes. Here's a simple example demonstrating a proxy for accessing a database:

```
// Interface for the service
public interface IDatabaseService
{
    void WriteData(string data);
}

// Real service implementation
public class DatabaseService : IDatabaseService
{
    public void WriteData(string data)
    {
        // Complex logic to write data to the database
        Console.WriteLine($"Writing data: {data}");
    }
}

// Proxy class
public class DatabaseProxy : IDatabaseService
{
    private readonly IDatabaseService _realService;

    public DatabaseProxy(IDatabaseService realService)
    {
        _realService = realService;
    }

    public void WriteData(string data)
    {
        // Add pre-processing logic
        Console.WriteLine("Pre-processing data...");

        // Call the real service
        _realService.WriteData(data);

        // Add post-processing logic
        Console.WriteLine("Post-processing data...");
    }
}
```

In this example:

- IDatabaseService is the interface defining the operations that can be performed on the database.
- DatabaseService is the real implementation of the service, containing the actual logic to interact with the database.
- DatabaseProxy is the proxy class that implements the IDatabaseService interface. It wraps the DatabaseService and can add additional logic before and after calling the real service, such as logging, caching, or access control.

To use the proxy, you would typically inject the DatabaseProxy into your application instead of directly using the DatabaseService. This allows you to add additional behavior (like logging or caching) without modifying the clients that use the service.

```
public class Application
{
    private readonly IDatabaseService _databaseService;

    public Application(IDatabaseService databaseService)
    {
        _databaseService = databaseService;
    }

    public void ProcessData(string data)
    {
        _databaseService.WriteData(data);
    }
}
```

In this setup, Application uses the IDatabaseService interface, which could be either the DatabaseService directly or the DatabaseProxy depending on how you configure your application. This decouples the Application class from the specific implementation details of the database service, adhering to the Dependency Inversion Principle and making the system more flexible and maintainable.

2.4 Bestmix analysis

In Bestmix's production code, there are a lot of majestically applied design patterns which improve the code in many aspects, let's look at these and see just how they affect the maintainability, scalability and readability . However like with any large application there are also badly applied patterns and anti-patterns to be found in certain places. We will find some specific examples of these as well and find out how a design pattern can improve them.

2.4.1 About Bestmix

Adifo NV is a software development company specializing in providing solutions for the nutrition industries. Founded in 1974 by the De Lille family, it initially focused on developing software applications for payroll and invoicing, which evolved into the MILAS® ERP package. With the acquisition of BESTMIX®, Adifo expanded its offerings to include recipe management for the animal feed industry, later extending to the food industry in the late 1990s. As of 2017, Adifo is part of the Info Support International Group. Following a rebranding initiative in early 2022, all brands are consolidated under BESTMIX Software.

2.4.2 Analysis

In the following chapters I will find and analyze a few cases where a design pattern is used in Bestmix to solve a problem or improve a piece of code. First I will break down cases that were already in Bestmix, explain why and how a design pattern is used.

Next I will find ant-patterns or cases where design patterns could improve the code. I will implement a design pattern and explain why and how along the way.

2.5 Design patterns used in Bestmix

2.5.1 Factory method pattern: filterclauses

Context

In Bestmix, you have the capability to construct and employ queries to pinpoint entities that meet specific conditions. These queries can become quite complex, incorporating various clauses. Given the diversity of clause types, each with its unique viewModel, the factory method pattern is employed to generate these viewModels.

Here's an example of how query with filter clauses appear within Bestmix.

Design pattern application

The Factory Method pattern was chosen to encapsulate the creation logic of different types of FilterClauseViewModel objects, because it enables flexibility, extensibility, and decoupling from client code. Let's break down what exactly the factory method pattern does here.

Factory Method Declaration: The CreateClause() method serves as the factory method. It is responsible for creating instances of FilterClauseViewModel based on different conditions.

Conditional Logic: The method uses conditional logic (in this case, a switch statement) to determine the Property and OperatorValue of the FilterClauseViewModel based on type of IFilterClause.

```
public async Task<FilterClauseViewModel> CreateClause(IFilterClause clause)
{
    // create clause view model
    var clauseViewModel = CreateClause();

    // set Property And OperatorAndValueViewModel
    switch (clause)
    {
        case null:
            return null;
        case EqualToFrameworkFormulaFilter { PropertyName: nameof(QueryProperty.Code) } f:
            {
                var operatorAndValue = CreateStringOperatorAndValue(...);
                operatorAndValue.Operator = EqualToFrameworkFormulaFilterOperator...
                clauseViewModel.Property = AdiFoEnum.GetByName<QueryProperty>(f.PropertyName);
                clauseViewModel.OperatorAndValue = operatorAndValue;
                break;
            }
        case EqualToFrameworkFormulaFilter { PropertyName: nameof(QueryProperty.Folder) } f:
            {
                var operatorAndValue = CreateForeignKeyOperatorAndValue<ListFolder>(...);
                clauseViewModel.Property = AdiFoEnum.GetByName<QueryProperty>(f.PropertyName);
                clauseViewModel.OperatorAndValue = operatorAndValue;
                break;
            }
        case ...
     }
}


public FilterClauseViewModel CreateClause()
{
    var filterClauseViewModel = new FilterClauseViewModel(queryProperties);
    return filterClauseViewModel;
}
```

Encapsulation: The creation of FilterClauseViewModel objects is encapsulated within the factory method. This encapsulation hides the details of object creation from the client code, promoting loose coupling and flexibility.

Conclusion

the Factory Method pattern is a well-suited choice for the scenario described. It enhances the system's flexibility, extensibility, and maintainability by encapsulating the creation logic of FilterClauseViewModel objects. This approach not only facilitates the addition of new clause types but also ensures that the system remains modular and easy to understand, adhering to good software design principles.

2.5.2 Observer pattern: validator

Context

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.

Design pattern application

The observer pattern is implemented in 2 ways here, the validator subscribes to an entity, so that the entity is validated the moment it changes. But an entity can also subscribe to the validator, so that it knows and can react to it being validated.

To do this, the validator class has an event and a delegate (as in 2.3.3). So that entities can subscribe to it.

```
public event EntityValidatedEventHandler EntityValidated;
public delegate void EntityValidatedEventHandler(object sender, T entity, string fieldPath);
```

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.

Result	Action	Generated code	Description	Folder	Template	Animal type	Site	Version
=	abc	abc	abc	abc	abc	abc	abc	=
✓	'Create	Auto-generate		001 All	AD_Default - % Rec...	006 - 11 Pigs - Co...	STBA SITE - Kaprijke	1
✓	'Create	Auto-generate		001 All	AD_Default - % Rec...	001 - 10 Pigs - Co...	03 - 003 Meppel NL	1

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

```
protected List<IValidator<TDomain>> CreateValidators(IList<CopyMultipleRow<TDomain>> rows)
{
    var validators = new List<IValidator<TDomain>>();
    foreach (var row in rows)
    {
        var validator = new Validator<TDomain>(this).AddDefaultRules();
        validators.Add(validator);

        validator.EntityValidated += (sender, entity, fieldPath) => row.UpdateValidity();
        validator.Subscribe(row.Entity);

    }
    return validators;
}
```

In the 2 highlighted lines you can see that first the row subscribes to its validator and assigns UpdateValidity() as event handler. Then the validator subscribes to the row's entity using its own Subscribe method. This subscribe method looks like this:

```
public void Subscribe(T entity, ...)
{
        entity.PropertyChanged += Validate

        // extra logic
}
```

Entity.PropertyChanged is an event that variables have by default in C#, and the validator subscribes to it, using Validate as event handler. So now when the entity is changed, the validate method will run..

```
public async Task Validate(T entity, ...)
{
    // validate the entity

    EntityValidated?.Invoke(this, entity, ...);
}
```

The validate method then triggers the EntityValidated, so now the row knows it has been validated and it's handler method can set the isValid parameter and potentially block it from being saved to the database.

```
public void UpdateValidity()
{
    IsValid = Entity.IsValid && PermissionSet.HasDataPermission(Entity, ...)
}
```

Conclusion

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.

2.6 Improvements with patterns

2.6.1 Strategy pattern: import translations

Context

In Bestmix, various entities such as libraries, parameters, and recipes 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.

Problem

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.

```
public override async Task<FunctionResponse> Execute(FunctionArgument args)
{
    // get old entities
    var entities = await repo.GetByIds(importData.Entities.Keys);


    // split regular entities and new "library field list" entities
    List<IDomainObject> fieldListEntities = new();
    if (importData.FieldTokenName == BestmixToken.Field_TextContent.Name)
    {
        fieldListEntities = entities.Where(e => e is Library lib && lib.IsFieldList).ToList();
        entities.RemoveAll(e => e is Library lib && lib.IsFieldList);
    }

    // save changes to regular entities
    foreach (var entity in entities)
    {
        // update translations in this entity
    }

    // save changes new "library field list" entities
    foreach (Library entity in fieldListEntities)
    {
        // update translations in this library field list entity

    }


    // merge entities and libraray field list entities back into one list
    entities.AddRange(fieldListEntities);
    await repo.Save(entities, true, false,
                        EntityFieldMode.SpecifiedFields, specifiedFields.ToArray());

    return FunctionResponse.Ok();
}
```

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

Design pattern application

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.

```
interface IImportStrategy
{
    public Task<FunctionResponse> Execute(ImportExportTranslationField importData, IEntityRepos-
    itory repo,);
}
```

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

```
abstract class BaseImportStrategy : IImportStrategy
{
    public abstract Task<FunctionResponse> Execute(...);

    protected async Task SaveToDb(...)
    {
        // save to db logic
    }

    // other shared methods
}
```

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

```
class DefaultImportStrategy : BaseImportStrategy
{
        public async override Task<FunctionResponse> Execute(..)
        { // default import logic }
}

        class LibraryImportStrategy : BaseImportStrategy
        {
                public async override Task<FunctionResponse> Execute(...)
                { // library import logic }

                // library specific helper methods
        }
```

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

```
class ParameterImportStrategy : BaseImportStrategy
{
        public async override Task<FunctionResponse> Execute(...)
        { // parameter import logic }

        // Parameter specific helper methods
}
```

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

```
private IImportStrategy GetImportStrategy(AdifoDomainType type)
{
    return type.Name switch
    {
        nameof(BestmixDomainType.Library) => new LibraryImportStrategy(),

        nameof(BestmixDomainType.Parameter) => new ParameterImportStrategy(),

        _ => new DefaultImportStrategy(),
    };
}
```

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

```
[ExecutesFunction(nameof(ApiFunction.ImportTranslations))]
public class ImportTranslations : BestmixFunctionExecutor<IDomainObject>
{
    public override async Task<FunctionResponse> Execute(FunctionArgument args)
    {
        var importData = ...
        var domainType = AdifoDomainType.GetByType(Entity.GetType());
        var repo = GetRepository(domainType);

        IImportStrategy strategy = GetImportStrategy(domainType);

        return await strategy.Execute(importData, repo, domainType, this);
    }
}
```

conclusion

The Strategy Pattern currently enhances the Bestmix system's handling of multilingual data across various entity types by encapsulating translation logic into separate strategies. This approach currently improves code reusability, maintainability, and extensibility, allowing for the easy addition of new translation strategies without altering existing code.

3 Conclusion

In conclusion, this paper underscores the pivotal role of design patterns in software development. By leveraging established design patterns and adhering to best practices, developers can craft codebases that are not only easier to maintain but also more efficient.

Moreover, the paper emphasizes the importance of decomposing intricate problems into manageable components and regularly scrutinizing code to identify and rectify anti-patterns. It advocates for developers to dedicate time to selecting appropriate design patterns and structuring their code accordingly, fostering simpler maintenance and scalability down the line.

Ultimately, embracing design patterns facilitates the creation of robust, adaptable software solutions poised for success in an ever-evolving technological landscape.

AI Engineering Prompts

I used 2 ai chatbots: ChatGPT and Phind. A lot of the following prompts were entered in both chatbots so I could pick the best response.

Preloading the AI

I preloaded the conversations with this prompt to ensure the AI was aware of the topic:

> *The following questions are for a research paper about design patterns in software engineering, more specifically .Net*

AI prompts

> *explain in 3 sentences what the * design pattern does, be concise*
>
> *explain how the * design pattern is implemented concretely*
>
> *make the content and meaning of this text clearer …*
>
> *are there any mistakes in this explanation of the * design pattern? …*
>
> *how does the iterator pattern work in case of a tree data structure*
>
> *give me examples of anti-patterns*
>
> *break down this piece of code, which elements show the * design pattern being used?*

(*) this means that I used this prompt for multiple design patterns

(…) this means that I inserted a piece of text from the paper

Bibliography

[1] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable Object-Oriented software*. 1994.

[2] S. Rahman, "The 3 Types of Design Patterns All Developers Should Know (with code examples of each)," *freeCodeCamp.org*, Jul. 24, 2019. https://www.freecodecamp.org/news/the-basic-design-patterns-all-developers-need-to-know/ (accessed Mar. 24, 2024).

[3] GfG, "Gang of Four (GOF) Design Patterns," *GeeksforGeeks*, Sep. 27, 2023. https://www.geeksforgeeks.org/gang-of-four-gof-design-patterns/ (accessed Mar. 24, 2024).

[4] TechWebDots, "Design Patterns in C#," *YouTube*. Mar. 27, 2020. Accessed: Mar. 24, 2024. [YouTube Video]. Available: https://www.youtube.com/watch?v=bzQeMqcgILY&list=PLBEm2Vv2nD-Ppk8U_LaR8wXl47kgCI8Dl&ab_channel=TechWebDots

[5] "Design Patterns," *Refactoring.guru*, 2014. https://refactoring.guru/design-patterns (accessed Mar. 24, 2024).

[6] M. Alle, "Singleton Design Pattern In C#," *C-sharpcorner.com*, 2023. https://www.c-sharpcorner.com/UploadFile/8911c4/singleton-design-pattern-in-C-Sharp/ (accessed Mar. 24, 2024).

[7] BillWagner, "lock statement - synchronize thread access to a shared resource - C#," *Microsoft.com*, Apr. 28, 2023. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock (accessed Mar. 24, 2024).

[8] dotnet-bot, "Lazy Class (System)," *Microsoft.com*, 2024. https://learn.microsoft.com/en-us/dotnet/api/system.lazy-1?view=net-8.0 (accessed Mar. 24, 2024).

[9] Rick-Anderson, "Dependency injection in ASP.NET Core," *Microsoft.com*, Nov. 07, 2023. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0 (accessed Mar. 24, 2024).

[10] L. Tutor, "Observer Pattern in C#: From Basics to Advanced - Laks Tutor - Medium," *Medium*, Aug. 17, 2023. https://medium.com/@lexitrainerph/observer-pattern-in-c-from-basics-to-advanced-ea4b2d748e (accessed Mar. 24, 2024).

[11] "Coding Games and Programming Challenges to Code Better," *CodinGame*, 2018. https://www.codingame.com/playgrounds/503/design-patterns/origin-of-design-patterns (accessed Apr. 08, 2024).

[12] A. Krishna, "A Beginner's Guide to the Strategy Design Pattern," *freeCodeCamp.org*, May 04, 2023. https://www.freecodecamp.org/news/a-beginners-guide-to-the-strategy design-pattern/#:~:text=The%20Strategy%20Design%20Pattern%20works%20by%20separating%20the%20behavior%20of,it%20through%20a%20common%20interface. (accessed Apr. 20, 2024).

[13] Wikipedia Contributors, "Anti-pattern," *Wikipedia*, Apr. 07, 2024. https://en.wikipedia.org/wiki/Anti-pattern (accessed Apr. 21, 2024).

[14] J. Singh, "Proxy Design Pattern Using C#," *C-sharpcorner.com*, 2017. https://www.c-sharpcorner.com/UploadFile/b1df45/proxy-design-pattern-using-C-Sharp/ (accessed May 05, 2024).

[15] C. Eve, “Design patterns in C# - The Proxy Pattern,” *Endjin.com*, Dec. 07, 2020. https://endjin.com/blog/2020/12/design-patterns-in-csharp-the-proxy-pattern (accessed May 05, 2024).

Appendices

All attachments are collected here. Appendices are intended for readers interested in more detailed information about your topic.