**Builder Pattern in Meal Ordering System**

**Detailed Problem Statement:**

**Background and Context**

In the restaurant industry, providing a flexible and customer-friendly meal ordering system is essential to cater to diverse preferences and dietary needs. Consider the scenario of "QuickBite Diner," a popular chain of restaurants offering a wide range of customizable meal options to accommodate various customer tastes, including fast food, healthy dining, and family-friendly choices. Customers can order meals with different components, such as a main course (e.g., burger, pizza, or grilled chicken), a side dish (e.g., fries, salad, or onion rings), a drink (e.g., soda, water, or juice), and an optional dessert (e.g., ice cream, cake, or none). The system must allow customers to build their meals step-by-step through a user interface, such as a web portal, mobile app, or in-store kiosk, and support predefined meal combinations like a "Value Meal" or "Healthy Meal" for convenience.

Directly constructing a meal object using constructors poses several challenges:

* **Complexity**: A Meal class with constructors for every possible combination of components (e.g., main only; main and side; main, side, and drink; etc.) leads to a combinatorial explosion of constructor overloads, known as the telescoping constructor problem.
* **Inflexibility**: Adding new components (e.g., a sauce or appetizer) or modifying existing ones requires significant changes to the Meal class and its constructors.
* **Validation**: Ensuring valid combinations (e.g., a meal must have a main course) becomes cumbersome without a structured process.
* **User Experience**: Customers need a clear, step-by-step process to customize meals, and predefined meal options should streamline common orders without sacrificing flexibility.

The Builder Pattern is ideal for this scenario, as it separates the construction of a complex object (the meal) from its representation, allowing step-by-step assembly and enabling both custom and predefined configurations. This pattern promotes a fluent interface for building meals, improves code readability, and simplifies the addition of new components or meal types in the future, aligning with the restaurant's goal of scalability and maintainability.

**Requirements**

1. **System Architecture**:
   * Define a Meal class (the product) to represent the final meal object, with properties for MainCourse, Side, Drink, and Dessert (all as strings, nullable for optional components like dessert).
   * Include a Display() method in the Meal class to print the meal's components, clearly indicating "None" for unset components.
   * Create an interface IMealBuilder that defines methods for setting each component: SetMainCourse(string main), SetSide(string side), SetDrink(string drink), and SetDessert(string dessert), each returning the builder for method chaining (fluent interface). Include a Build() method to return the final Meal object.
   * Implement a concrete MealBuilder class that constructs a Meal object incrementally, storing intermediate state in a private Meal instance.
2. **Director for Predefined Meals**:
   * Implement a MealDirector class to encapsulate the construction logic for predefined meal types:
     + **Value Meal**: Includes a burger (main), fries (side), and soda (drink), with no dessert.
     + **Healthy Meal**: Includes a pizza (main), salad (side), water (drink), and ice cream (dessert).
   * The director should have methods like ConstructValueMeal(IMealBuilder builder) and ConstructHealthyMeal(IMealBuilder builder) that use the builder to assemble the meal and return the final Meal object.
   * Ensure the director is optional, allowing clients to use the builder directly for custom meals.
3. **Builder Functionality**:
   * The MealBuilder should allow step-by-step construction, where each method sets the corresponding component and returns the builder for chaining (e.g., builder.SetMainCourse("Burger").SetSide("Fries")).
   * After calling Build(), the builder should return the constructed Meal object. To allow multiple meals to be built, the builder should either reset its internal state or create a new Meal instance for each build process.
   * Optionally, enforce constraints (e.g., a meal must have a main course) by throwing exceptions in Build() if critical components are missing, though for simplicity, this case study can assume all components are optional.
4. **Client Usage and Demonstration**:
   * In a main program (e.g., a console application), demonstrate the system by:
     + Creating a Value Meal using the MealDirector and displaying its components.
     + Creating a Healthy Meal using the MealDirector and displaying its components.
     + Optionally, demonstrate a custom meal using the MealBuilder directly (e.g., a meal with only a main course and drink).
   * The output should clearly show each meal's components, formatted consistently for readability.
   * Ensure the client code interacts only with the IMealBuilder interface and MealDirector, promoting loose coupling.
5. **Non-Functional Requirements**:
   * **Flexibility**: The design must allow easy addition of new components (e.g., a sauce or appetizer) by extending the IMealBuilder interface and Meal class without modifying existing client code or the director.
   * **Reusability**: The builder should support constructing multiple meals sequentially, either by resetting its state or creating new instances.
   * **Maintainability**: Use clear naming conventions (e.g., SetMainCourse instead of AddMain) and follow C# best practices, including proper encapsulation and minimal dependencies (use only System namespace for this case study).
   * **Scalability**: The system should handle additional meal types (e.g., "Kids Meal") by adding new methods to the MealDirector without altering existing logic.
   * **Error Handling**: While not mandatory, consider basic validation (e.g., non-null strings for components) to prevent invalid states. For simplicity, assume valid inputs for this case study.

**Expected Outcomes**

* The system should produce clear output for each meal, showing all components (e.g., "Main: Burger, Side: Fries, Drink: Soda, Dessert: None" for a Value Meal).
* The Builder Pattern ensures a clean, fluent API for meal construction, making it intuitive for developers to extend and for customers to customize orders via a UI (in a real application).
* This implementation mirrors real-world restaurant systems like those used by fast-food chains (e.g., McDonald's or Starbucks), where customizable orders are common, and predefined combos streamline operations.