Design Patterns

Creational Patterns - Abstract Method

By: Alpesh Chaudhari

This document covers the Abstract Method, a key concept in Creational Design Patterns. This pattern, which solves the problem of creating entire product families without specifying their concrete classes.

Abstract Method

The **Abstract Factory** design pattern is a creational pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It is particularly useful when the system needs to be independent of how its objects are created, composed, and represented, and when families of related objects are designed to be used together.

When to Use the Abstract Factory Pattern

- **Families of Related Objects:** When you need to create families of related or dependent objects without specifying their concrete classes.
- **Consistency Among Products:** Ensures that products from the same family are used together.
- **Isolation of Concrete Classes:** Helps in isolating the client code from concrete classes, promoting loose coupling.

Components of the Abstract Factory Pattern

1. **Abstract Factory:** An interface declaring a set of methods for creating abstract products.
2. **Concrete Factory:** Implements the Abstract Factory interface to create concrete products.
3. **Abstract Product:** An interface for a type of product object.
4. **Concrete Product:** Implements the Abstract Product interface.
5. **Client:** Uses the Abstract Factory and Abstract Products without knowing their concrete implementations.

Problem

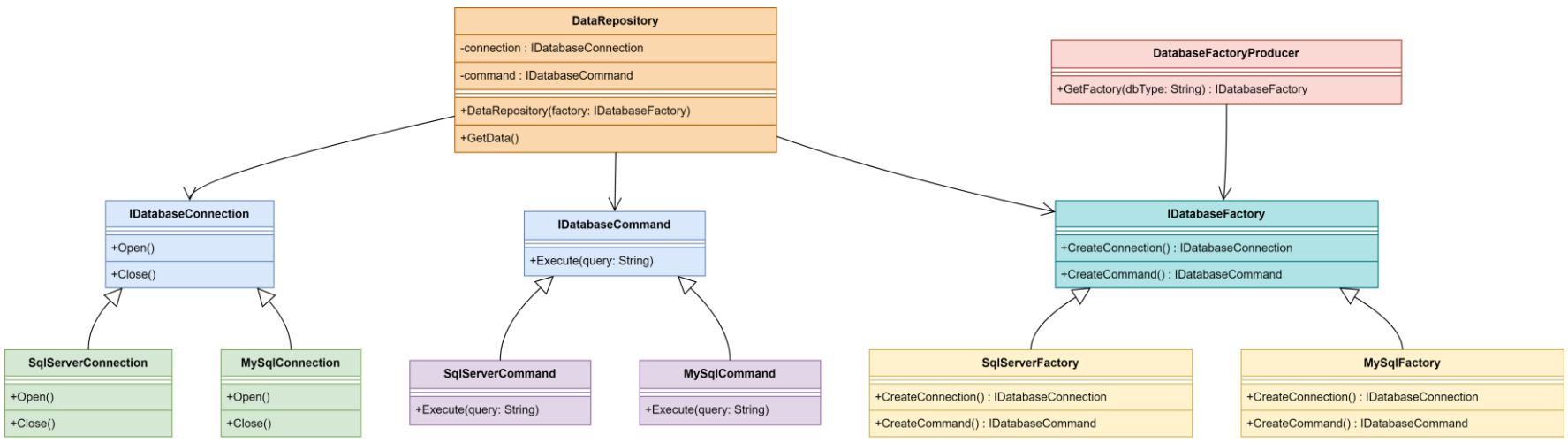
Imagine you're building a backend API that needs to support multiple database systems (e.g., SQL Server and MySQL). You want your application to be flexible enough to switch between different databases without changing the core business logic. The **Abstract Factory** pattern is ideal for this scenario as it allows you to create families of related objects (database connections and commands) without specifying their concrete classes.

Solution Using Factory Method Design Pattern

To address the challenges of adding new types of orders in your food delivery app while minimizing code changes and maintaining clean architecture, you can implement the **Factory Method** design pattern. This will help you create a flexible and extensible code structure.

Class Diagram

UML Class Diagram for the **SQL Server and MySQL** example using the **Abstract Factory** pattern will help visualize the relationships and structure of the classes and interfaces involved.



Here's how you can structure your code:

1. Define Abstract Products

These are interfaces for the database connection and command objects.

```
// Abstract Product - Database Connection
public interface IDatabaseConnection
{
    void Open();
    void Close();
}

// Abstract Product - Database Command
public interface IDatabaseCommand
{
    void Execute(string query);
}
```

2. Define Concrete Products

These are the concrete implementations of the abstract products for SQL Server and MySQL.

```
using System;
// Concrete Product - SQL Server Connection
public class SqlServerConnection : IDatabaseConnection
{
    public void Open()
    {
        Console.WriteLine("Opening SQL Server connection.");
    }

    public void Close()
    {
        Console.WriteLine("Closing SQL Server connection.");
    }
}
// Concrete Product - SQL Server Command
public class SqlServerCommand : IDatabaseCommand
{
    public void Execute(string query)
    {
        Console.WriteLine($"Executing SQL Server query: {query}");
    }
}
// Concrete Product - MySQL Connection
public class MySqlConnection : IDatabaseConnection
{
    public void Open()
    {
        Console.WriteLine("Opening MySQL connection.");
    }

    public void Close()
    {
        Console.WriteLine("Closing MySQL connection.");
    }
}
// Concrete Product - MySQL Command
public class MySqlCommand : IDatabaseCommand
{
    public void Execute(string query)
    {
        Console.WriteLine($"Executing MySQL query: {query}");
    }
}
```

3. Define the Abstract Factory

An interface for creating abstract products (connections and commands).

```
// Abstract Factory
public interface IDatabaseFactory
{
    IDatabaseConnection CreateConnection();
    IDatabaseCommand CreateCommand();
}
```

4. Define Concrete Factories

Implement the Abstract Factory to create concrete products for SQL Server and MySQL.

```
// Concrete Factory - SQL Server Factory
public class SqlServerFactory : IDatabaseFactory
{
    public IDatabaseConnection CreateConnection()
    {
        return new SqlServerConnection();
    }

    public IDatabaseCommand CreateCommand()
    {
        return new SqlServerCommand();
    }
}

// Concrete Factory - MySQL Factory
public class MySqlFactory : IDatabaseFactory
{
    public IDatabaseConnection CreateConnection()
    {
        return new MySqlConnection();
    }

    public IDatabaseCommand CreateCommand()
    {
        return new MySqlCommand();
    }
}
```

5. Client Code

The client (e.g., a repository or service) interacts with the Abstract Factory and Abstract Products without knowing their concrete implementations.

```
public class DataRepository
{
    private readonly IDatabaseConnection _connection;
    private readonly IDatabaseCommand _command;

    public DataRepository(IDatabaseFactory factory)
    {
        _connection = factory.CreateConnection();
        _command = factory.CreateCommand();
    }

    public void GetData()
    {
        _connection.Open();
        _command.Execute("SELECT * FROM Users");
        _connection.Close();
    }
}
```

6. Configuration or Environment Setup

Decide which factory to use based on configuration settings (e.g., appsettings.json, environment variables).

```
using System;

public static class DatabaseFactoryProducer
{
    public static IDatabaseFactory GetFactory(string dbType)
    {
        switch (dbType.ToLower())
        {
            case "sqlserver":
                return new SqlServerFactory();
            case "mysql":
                return new MySqlFactory();
            default:
                throw new NotSupportedException($"Database type '{dbType}' is not supported.");
        }
    }
}
```

7. Putting It All Together

Here's how you can use the Abstract Factory in your backend API application.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Example: Get database type from configuration
        Console.WriteLine("Enter database type (SqlServer/MySql): ");
        string dbType = Console.ReadLine();

        try
        {
            IDatabaseFactory factory = DatabaseFactoryProducer.GetFactory(dbType);
            DataRepository repository = new DataRepository(factory);
            repository.GetData();
        }
        catch (NotSupportedException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

8. Example Output

Case 1: Using SQL Server

```
Enter database type (SqlServer/MySql):
SqlServer
Opening SQL Server connection.
Executing SQL Server query: SELECT * FROM Users
Closing SQL Server connection.
```

Case 2: Using MySQL

```
Enter database type (SqlServer/MySql):
MySql
Opening MySQL connection.
Executing MySQL query: SELECT * FROM Users
Closing MySQL connection.
```

Advantages:

1. Ensures compatibility between products created by the factory.
2. Reduces tight coupling between concrete products and client code, enhancing flexibility.
3. Promotes the Single Responsibility Principle by centralizing product creation code, making it easier to maintain.
4. Adheres to the Open/Closed Principle, allowing new product variants to be introduced without affecting existing client code.

Disadvantages:

1. The code can become unnecessarily complex due to the introduction of multiple interfaces and classes as part of the pattern.

Conclusion

The Abstract Factory pattern is a powerful tool for creating families of related objects without specifying their concrete classes. In backend API development, it allows for flexible and maintainable code by decoupling the data access layer from specific database implementations. By following the Abstract Factory pattern, you can design a scalable and adaptable architecture that can evolve with changing requirements.

Note: For further examples of Abstract Method use cases in real-life scenarios within API development, please refer to my GitHub repository titled "DesignPattern." In this repository, I have included an additional example of the Abstract Method pattern, such as Payment Processing with multiple payment methods.

GitHub: https://github.com/iam-alpesh/DesignPatterns.git

Thank You

I would like to express my gratitude to the following individuals and resources that helped me in understanding and implementing the Abstract Method design pattern:

References

1. **Dive into Design patterns** by Alexander Shvets. Refactoring.Guru, 2021.
2. **Desing Patterns**. Refactoring Guru, https://refactoring.guru/