Ministerul Educației al Republicii Moldova

Universitatea Tehnica a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

DEPARTAMENTUL INGINERIA SOFTWARE și AUTOMATICĂ

Raport

**Proiect de an**

**Disciplina:** TMPS

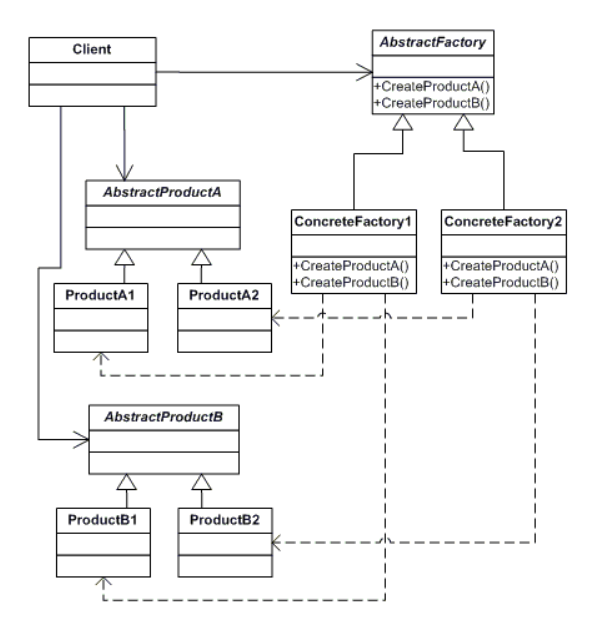
**A efectuat:** Bogdan Bobeico, TI-202

**A verificat:** asistent universitar, Gaidău Mihai

Chişinău 2023

**Creational Design Patterns**

**Abstract Factory**



The Abstract Factory design pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It allows the creation of objects that follow a common interface, but with varying implementations. This pattern promotes loose coupling and enhances flexibility, as it enables the system to be easily extended by adding new product variants

Intent:

The intent of the Abstract Factory pattern is to provide an interface for creating families of related objects, without explicitly specifying their classes. It abstracts the process of object creation and provides a way to decouple client code from the implementation details of the objects being created

Structure:

The key participants in the Abstract Factory pattern are:

Abstract Factory: This is the interface that declares the creation methods for the abstract products. It provides a common interface for creating different types of products.

Concrete Factories: These are the implementations of the Abstract Factory interface. Each concrete factory is responsible for creating a specific family of products. For example, a GUIFactory could have concrete factories like WinFactory and MacFactory, which create Windows and Mac-specific UI components, respectively.

Abstract Products: These are the abstract classes or interfaces that define the common methods to be implemented by the product families.

Concrete Products: These are the implementations of the Abstract Products. Each concrete product belongs to a specific family of products and implements the methods declared by the abstract product.

Interaction:

The client code interacts with the Abstract Factory and Abstract Products through their respective interfaces. The client is decoupled from the concrete classes and relies on the factory to create the appropriate products. The factory creates the products of a particular family and returns them to the client, which can then use the products without knowing their concrete types.

Benefits:

Encourages loose coupling: The Abstract Factory pattern enables the client code to work with abstract interfaces rather than concrete classes, promoting loose coupling between the client and the products being created.

Supports product variation: The pattern allows the system to be easily extended by adding new product families, as each family is encapsulated within a concrete factory. This makes it simple to introduce new variants of products without modifying existing client code.

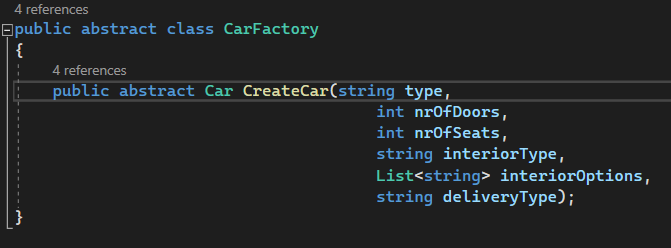
Provides a consistent interface: By using the Abstract Factory pattern, the client code can rely on a consistent interface to create and work with products, regardless of their concrete types. This simplifies the code and enhances maintainability.

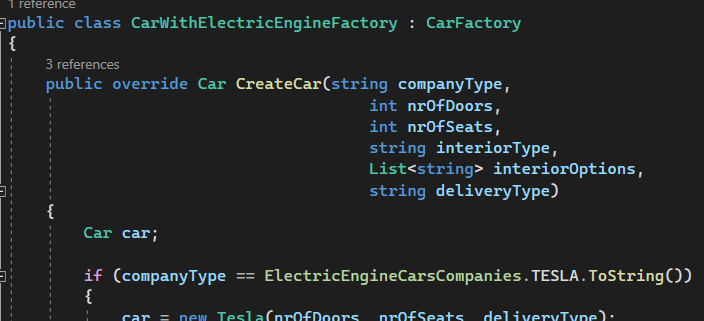
Limitations:

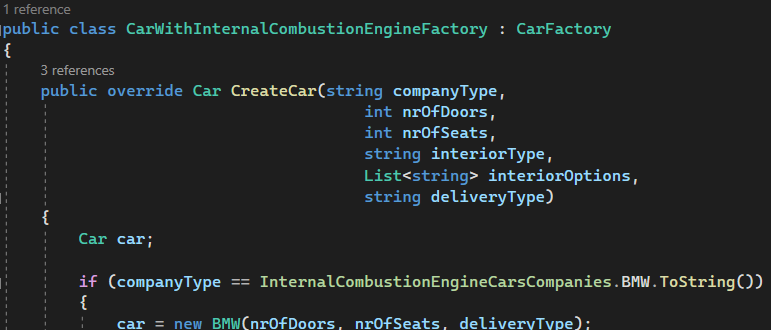
Complex to introduce new products: Adding a new product variant requires extending the abstract factory interface and implementing the new variant in all the concrete factories. This can make the codebase complex and increase the number of classes.

Difficult to extend with new factories: Introducing a new factory requires modifying the client code to work with the new factory interface. This can lead to code changes in multiple places, making it more difficult to extend the system.

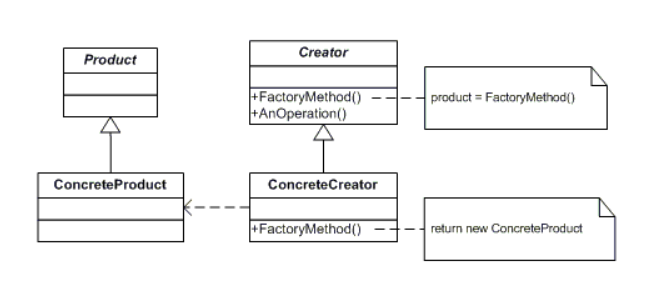
Implemented example:







**Factory Method**



The Factory Method design pattern is a creational design pattern that provides an interface for creating objects, but lets subclasses decide which class to instantiate. It encapsulates the object creation logic into separate methods, allowing subclasses to define the exact type of objects they want to create. This pattern promotes loose coupling and enhances extensibility by allowing the addition of new product classes without modifying the existing client code.

Intent:

The intent of the Factory Method pattern is to define an interface for creating objects, but defer the actual instantiation to subclasses. It provides a way to encapsulate object creation and allow subclasses to provide the specific implementation details.

Structure:

The key participants in the Factory Method pattern are:

Creator: This is the abstract class or interface that declares the factory method. It provides a common interface for creating objects but does not specify the concrete class.

Concrete Creators: These are the subclasses of the Creator that implement the factory method. Each concrete creator class decides which class to instantiate by overriding the factory method.

Product: This is the abstract class or interface that defines the common methods to be implemented by the products created by the factory method.

Concrete Products: These are the implementations of the Product interface. Each concrete product corresponds to a specific type of object created by the factory method.

Interaction:

The client code interacts with the Creator through its interface. The client requests an object creation by calling the factory method on the Creator. The Creator, in turn, invokes the factory method, which is overridden by the Concrete Creators. The Concrete Creator determines the specific class to instantiate and returns an object of that class, which is then used by the client.

Benefits:

Encourages loose coupling: The Factory Method pattern promotes loose coupling between the client code and the products being created. The client code only depends on the Creator interface and does not need to know the concrete classes of the products.

Supports extensibility: Adding new product classes is straightforward in the Factory Method pattern. By creating a new Concrete Creator subclass and implementing the factory method, new types of products can be introduced without modifying the existing code.

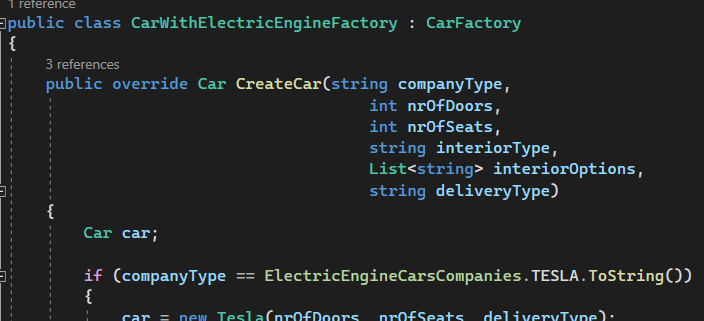
Allows customization: Subclasses of the Creator can provide different implementations of the factory method, allowing customization of the object creation process. This enables variations in the created objects while adhering to a common interface.

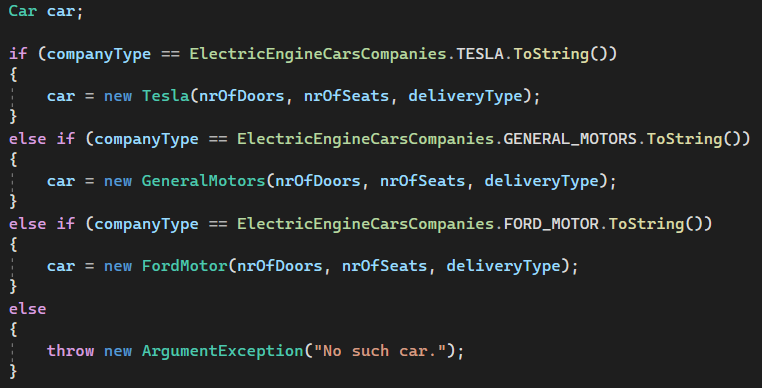
Limitations:

Can lead to class proliferation: As new product classes are added, the number of Concrete Creator subclasses can grow, potentially leading to a large number of classes in the system. This can make the codebase more complex and harder to maintain.

Can be challenging to understand: The Factory Method pattern introduces an additional level of indirection, which can make the code more difficult to understand, especially for developers who are unfamiliar with the pattern.

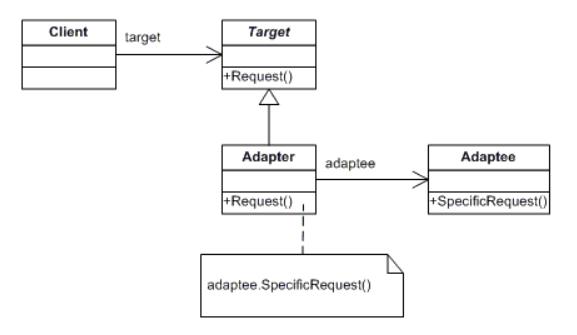
Implemented example:





**Structural Design Patterns**

**Adapter**



The Adapter design pattern is a structural design pattern that allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, converting the interface of one class into another interface that clients expect. This pattern promotes code reuse and enhances interoperability between classes with different interfaces.

Intent:

The intent of the Adapter pattern is to enable classes with incompatible interfaces to collaborate by providing a common interface that both classes can use. It allows objects with different interfaces to work together without modifying their source code.

Structure:

The key participants in the Adapter pattern are:

Target: This is the interface that the client code expects to work with. It defines the methods that the client code can use to interact with the Adapter.

Adapter: This is the class that implements the Target interface and acts as a bridge between the client code and the Adaptee. It wraps the Adaptee and translates its interface into the Target interface.

Adaptee: This is the class that has the incompatible interface that needs to be adapted. The Adapter communicates with the Adaptee and translates its interface to match the Target interface.

Interaction:

The client code interacts with the Adapter through the Target interface. When the client code calls a method on the Target interface, the Adapter translates the request and forwards it to the Adaptee, using the Adaptee's specific interface. The Adaptee processes the request and returns the result, which is then passed back to the client through the Adapter.

Benefits:

Enhances code reuse: The Adapter pattern allows the reuse of existing classes that have incompatible interfaces. By adapting their interfaces to a common Target interface, these classes can be easily integrated into the client codebase.

Promotes interoperability: The Adapter pattern promotes interoperability between classes with different interfaces. It enables objects from different sources or third-party libraries to work together seamlessly, even if they have incompatible interfaces.

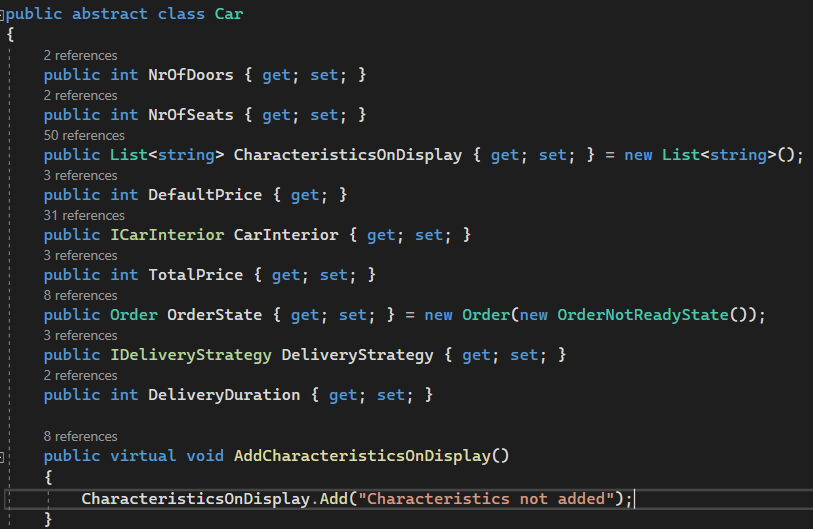
Provides flexibility: The Adapter pattern provides flexibility by allowing the client code to work with the Target interface, which is independent of the specific implementation details of the Adaptee. This makes it easier to switch or replace Adaptee implementations without affecting the client code.

Limitations:

May increase complexity: The Adapter pattern introduces an additional layer of indirection, which can make the code more complex and harder to understand. It is important to carefully design and document the adapters to ensure clarity and maintainability.

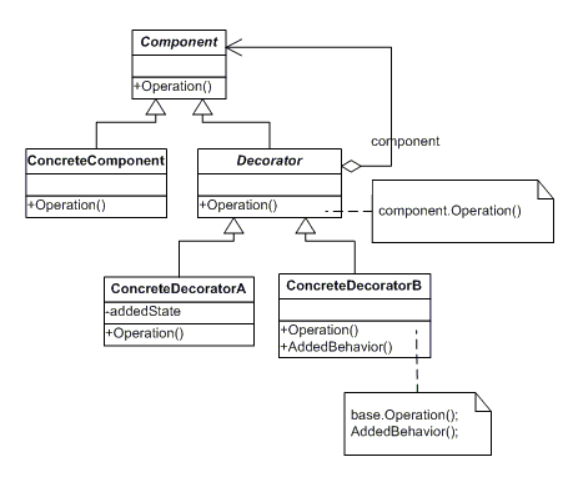
Performance overhead: The Adapter pattern may introduce some performance overhead due to the need for interface translation and additional method calls. However, in most cases, the impact on performance is negligible unless the adapters are heavily used in performance-critical scenarios.

Implemented example:





**Decorator**



The Decorator design pattern is a structural design pattern that allows behavior to be added to an object dynamically. It provides a flexible alternative to subclassing for extending the functionality of an object. The pattern involves wrapping the original object with one or more decorator objects, which modify the behavior of the original object without changing its interface. This pattern promotes code reusability, flexibility, and easy customization of objects at runtime.

Intent:

The intent of the Decorator pattern is to enhance the functionality of an object dynamically by adding new behaviors to it at runtime. It provides a way to extend the functionality of individual objects without the need to create numerous subclasses.

Structure:

The key participants in the Decorator pattern are:

Component: This is the abstract class or interface that defines the common interface for both the concrete component and its decorators. It declares the methods that represent the core functionality of the object.

Concrete Component: This is the class that provides the basic functionality and defines the behavior that can be decorated.

Decorator: This is the abstract class or interface that implements the Component interface and maintains a reference to the Component object. It acts as the base class for all the decorators and provides the common functionality shared among them.

Concrete Decorators: These are the classes that extend the Decorator class and add specific behaviors or modify existing behaviors. Each concrete decorator adds its own functionality while delegating the core behavior to the decorated object.

Interaction:

The client code interacts with the Component through its interface. The client can wrap the Component object with one or more decorators, effectively adding new behaviors to the object. Each decorator receives requests from the client, performs its specific tasks before or after forwarding the request to the decorated object, and returns the results to the client.

Benefits:

Code reusability: The Decorator pattern promotes code reusability by allowing the reuse of both the original component and the decorators. Different combinations of decorators can be applied to the same component, providing a flexible way to add and remove functionalities.

Dynamic behavior extension: The Decorator pattern enables the dynamic addition of behaviors to objects at runtime. Decorators can be easily added or removed without impacting the original object or other decorators. This allows for greater flexibility in customizing the behavior of objects.

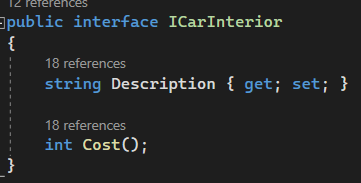
Open-closed principle: The Decorator pattern follows the Open-Closed Principle, as it allows new functionality to be added to an object without modifying its existing code. This reduces the need for extensive subclassing and minimizes the impact on existing code.

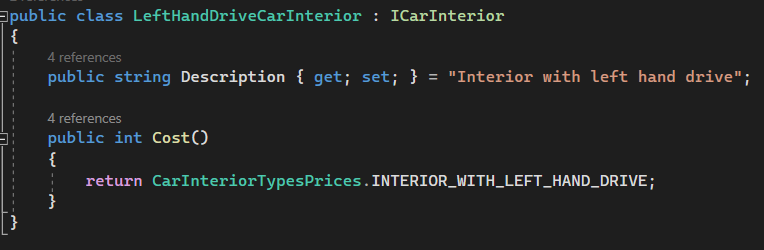
Limitations:

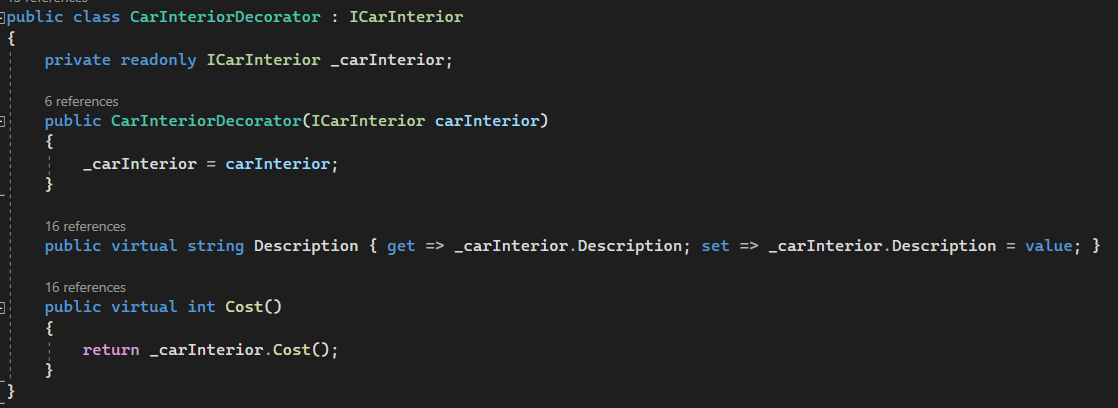
Increased complexity: The Decorator pattern can introduce a higher level of complexity, as it involves multiple layers of wrapping objects and delegation. This complexity can make the code more difficult to understand and maintain, especially if there are many decorators and complex interactions between them.

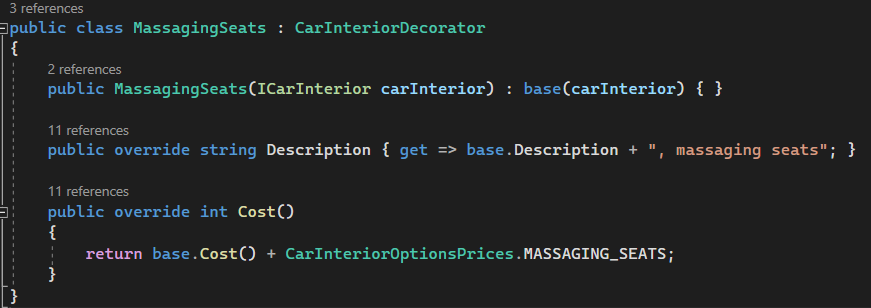
Order of decorators: The order in which decorators are applied can affect the final behavior of the object. Care must be taken when composing decorators to ensure the desired behavior is achieved. In some cases, the order of decoration may need to be predefined or controlled.

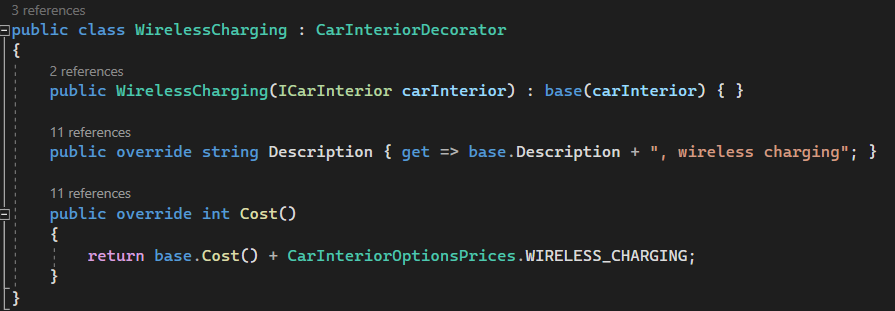
Implemented Example:





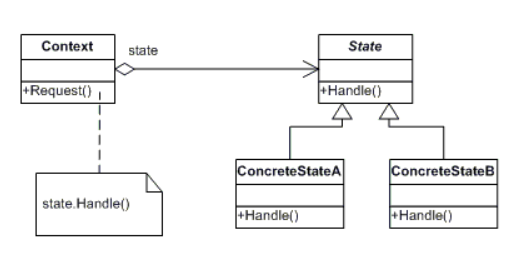






**Behavioral Design Patterns**

**State**



The State design pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. It encapsulates different behaviors into separate state objects and delegates the behavior execution to the current state object. This pattern promotes the flexibility to add new behaviors without modifying existing code and improves the clarity of state-dependent behavior.

Intent:

The intent of the State pattern is to allow an object to change its behavior dynamically based on its internal state. It enables the object to appear as if it changes its class when its internal state changes, without modifying the object's interface.

Structure:

The key participants in the State pattern are:

Context: This is the class that contains the state and defines the interface for interacting with the state objects. It maintains a reference to the current state object and delegates the behavior execution to the current state.

State: This is the interface or abstract class that defines the common methods for all concrete state classes. It encapsulates the behavior associated with a particular state of the Context.

Concrete States: These are the classes that implement the State interface. Each concrete state class provides its own implementation of the behavior associated with a specific state of the Context.

Interaction:

The client code interacts with the Context object, which delegates the behavior execution to the current State object. The Context holds a reference to the current state, and when the state changes, the Context updates its reference to the new state. The client can trigger state transitions or operations on the Context, and the Context forwards these requests to the current State for processing.

Benefits:

Simplifies state-dependent behavior: The State pattern simplifies complex state-dependent behavior by encapsulating each state into a separate class. It improves the clarity of code by separating the behavior associated with different states.

Open-closed principle: The State pattern follows the Open-Closed Principle, as it allows new states to be added without modifying the existing Context or State classes. This makes the pattern easy to extend with new behaviors.

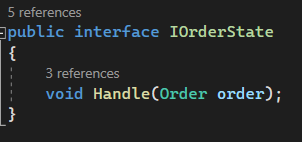
Improves maintainability: By encapsulating state-specific behavior into separate classes, the State pattern improves the maintainability of the codebase. It reduces the likelihood of introducing bugs and makes it easier to understand and modify the behavior of individual states.

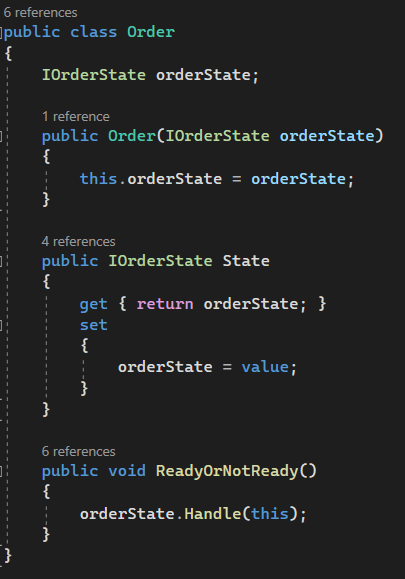
Limitations:

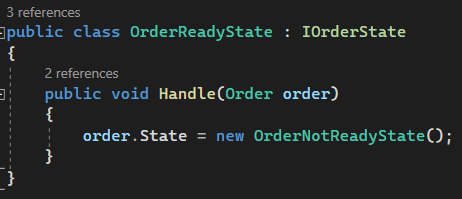
Increased number of classes: The State pattern can lead to a larger number of classes in the system, as each state is encapsulated into a separate class. This can make the codebase more complex and harder to manage, especially if there are many states with similar behavior.

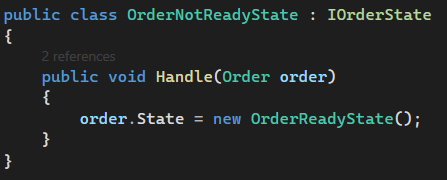
Context-awareness: The Context needs to maintain the current state and delegate behavior execution to the appropriate state object. This implies that the Context needs to be aware of the states and their transitions, which can introduce complexity and tight coupling.

Implemented example:

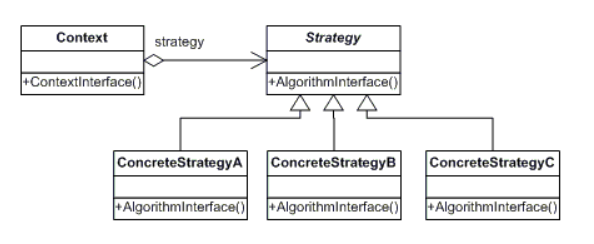








**Strategy**



The Strategy design pattern is a behavioral design pattern that enables interchangeable algorithms to be defined and encapsulated within separate strategy objects. It allows the selection of a specific algorithm at runtime, providing flexibility and promoting the principle of composition over inheritance. This pattern allows different algorithms to be easily interchanged without affecting the client code.

Intent:

The intent of the Strategy pattern is to define a family of algorithms, encapsulate each one as a separate strategy object, and make them interchangeable. It allows the client code to dynamically select and use different strategies based on the desired behavior.

Structure:

The key participants in the Strategy pattern are:

Context: This is the class that holds a reference to the selected strategy object. The context provides a method for the client code to set the desired strategy and delegates the execution of the algorithm to the strategy object.

Strategy: This is the interface or abstract class that defines the common methods for all concrete strategy classes. It encapsulates the algorithm and provides a common interface for executing the algorithm.

Concrete Strategies: These are the classes that implement the Strategy interface. Each concrete strategy class provides its own implementation of the algorithm.

Interaction:

The client code interacts with the Context object, which holds a reference to the current strategy object. The client can set or change the strategy of the Context at runtime by providing the desired concrete strategy object. When the algorithm needs to be executed, the Context delegates the execution to the current strategy object.

Benefits:

Flexibility and extensibility: The Strategy pattern provides flexibility by allowing different algorithms to be selected at runtime. It enables the addition of new strategies without modifying the existing code, promoting code extensibility.

Code reusability: The Strategy pattern promotes code reusability by encapsulating algorithms into separate strategy objects. Different strategies can be shared among multiple contexts, reducing code duplication.

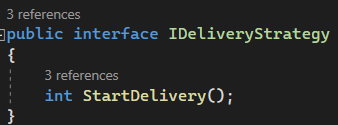
Improved maintainability: The Strategy pattern improves the maintainability of the codebase by encapsulating each algorithm in a separate strategy class. This makes it easier to understand, modify, and test individual strategies without impacting other parts of the code.

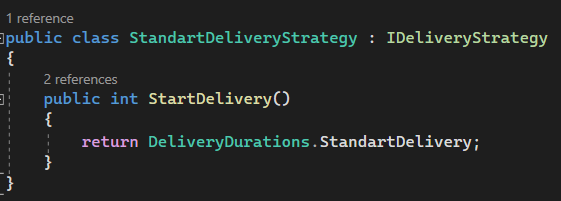
Limitations:

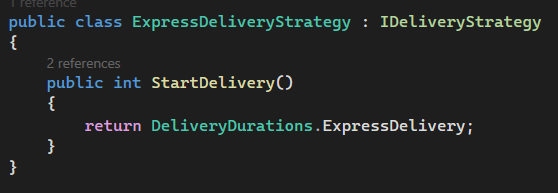
Increased complexity: The Strategy pattern introduces additional classes and indirection, which can make the codebase more complex. The increased number of classes and the need to manage the selection and configuration of strategies can add complexity to the system.

Overhead of strategy selection: The Strategy pattern incurs a small overhead in selecting and invoking the appropriate strategy object. This overhead may be negligible in most cases, but it is important to consider performance implications for highly time-critical systems.

Implemented example:

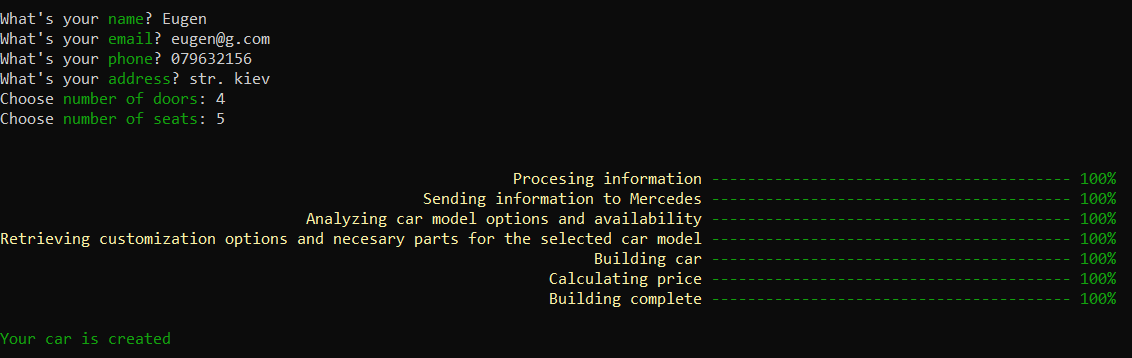


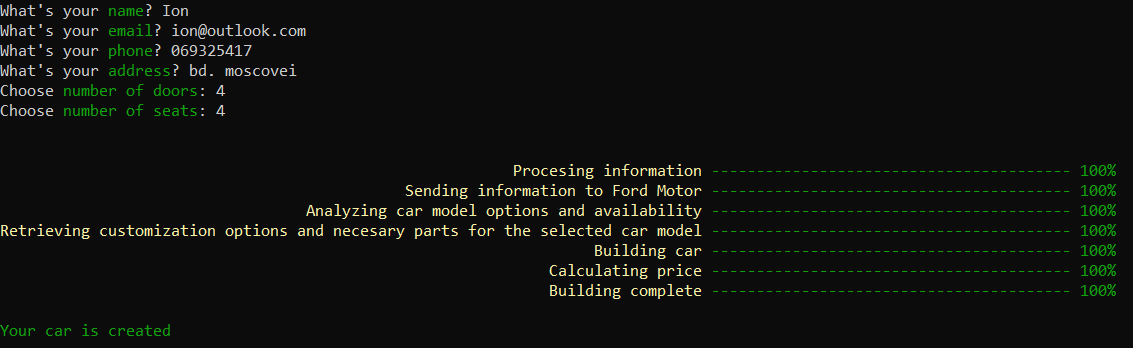


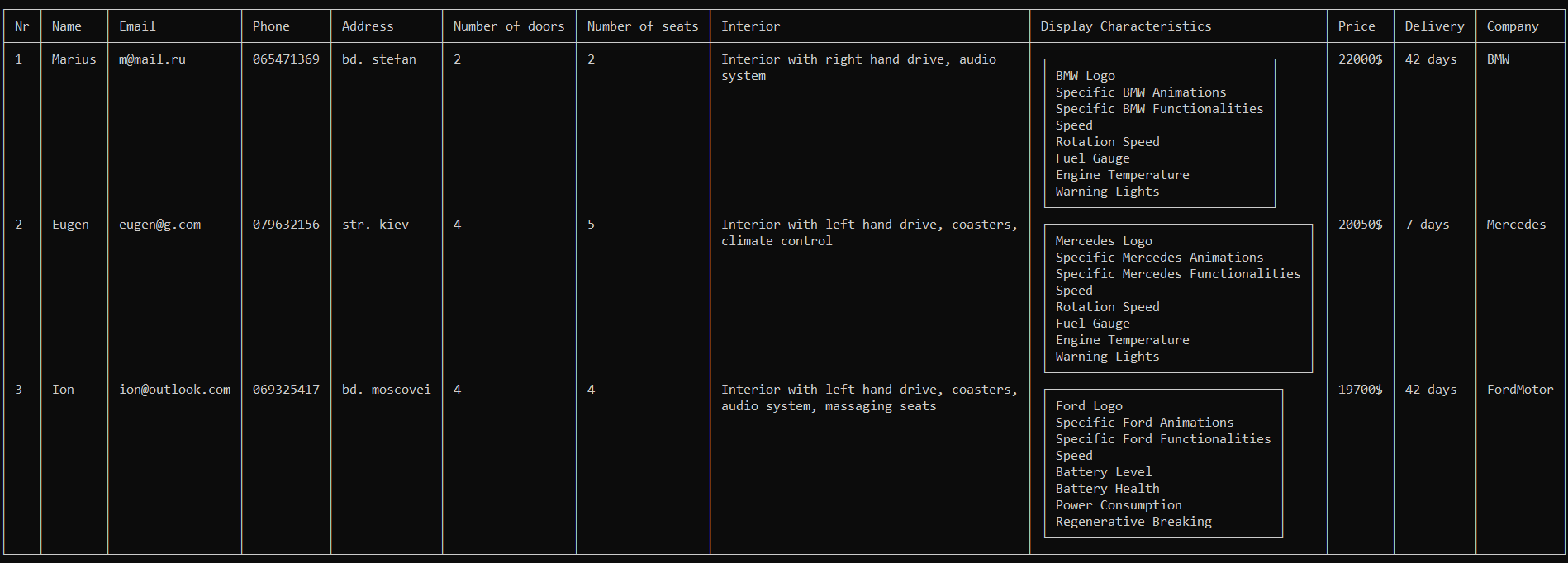


**Anexe**

****

****

****

****

**Conclusion**

In conclusion, if you were to implement a project incorporating the Abstract Factory, Factory Method, Adapter, Decorator, State, and Strategy design patterns, you would have a codebase that exhibits key characteristics such as flexibility, reusability, maintainability, and extensibility.

The Abstract Factory pattern would provide a way to create families of related objects without specifying their concrete classes, allowing for easy interchangeability and adherence to the open-closed principle. It would promote code scalability and facilitate the addition of new product variants.

The Factory Method pattern would enable the creation of objects without specifying their exact class, providing a flexible way to delegate object instantiation to subclasses. This pattern would enhance code extensibility and maintainability, allowing for the addition of new product creators without modifying existing code.

The Adapter pattern would bridge the gap between incompatible interfaces, allowing objects with different interfaces to work together seamlessly. It would promote code reusability, enhance interoperability, and provide flexibility in integrating classes from different sources.

The Decorator pattern would provide a means to dynamically add behavior to an object by wrapping it with decorator objects. It would enable code reusability, customization, and the open-closed principle, allowing for the addition of new behaviors without modifying existing code.

The State pattern would allow an object to change its behavior dynamically based on its internal state. It would simplify state-dependent behavior and improve maintainability by encapsulating different behaviors into separate state objects.

The Strategy pattern would offer the flexibility to interchange algorithms at runtime. It would provide code extensibility, reusability, and improved maintainability by encapsulating algorithms into separate strategy objects.

By incorporating all these design patterns into your project, you would have a codebase that demonstrates a modular and flexible architecture. It would be easier to maintain, extend, and customize the behavior of your objects without the need for extensive modifications. These patterns would promote code reusability, improve collaboration between components, and facilitate the addition of new features or variations in the future. Overall, the implementation of these design patterns would contribute to a robust and scalable software solution.