

Daily Fitness Plan Builder



Builder Pattern Implementation

Yuhang Zhang
Fall 2025
December 9, 2025

Project Overview

CS-665 Software Design & Patterns



Goal of the Project

- Construct a composite object consisting of WorkoutPlan and MealPlan.
- Apply the Builder Pattern to separate how a plan is built from its representation.
- Demonstrate an extensible architecture where the Director (Coach) defines the flow, while Concrete Builders define the details.



Why Fitness as an Example?

- Naturally step-based domain: workout, meals, and scheduling map cleanly to staged construction.
- High variability: Bulking, Cutting, or Beginner styles are perfect examples of different Builders.
- Intuitive: Easy for readers to understand the difference between the Coach (Director) and the Plan (Product).



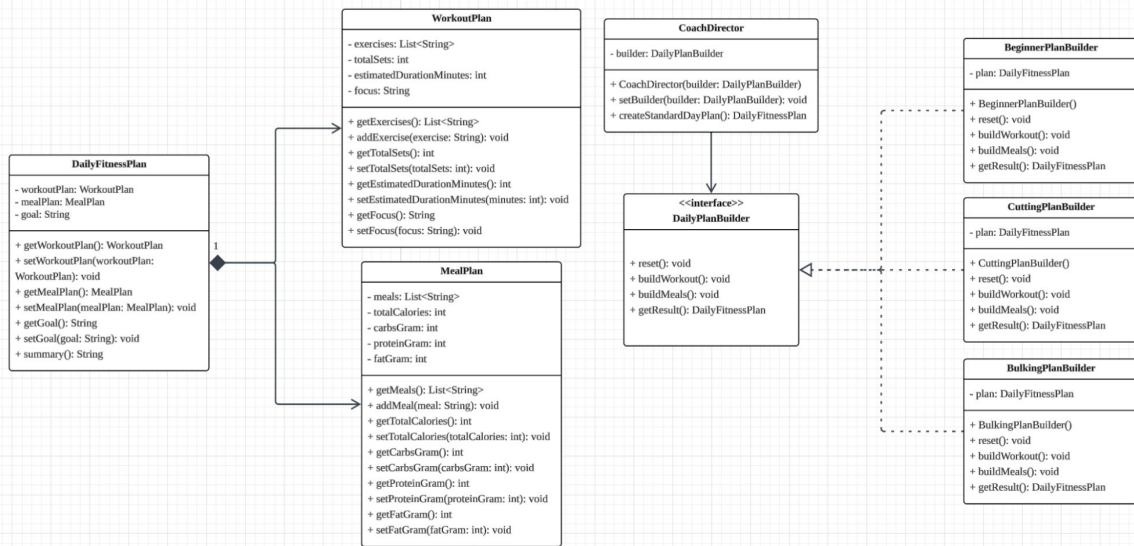
+



Daily Fitness Plan

System Architecture

UML Class Diagram & Component Interactions



DailyFitnessPlan Product

The final complex object that aggregates **WorkoutPlan** and **MealPlan**. It holds the data but doesn't contain creation logic.

DailyPlanBuilder Interface

Defines the standard construction steps:
`buildWorkout()`, `buildMeals()`, and `getResult()`.

Concrete Builders Impl

Bulking, Cutting, and Beginner builders implement the interface to provide specific data (calories, sets).

CoachDirector Director

Orchestrates the sequence. It calls the builder steps in a specific order (e.g., plan workout first, then meals) to ensure validity.

Builder Pattern Explanation

Pattern Roles & Application Rationale

Core Idea



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



Director

`CoachDirector`

Controls the algorithm. It knows *how* to build (the order of steps) but not *what* is being built.

`</> Calls: buildWorkout() → buildMeals()`



Builder Interface

`<<interface>>`

Specifies the contract. Declares all necessary steps to create the product parts.

Abstracts the construction process from specific implementations.



Concrete Builders

`CBulking / Cutting`

Implements the steps. Defines specific details (e.g., Heavy weights for Bulking vs. Cardio for Cutting).

Each builder produces a different variant of the product.



Product

`DailyFitnessPlan`

The complex object. The final result containing all assembled parts (Workout + Meals).

Passive data holder; no logic for how it was created.

Implementation Details

Key Classes & Concrete Builder Outputs

Class Definitions

DailyFitnessPlan

Product

```
private WorkoutPlan workout;  
  
private MealPlan meals;  
  
// Getters, setters, and toString()
```

WorkoutPlan

Component

```
List<String> exercises;  
  
int totalSets;  
  
String focus;
```

MealPlan

Component

```
List<String> meals;  
  
int totalCalories;  
  
Map<String, Int> macros;
```

Builder Output Examples

Bulking Plan

BulkingPlanBuilder

WORKOUT

- Bench Press (5x5)
- Squats (5x5)
- Focus: Strength

DIET

- Calories: ~3000 kcal
- High Protein, High Carb

Cutting Plan

CuttingPlanBuilder

WORKOUT

- HIIT Cardio (30m)
- Circuit Training (3x15)
- Focus: Fat Loss

DIET

- Calories: ~1800 kcal
- High Protein, Low Fat

Beginner Plan

BeginnerPlanBuilder

WORKOUT

- Full Body Basics
- Machine Press (3x10)
- Focus: Form

DIET

- Calories: ~2200 kcal
- Balanced Macros

Test Strategy

JUnit 5 Test Coverage & Validation

✓ Product Assembly

Verify that the `DailyFitnessPlan` is correctly instantiated and contains non-null `Workout` and `Meal` components.

✓ Director Invocation

Ensure `CoachDirector` calls all necessary builder steps (`buildWorkout`, `buildMeals`) in the correct sequence.

✓ Variant Behavior

Confirm that different Concrete Builders (e.g., `Bulking` vs. `Cutting`) produce objects with distinct properties.

✓ Internal Structure

Validate the integrity of list data (exercises) and numeric values (calories, sets) within the components.

```
[INFO] -----
[INFO]   T E S T S
[INFO] -----
[INFO] Running edu.bu.met.cs665.BulkingPlanTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.034 s -- in edu.bu.met.cs665.BulkingPlanTest
[INFO] Running edu.bu.met.cs665.DailyPlanBuilderTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s -- in edu.bu.met.cs665.DailyPlanBuilderTest
[INFO] Running edu.bu.met.cs665.CoachDirectorTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in edu.bu.met.cs665.CoachDirectorTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 1.845 s
[INFO] Finished at: 2025-12-07T17:57:17-05:00
[INFO] -----
```

Advantages & Conclusion

Project Retrospective



High Extensibility

Adding a new plan type only requires creating a new Builder class. The existing Director and client code remain unmodified.



Clear Separation

The Director manages the process, the Builder handles details, and the Product holds the result.



Less Duplication

Common construction steps are defined once in the interface, reducing redundancy across different plan variants.



High Maintainability

Adheres strictly to SOLID principles. Classes have single responsibilities and low coupling, making maintenance easier.

Conclusion

This project successfully demonstrates the power of the Builder Pattern in constructing complex objects like the Daily Fitness Plan.

We achieved a system with a stable construction process but flexible outputs, high extensibility, and low coupling—proving the pattern's value in scalable software design.

GitHub & Q&A

Resources & Discussion

Thank You!

Questions are welcome.

[Github](#) Link