Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Proiect de an**

**Disciplina:** TMPS

**A efectuat:**  Bodarev Victor, TI-202

**A verificat**  asistent universitar, Gaidău Mihai

**Chișinău 2023**

# Summary

[Summary 1](#_Toc137075569)

[Introduction 2](#_Toc137075570)

[1. General description 3](#_Toc137075571)

[Creational patterns 3](#_Toc137075572)

[Structural patterns 7](#_Toc137075573)

[Behavioral patterns 11](#_Toc137075574)

[2. Implemented examples 16](#_Toc137075575)

[Creational patterns 16](#_Toc137075576)

[Structural and Behavioral patterns 20](#_Toc137075577)

[Conclusion 24](#_Toc137075578)

# Introduction

In the discipline of software engineering, the application of design patterns is crucial for developing robust and maintainable software solutions. As part of my final year project in the "Techniques and Mechanisms of Software Design" course, I undertook the task of implementing a blog application in C#. Throughout the project, I incorporated various creational, structural, and behavioral design patterns, namely Singleton, Prototype, Builder, Proxy, Adapter, Strategy, and Visitor.

The objective of this project was to showcase the practical implementation of these design patterns and highlight their significance in creating flexible, scalable, and efficient software systems. By leveraging these patterns, I aimed to enhance the overall design quality and promote code reusability in the blog application.

For the creational design patterns, I implemented the Singleton pattern, ensuring that only one instance of a particular class is created and providing a global point of access to it. The Prototype pattern allowed for creating clones of existing objects, reducing the need for costly object creation. Additionally, the Builder pattern facilitated the construction of complex objects step by step, providing a flexible approach to object creation.

In terms of structural design patterns, I utilized the Proxy pattern, which acted as a surrogate for another object, controlling access to it. This pattern was useful in scenarios where indirect or controlled access to objects was required. Furthermore, the Adapter pattern was implemented to enable the collaboration between incompatible interfaces, promoting the reusability of existing code.

Regarding behavioral design patterns, the Strategy pattern allowed for the encapsulation of interchangeable algorithms, enabling dynamic selection and execution of algorithms at runtime. This pattern enhanced the flexibility and extensibility of the blog application. Additionally, the Visitor pattern facilitated the separation of algorithms from the objects they operate on, promoting clean and modular code structure.

Throughout the project, I focused on effectively implementing these design patterns within the context of the blog application. By incorporating them, the application exhibited improved code maintainability, flexibility, and performance.

In conclusion, this project provided a comprehensive exploration of creational, structural, and behavioral design patterns in the implementation of a blog application using C#. By successfully incorporating the Singleton, Prototype, Builder, Proxy, Adapter, Strategy, and Visitor patterns, I showcased their practicality and significance in developing scalable and maintainable software systems. This project served as a valuable learning experience, solidifying my understanding of design patterns and their role in software engineering.

# General description

## Creational patterns

Creational patterns are a set of design patterns in software engineering that provide guidelines for creating objects or instances of classes. They aim to address the process of object instantiation, hiding the complexities involved and promoting flexibility and reusability in object-oriented systems. Creational patterns are used to ensure that object creation is done in a controlled and standardized manner, making it easier to manage and maintain the codebase.

There are several reasons why creational patterns are needed in software development:

1. Encapsulating object creation logic: Creational patterns help encapsulate the details of object creation within dedicated classes or methods. This promotes loose coupling between objects and makes the code more modular and maintainable. Changes in object creation logic can be localized to the pattern implementation, without affecting the rest of the codebase.
2. Hiding complexity: Creating objects can involve complex processes, such as initializing dependencies, performing configuration, or managing resource allocation. Creational patterns abstract away this complexity, providing a simplified interface for object creation. This allows developers to focus on the essential aspects of object usage, without being burdened by intricate instantiation details.
3. Promoting flexibility and extensibility: Creational patterns enable flexibility in creating objects by decoupling the code from specific classes or implementations. This allows for easy substitution of different object types or configurations without modifying the client code. It promotes extensibility by providing a framework to introduce new types of objects or variations in the future.
4. Supporting dependency injection: Many creational patterns facilitate the use of dependency injection, a technique that promotes loose coupling and modularity. By centralizing the creation and management of dependencies, these patterns help improve testability, maintainability, and scalability of the code.
5. Controlling object creation: Creational patterns provide mechanisms to control the creation of objects, enforcing certain constraints or ensuring a specific number of instances are created. This can be helpful in scenarios where limited resources need to be managed effectively or when a specific object lifecycle needs to be enforced.

Some commonly used creational patterns include:

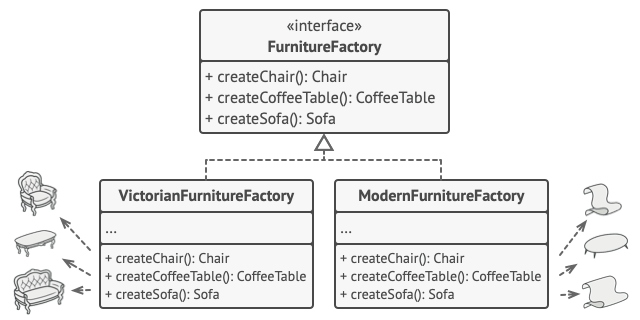
1. Singleton Pattern: The Singleton pattern ensures that only one instance of a class is created throughout the application. It involves a static method or property that returns the same instance of the class every time it is called. This pattern is useful when you want to limit the number of instances of a class and provide global access to that single instance.



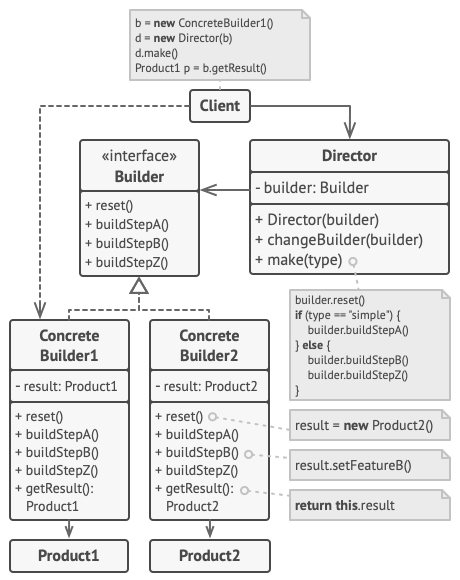
1. Factory Method Pattern: The Factory Method pattern provides an interface for creating objects, but lets subclasses decide which class to instantiate. It encapsulates the object creation logic in a separate method, allowing the subclasses to redefine or extend the instantiation process. This pattern is useful when you want to delegate the responsibility of object creation to subclasses, while ensuring the client code remains unaware of the concrete classes being instantiated.



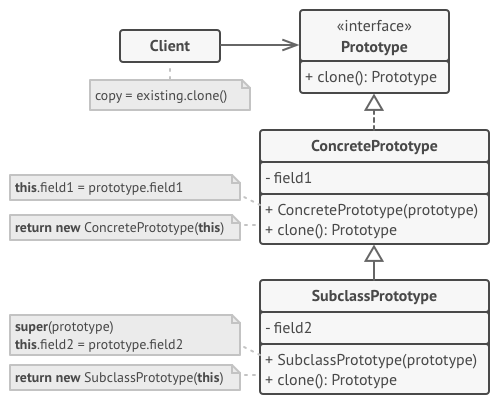
1. Abstract Factory Pattern: The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It allows you to create objects that are part of a specific family or group, ensuring that the created objects are compatible and work together seamlessly. This pattern is useful when you need to create multiple related objects, but want to abstract the client code from the specific implementations.



1. Builder Pattern: The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations. It involves a separate builder class that controls the construction step by step and provides methods to set or configure various properties of the object being built. This pattern is useful when you need to create complex objects with multiple configuration options, while keeping the construction process consistent.



1. Prototype Pattern: The Prototype pattern allows you to create new objects by cloning existing ones. It involves defining a prototype object that serves as a template and creating new instances by copying the prototype. This pattern is useful when creating objects is expensive or complex, and you can benefit from cloning existing objects rather than creating new ones from scratch.



By leveraging creational patterns, developers can improve code maintainability, flexibility, and modularity while reducing the dependencies and complexities associated with object creation. These patterns have proven to be valuable tools in software engineering, providing guidance and best practices for creating objects in a structured and efficient manner.

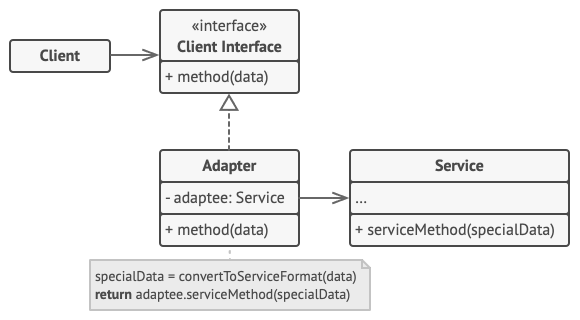
## Structural patterns

Structural patterns are a category of design patterns in software engineering that focus on the composition of classes and objects to form larger structures or systems. These patterns provide solutions for organizing classes and objects to achieve desired functionality while promoting code reusability, flexibility, and maintainability. They primarily deal with relationships between objects, how they are connected, and how they work together to form larger components.

Here are some commonly used structural patterns along with a detailed explanation of each:

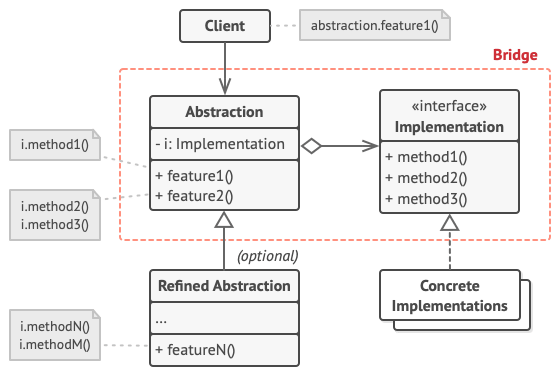
1. Adapter Pattern:

The Adapter pattern allows objects with incompatible interfaces to work together by creating a bridge between them. It wraps one object with another object that presents a standard interface that the client code expects. This pattern is useful when integrating existing or third-party code into a system or when two incompatible interfaces need to collaborate.



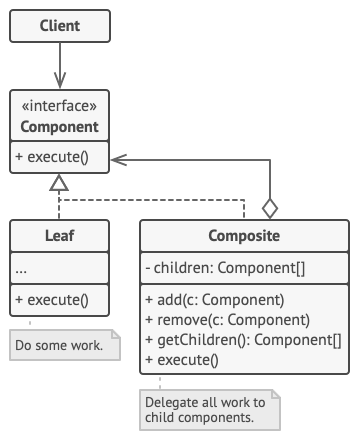
1. Bridge Pattern:

The Bridge pattern decouples an abstraction from its implementation, allowing them to vary independently. It involves creating separate class hierarchies for the abstraction and its implementation, connecting them through composition. The Bridge pattern promotes flexibility by enabling changes to either the abstraction or the implementation without affecting each other.



1. Composite Pattern:

The Composite pattern represents a part-whole hierarchy of objects, treating individual objects and groups of objects uniformly. It allows clients to work with individual objects or collections of objects in a consistent manner. The pattern is useful when there is a need to represent hierarchical structures and apply operations uniformly across the hierarchy.



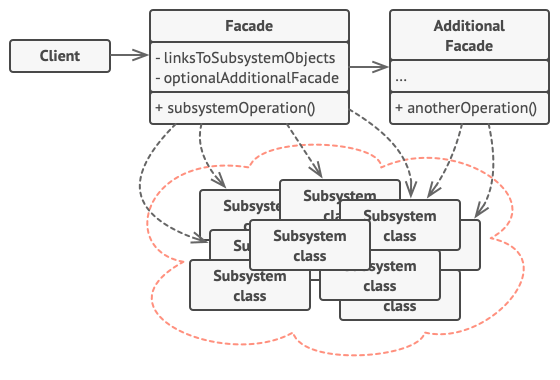
1. Decorator Pattern:

The Decorator pattern dynamically adds new behaviors or responsibilities to objects by wrapping them with additional functionality. It provides a flexible alternative to subclassing for extending object behavior. The Decorator pattern allows for the addition of new features at runtime, without affecting the existing codebase.



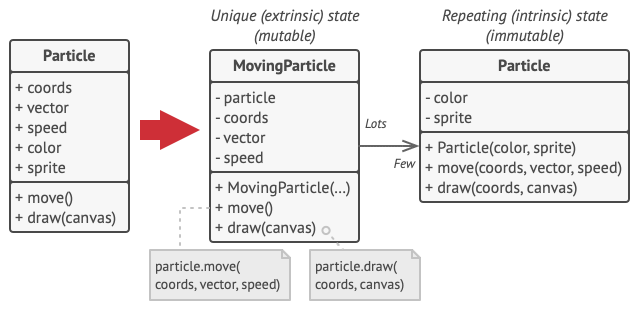
1. Facade Pattern:

The Facade pattern provides a unified interface to a set of interfaces in a subsystem. It simplifies the usage of complex subsystems by providing a high-level interface that shields clients from the underlying complexity. The Facade pattern promotes loose coupling and improves code readability and maintainability.

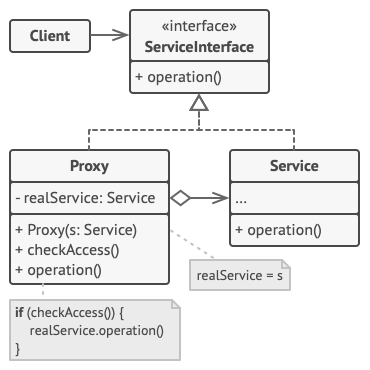


1. Flyweight Pattern:

The Flyweight pattern minimizes memory usage by sharing common state between multiple objects instead of storing it redundantly. It is useful when there is a large number of similar objects that can be effectively shared to reduce memory overhead. The pattern separates the intrinsic and extrinsic state of objects, where intrinsic state is shared, and extrinsic state can vary.



1. Proxy Pattern:



The Proxy pattern provides a surrogate or a placeholder for another object to control access to it. It acts as an intermediary between the client and the real object, providing additional functionality or controlling access permissions. The Proxy pattern is useful when there is a need for additional layers of indirection, such as lazy initialization, access control, or caching.

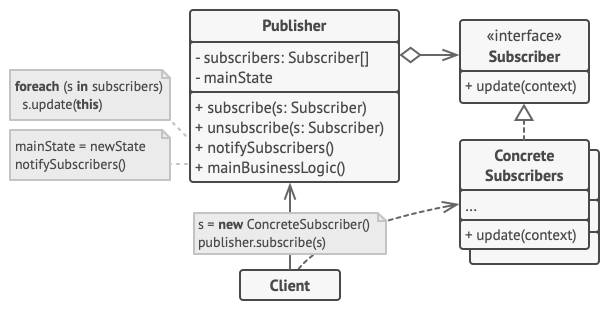
These structural patterns provide solutions for organizing classes and objects in a way that promotes reusability, flexibility, and maintainability. By leveraging these patterns, developers can effectively manage complex relationships between objects, achieve separation of concerns, and build scalable and modular software systems. Each pattern offers a specific solution for a particular set of problems, and understanding their principles and application can significantly enhance software design and architecture.

## Behavioral patterns

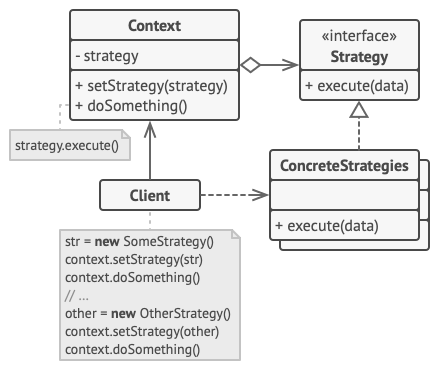
Behavioral patterns are a category of design patterns in software engineering that focus on the interaction and communication between objects. These patterns provide solutions for designing and implementing the behavior and communication patterns among classes and objects. They address how objects collaborate and distribute responsibilities to achieve specific functionalities and behaviors in a system. Behavioral patterns emphasize the dynamics of the system rather than its structure, and they play a crucial role in defining the interactions and flow of control within an application.

Let's explore some commonly used behavioral patterns along with detailed explanations of each:

1. Observer Pattern: The Observer pattern establishes a one-to-many dependency between objects, so that when one object changes its state, all its dependents are automatically notified and updated. It promotes loose coupling between objects, allowing for flexibility and maintainability. The Observer pattern is commonly used in event-driven systems and graphical user interfaces (GUIs).



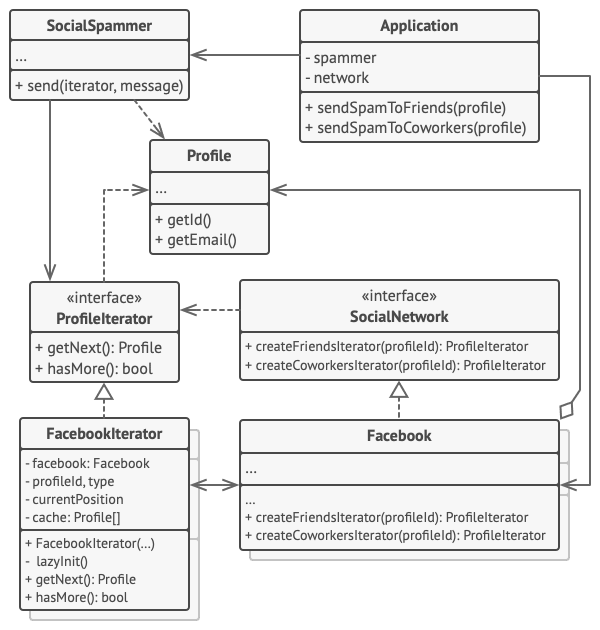
1. Strategy Pattern: The Strategy pattern defines a family of interchangeable algorithms or behaviors encapsulated in separate classes, allowing clients to dynamically select and use different algorithms at runtime. It promotes flexibility and modularity by separating the behavior from the context that uses it. The Strategy pattern is beneficial when there are multiple algorithms or variations of a behavior, and the selection of a specific algorithm needs to be dynamic.



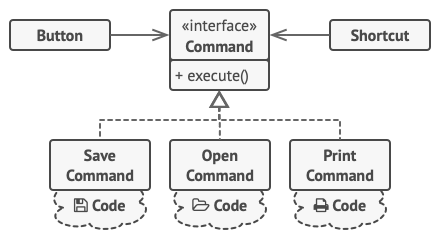
1. Template Method Pattern: The Template Method pattern defines the skeleton of an algorithm in a base class, allowing subclasses to provide or override specific steps of the algorithm as needed. It promotes code reuse by encapsulating the common parts of the algorithm in the base class while allowing subclasses to customize certain steps. The Template Method pattern is useful when there is a common algorithm structure, but the implementation of certain steps may vary.



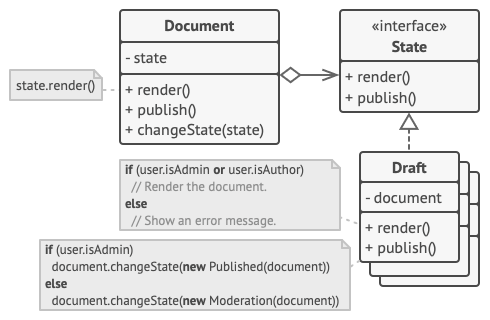
1. Iterator Pattern: The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It decouples the traversal logic from the aggregate object, allowing for different traversal mechanisms. The Iterator pattern simplifies the iteration process and provides a consistent interface for iterating over different types of collections.



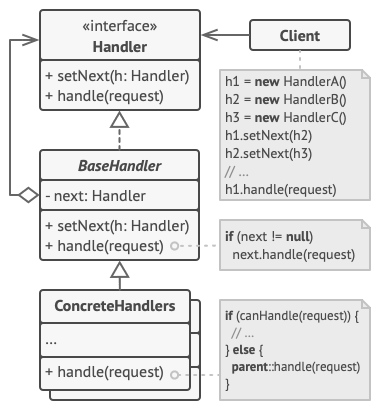
1. Command Pattern: The Command pattern encapsulates a request as an object, allowing clients to parameterize and decouple the sender of a request from its receiver. It provides a way to encapsulate actions or operations as objects, enabling the implementation of features like undo/redo, logging, or queuing requests. The Command pattern promotes loose coupling and flexibility in command execution.



1. State Pattern: The State pattern allows an object to alter its behavior when its internal state changes. It encapsulates different behaviors as separate state objects, promoting modularity and extensibility. The State pattern is useful when an object's behavior varies based on its internal state, and the transitions between states need to be managed.



1. Chain of Responsibility Pattern: The Chain of Responsibility pattern establishes a chain of objects, where each object in the chain has the ability to handle a request or pass it to the next object in the chain. It decouples the sender of a request from its receiver and allows multiple objects to have a chance to handle the request. The Chain of Responsibility pattern promotes flexibility and scalability by providing a dynamic way to handle requests.



These behavioral patterns provide solutions for designing and implementing the communication and interaction between objects, enabling flexible and maintainable software systems. By applying these patterns, developers can effectively manage object behaviors, separate concerns, and promote code reuse and extensibility. Each pattern offers a specific solution for a particular set of problems, and understanding their principles and application can greatly enhance the design and implementation of software systems.

# Implemented examples

## Creational patterns

1. Singletone

In my project, I implemented the Singleton pattern to handle the configuration of the SMTP service and credentials for the email sender service. The Singleton pattern ensures that there is only one instance responsible for managing these configurations throughout my application.

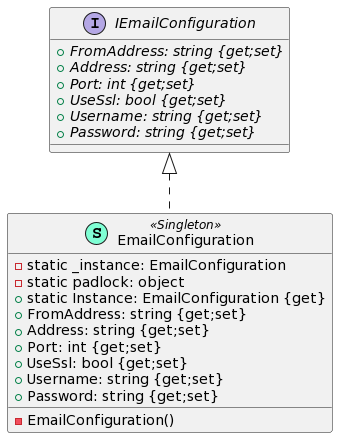
I chose to utilize the Singleton pattern because I wanted to maintain a centralized configuration for the SMTP service used for sending emails. By having a single instance responsible for managing these configurations, I could avoid duplication and inconsistencies that may arise from having multiple instances.

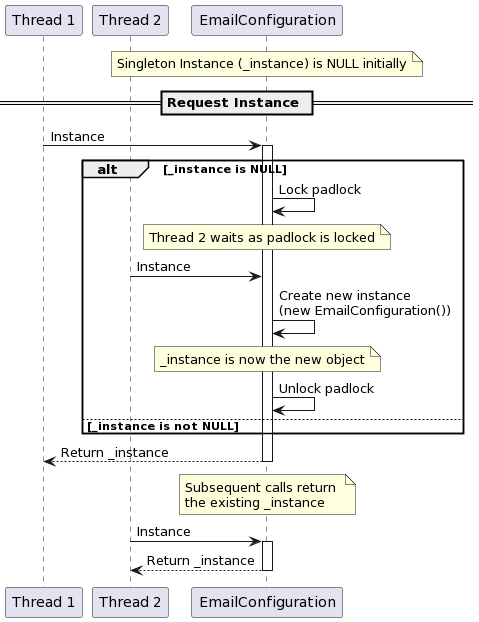
To implement the Singleton pattern, I used classic lock implimentation in C#. This means that the Singleton instance is created only when it is first requested, rather than eagerly creating it. This approach improves performance by deferring the creation of the Singleton instance until it is actually needed, saving unnecessary resource allocation.

Within the email sender service, the Singleton instance plays a crucial role in configuring the SMTP service and storing the necessary credentials. It holds information such as the SMTP server address, port number, authentication details, and sender email credentials. These configurations are typically retrieved from the appconfig or configuration files.

By encapsulating the SMTP configuration and sender credentials within the Singleton instance, I ensured that all parts of my code that need to send emails can access the same configurations without the need for redundant setup or multiple instances. This promotes consistency and simplifies the maintenance of the email sender functionality throughout my application.

Since the Singleton instance is globally accessible, any part of my code that requires access to the email sender service can obtain it through the Singleton instance. This eliminates the need for multiple instances and guarantees that all email-sending operations within my application consistently use the same SMTP configuration and sender credentials.





1. Prototype and builder

In my project, I have incorporated the Prototype design pattern to facilitate the cloning of topic entities. This pattern allows me to create copies or clones of existing topic instances without relying on their specific classes.

The Prototype pattern is particularly useful when I want to create new instances with similar initial state as an existing object, while still having the flexibility to modify and customize the copied object as needed. In the context of my project, I utilize the Prototype pattern on the edit page, specifically when a user clicks the "Submit as a copy" action.

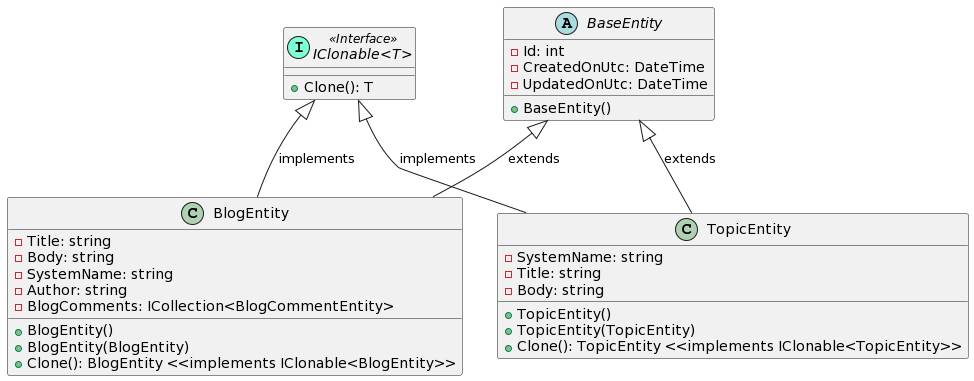
When the user initiates the "Submit as a copy" action, I create a new prototype of the source topic using the Builder pattern. The Builder pattern allows for the step-by-step construction of complex objects, enabling me to specify different attributes and behaviors for the new topic instance.

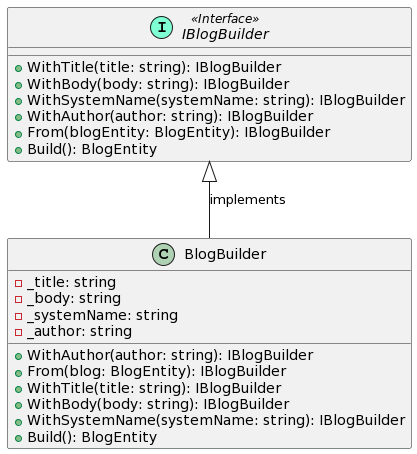
By leveraging the Prototype pattern in this scenario, I am able to create a new topic instance that starts with a similar state to the original topic. This means that the copied topic inherits attributes and properties from the source topic, such as its content, title, and other relevant details.

Additionally, during the cloning process, I assign a new Author to the copied topic. This Author represents the user who initiated the submission as a copy. This customization of the cloned object allows me to maintain a clear distinction between the original topic and the copied version, while still preserving the overall structure and properties.

The Prototype pattern provides significant advantages in terms of efficiency and flexibility. Instead of creating a new topic from scratch, I can leverage an existing instance as a starting point, avoiding the need to duplicate complex initialization processes. This approach not only saves resources but also ensures consistency among related topic instances.

In summary, the Prototype pattern in my project allows for the creation of topic clones, starting from an existing topic and customizing attributes such as the Author. By utilizing the Builder pattern, I can construct the cloned topic step by step, resulting in efficient and flexible object creation. This design pattern enhances the overall functionality and user experience of my project.





## Structural and Behavioral patterns

* 1. Strategy and Adapter:

As I implemented the Strategy and Adapter design patterns in my project, I aimed to improve the structural and behavioral aspects of my code. These patterns provided me with the necessary flexibility and adaptability to handle different strategies for working with the Topic entity and converting it into various formats.

For the Strategy pattern, I created adapters specifically tailored for the Topic entity. These adapters, namely Topic -> Blog and Topic -> string, represented different strategies for handling and transforming Topic objects. Each adapter encapsulated the logic required to convert a Topic into the desired output format.

By employing the Strategy pattern, I achieved separation of concerns and modularity in my code. I could easily add new adapters or modify existing ones without impacting the core logic of my application. This flexibility allowed me to support a wide range of conversions and output formats for the Topic entity.

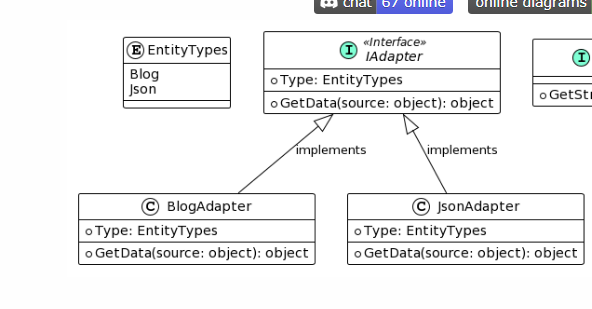
To resolve the appropriate adapter for a given conversion, I implemented an adapter resolver. This resolver analyzed the type of conversion needed and selected the corresponding adapter accordingly. This ensured that the correct adapter was used based on the desired output format, promoting efficient and accurate conversion processes.

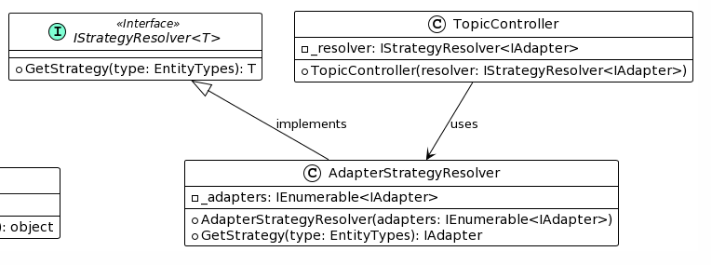
The Topic -> Blog adapter served the purpose of exporting a Topic as a new Blog entity. Within this adapter, I transformed the structure and content of the Topic to align it with the expected format of the Blog entity. This seamless integration between the Topic and Blog objects allowed for smooth interaction and compatibility.

On the other hand, the Topic -> string adapter provided a way to extract the content of a Topic and convert it into a string representation. This adapter was particularly useful when I needed to send a copy of the Topic's content via email or perform text-based operations that required a string format.

By implementing the Adapter pattern, I ensured the interoperability of different classes and achieved the conversion of Topic objects into alternative representations. This was accomplished without directly modifying the Topic class itself, maintaining its integrity and separation of concerns.

In summary, my implementation of the Strategy and Adapter patterns in my project offered enhanced flexibility, modularity, and maintainability. These patterns allowed me to dynamically select different strategies for handling Topic objects and adapt their interfaces to match the desired output formats. As a result, my code became more versatile and capable of seamlessly working with various formats and conversions.





* 1. Proxy

In my project, I have implemented the Proxy design pattern in multiple instances to enhance the security and access control of certain components. One of the proxies serves as an intermediate layer between the application's service layer and the repository level, while the other is implemented using ASP.NET Identity Privacy Policies for resource authorization.

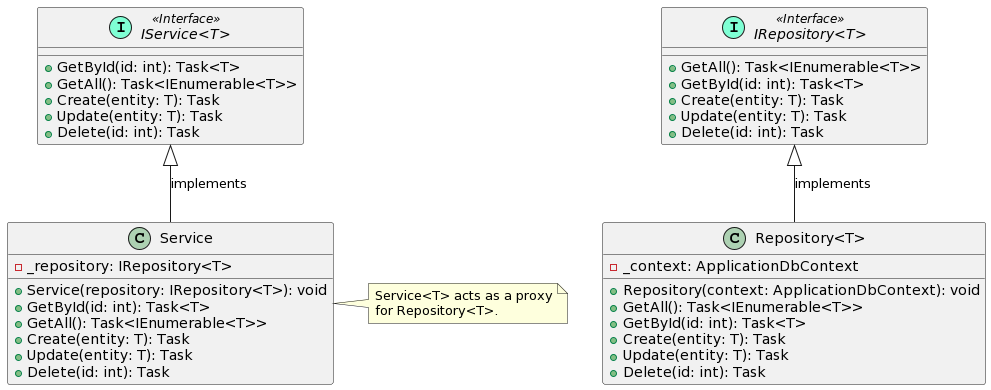
The first proxy acts as a protective layer between the service layer and the repository level. It functions similarly to a proxy by intercepting calls made to the repository and validating the input parameters before allowing access to the underlying data. Additionally, it transforms the data returned from the repository to prevent exposing IQueryable data directly to the consuming code.

By utilizing this proxy, I ensure that all incoming requests to the repository are validated before proceeding. This validation process can include checking the integrity of the input data, enforcing business rules, and applying any necessary security measures. The proxy acts as a safeguard, ensuring that only valid and authorized requests are allowed to access the repository, enhancing the overall security and integrity of the application's data.

The second proxy implementation utilizes the ASP.NET Identity Privacy Policies to control resource authorization. This proxy intercepts requests made to specific resources and checks the user's authorization before allowing access. By leveraging the privacy policies provided by ASP.NET Identity, I can easily define and enforce access restrictions based on user roles, claims, or any other criteria defined in the policies.

This proxy serves as an additional layer of security, ensuring that only authorized users can access specific resources within the application. It provides a fine-grained control mechanism, allowing me to restrict access to sensitive data or operations based on the user's permissions. By integrating this proxy into the application's authentication and authorization framework, I can enforce access control policies effectively.

In summary, the Proxy pattern in my project is employed in multiple instances to enhance security and access control. One proxy acts as an intermediate layer between the service and repository, validating input and transforming data, while the other proxy utilizes ASP.NET Identity Privacy Policies to enforce resource authorization. These proxies contribute to a robust and secure application architecture by adding an additional layer of validation and access control at critical points within the system.



* 1. Visitor

In my project, I have employed the Visitor design pattern to implement a mechanism for notifying top-level comments with emails about replies in the comment section. To achieve this, I extended the relevant classes using class extension.

The Visitor pattern allows for separating the algorithm or behavior from the elements it operates on. It enables adding new operations or behaviors to existing classes without modifying their structure or code directly. In this case, I extended the classes involved in the comment system to support the email notification functionality.

To implement the Visitor pattern, I created an extension class that acts as the visitor. This class encapsulates the logic for sending email notifications to top-level comments when replies are made. It defines a visit method specifically for the comment classes, which is responsible for performing the email notification.

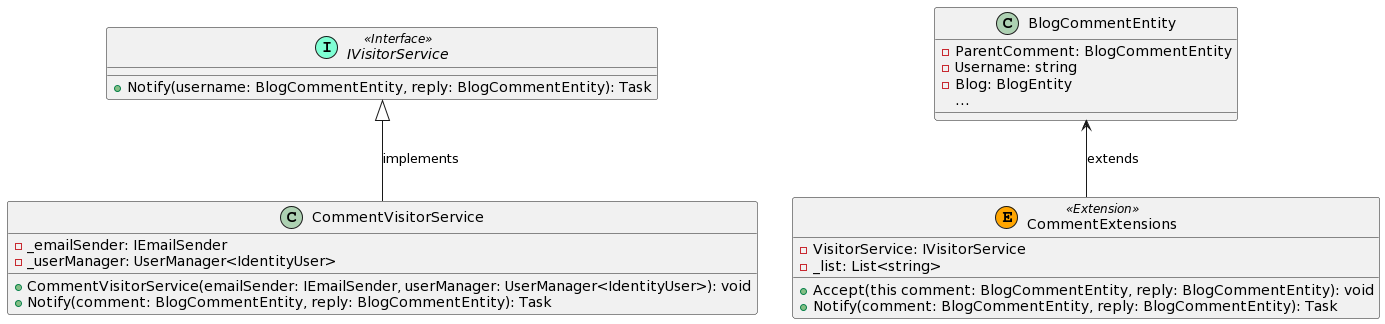
Within the visit method, the visitor can access the necessary information from the comment, such as the comment author, content, and the parent comment if applicable. Based on this information, the visitor can generate and send an email notification to the author of the top-level comment, informing them about the new reply.

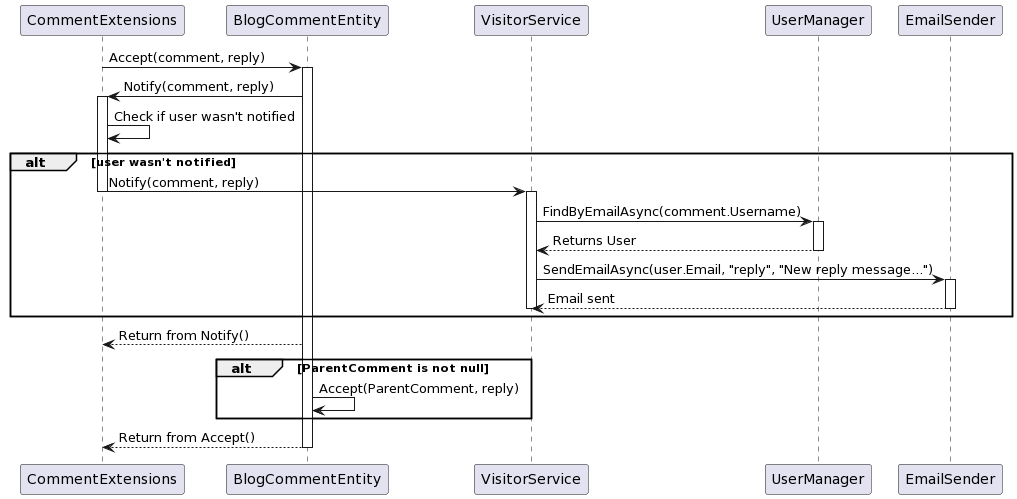
By extending the comment classes with the visitor's visit method, I enabled the comments to accept and process the visitor. This integration allows the comments to notify the visitor about their presence, triggering the email notification process.

Additionally, I implemented the necessary logic in the comment system to invoke the visitor's visit method whenever a reply is added to a comment. This ensures that the visitor is notified and performs the email notification for the relevant top-level comment.

By utilizing the Visitor pattern and extending the classes, I have achieved a modular and non-intrusive way to add email notification functionality to the comment system. The separation of concerns and the ability to extend classes with new behaviors allow for easier maintenance and scalability of the system.

In summary, by implementing the Visitor pattern through class extension, I have successfully integrated email notification functionality into the comment system. The visitor class encapsulates the logic for sending notifications, while the extended comment classes invoke the visitor's visit method to trigger the notification process. This approach promotes modularity, maintainability, and scalability in handling comment replies and email notifications.





# Conclusion

In conclusion, I could say that design patterns are essential building blocks for software development projects, providing well-established solutions to common design challenges. By incorporating design patterns, projects can achieve various benefits that contribute to their overall success.

One significant advantage of design patterns is the enhancement of maintainability. By following proven patterns, developers create code that is modular, loosely coupled, and easy to understand. This promotes maintainability by allowing for easy modification, debugging, and extension of the codebase. With clear and organized code, future updates and maintenance become more manageable and less error-prone.

Scalability is another area where design patterns excel. By employing patterns that support flexible and extensible architectures, projects can adapt and grow as requirements change. Patterns such as Factory Method, Builder, or Composite enable the addition of new features or components without affecting existing functionality. This scalability ensures that the project can evolve and meet future needs efficiently.

Reusability is a key benefit of design patterns. By encapsulating common solutions to recurring problems, patterns enable developers to extract and reuse well-defined components. This reduces code duplication and effort, leading to increased productivity and consistency across the project. Patterns also promote code consistency by establishing a shared vocabulary and best practices within the development team.

Design patterns facilitate collaboration and communication among team members. By using patterns, developers have a common language and framework to discuss and understand the design decisions and structures. This shared understanding improves coordination, fosters effective teamwork, and allows for more efficient collaboration during the development process.

Performance and optimization are areas where design patterns can make a significant impact. Certain patterns, such as Flyweight, Singleton, or Observer, provide efficient algorithms and data structures that optimize resource usage and improve system performance. By leveraging these patterns, projects can achieve better runtime efficiency, reduced memory consumption, and improved overall system performance.

Lastly, design patterns contribute to the overall quality and robustness of a project. They are proven solutions that have undergone extensive testing and refinement, reducing the risk of common pitfalls and design flaws. By following established patterns, projects can benefit from reliable and stable solutions, enhancing the reliability and quality of the final product.

In summary, design patterns are crucial tools in software development projects, offering numerous advantages in maintainability, scalability, reusability, collaboration, performance, and overall quality. By embracing and applying appropriate design patterns, projects can establish a solid foundation for success, ensuring efficient development, easier maintenance, and a high-quality end product.