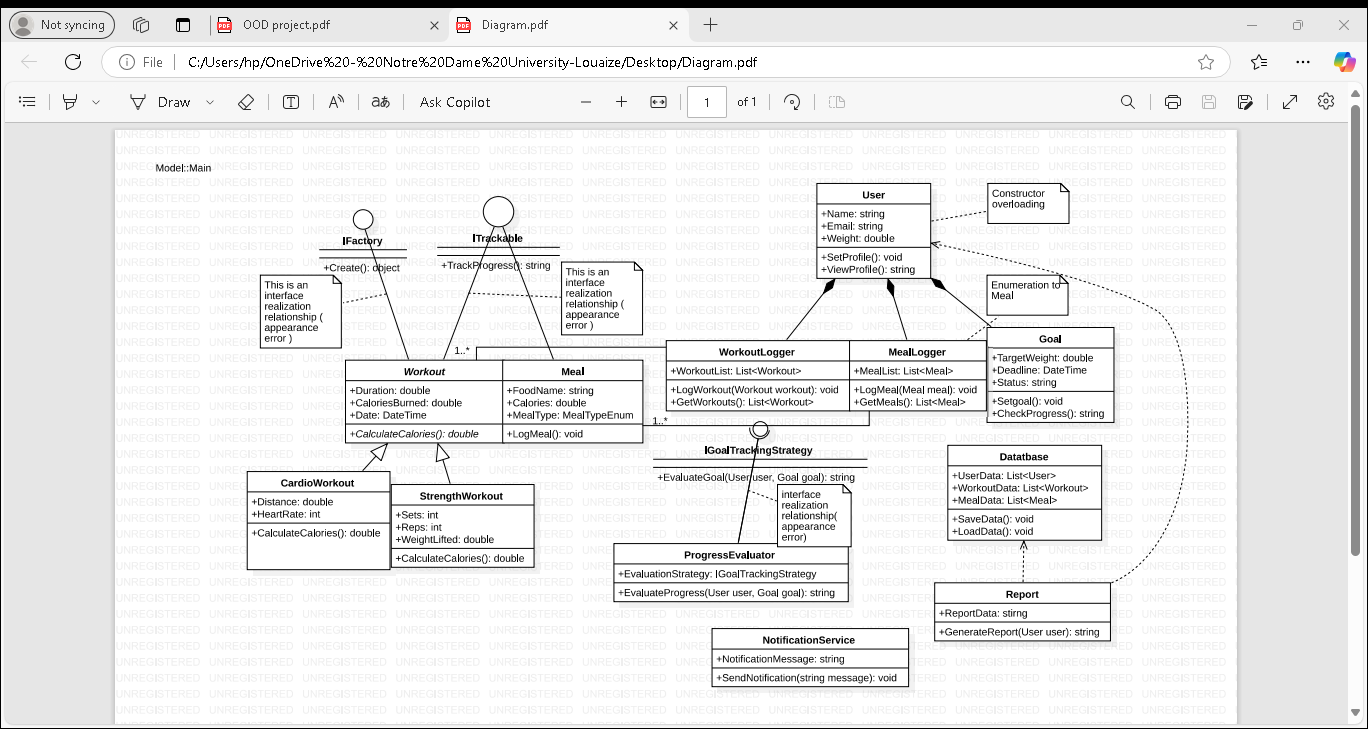
|  |  |
| --- | --- |
| cid:3461564935_486740 | **Faculty of Natural and Applied Sciences**  **Computer Science Department** |

|  |
| --- |
| **CSC 323: Object-Oriented Design**  **Project – Report** |
| Submitted by:   |  |  |  | | --- | --- | --- | | Student ID | Name | Major | | 20238106 | **Wissam Fares** | **Computer Science (CS)** | | 20238036 | **Ranim Hdaife** | **Computer Science (CS)** | | 20238031 | **Ribal Saleh** | **Computer Science (CS)** | |

**Project Description** (around 500 words)

|  |
| --- |
| My project is a software application about health and fitness tracker, an application tool designed to be a comprehensive tool for individuals who wish to track and manage their fitness journey. It allows users to record their workouts, monitor their meals, set personal goals, and track their progress in a clear, easy-to-understand format. The primary aim of the application is to help users achieve their health and fitness objectives by providing them with insights into their physical activities, calorie intake, and progress overtime. The application consists of 15 types of classes and interfaces, each representing a specific aspect of the fitness tracking process. User is a core class, representing an individual user of the app, with properties like name, email, and weight. Users can create profiles, set goals, and track their personal progress. Workouts are modeled as abstract classes, with CardioWorkout and StrengthWorkout as concrete subclasses that store specific details for cardio and strength exercises, such as duration, calories burned, and heart rate for cardio or weight lifted for strength training. The Meal class allows users to log onto meal details, including calories and nutritional values. The Goal class represents user-specific fitness objectives, such as weight loss or muscle gain, while the GoalTracker interface defines methods for tracking progress toward these goals.  Other classes include WorkoutLogger and MealLogger, responsible for managing the logging of workouts and meals. Report generates summary reports about user activities and progress, while ProgressEvaluator evaluates whether the user is meeting their goals based on their workout and meal logs. A NotificationService sends alerts to remind users to log their meals or workouts, and the Database class manages the storage of user data and activity logs.  The application uses several key object-oriented programming concepts, including abstraction, concrete classes, interfaces, methods, and properties. Abstraction is achieved through abstract classes like Workout, allowing for shared functionality across different types of workouts. Concrete classes like CardioWorkout and StrengthWorkout define specific behaviors for those workout types. Interfaces such as GoalTracker and MealTracker ensure that different types of goal tracking and meal logging can be implemented flexibly. Overriding and overloading are used to customize methods, such as the CalculateCalories method for different workout types, while constructor overloading allows for different ways of instantiating classes like User.  The system also utilizes enumerations (enums) to categorize workout types (e.g., Cardio, Strength), while collections like lists store and manage multiple workouts and meal records. The application also incorporates loops (such as foreach) to iterate over collections, especially when generating reports or evaluating progress.  To ensure the software adheres to industry’s best practices, the application implements the SOLID principles. For example, the Single Responsibility Principle is followed by having distinct classes for logging workouts, meals, and tracking goals. The Open/Closed Principle is applied by designing the system so that new features, such as additional workout types, can be added without modifying existing classes. The Liskov Substitution Principle is maintained by ensuring that subclasses of Workout can be substituted for the base class without affecting the system. The Interface Segregation Principle is followed by creating specialized interfaces for different functionalities (e.g., GoalTracker and MealTracker), and the Dependency Inversion Principle is used by depending on abstractions rather than concrete classes.  In addition to SOLID principles, design patterns such as the Factory Pattern are used to create workout or meal objects dynamically based on user input. The Builder Pattern is employed to construct complex User and Goal objects with multiple optional parameters, and the Strategy Pattern is used to offer different goal tracking strategies.  Overall, the Health and Fitness Tracker application combines robust software design principles with practical functionality to create a tool that can help users achieve their fitness goals while ensuring the system remains flexible and easy to extend. |

**Class Diagram (UML)**

Source Code (C#)

using System;

using System.Windows.Forms;

using HealthAndFitnessTracker.Models;

namespace HealthAndFitnessTrackerUI

{

public partial class MainForm : Form

{

public MainForm()

{

InitializeComponent();

}

private void btnLogWorkout\_Click(object sender, EventArgs e)

{

// Logic to log a workout

MessageBox.Show("Workout logged successfully!");

}

private void btnLogMeal\_Click(object sender, EventArgs e)

{

// Logic to log a meal

MessageBox.Show("Meal logged successfully!");

}

private void btnSetGoal\_Click(object sender, EventArgs e)

{

// Logic to set a goal

MessageBox.Show("Goal set successfully!");

}

private void btnGenerateReport\_Click(object sender, EventArgs e)

{

// Logic to generate and display a report

MessageBox.Show("Report generated!");

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using AppUser = ConsoleApp1.Models.User;

namespace ConsoleApp1.Models

{

internal class Database

{

public List<User> Users { get; private set; } = new List<User>();

public List<Workout> Workouts { get; private set; } = new List<Workout>();

public List<Meal> Meals { get; private set; } = new List<Meal>();

// Method to Save Data (Simulated)

public void SaveData()

{

Console.WriteLine("Data has been saved successfully.");

// In a real application, data would be written to a database or file system.

}

// Method to Load Data (Simulated)

public void LoadData()

{

Console.WriteLine("Data has been loaded successfully.");

// In a real application, data would be read from a database or file system.

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

internal class Report

{

public string GenerateWorkoutReport(List<Workout> workouts)

{

if (workouts == null || workouts.Count == 0)

{

return "No workouts logged.";

}

double totalDuration = 0;

double totalCalories = 0;

foreach (var workout in workouts)

{

totalDuration += workout.Duration;

totalCalories += workout.CaloriesBurned;

}

return $"Total Workouts: {workouts.Count}, Total Duration: {totalDuration} minutes, Total Calories Burned: {totalCalories} calories.";

}

// Method to Generate Meal Report

public string GenerateMealReport(List<Meal> meals)

{

if (meals == null || meals.Count == 0)

{

return "No meals logged.";

}

double totalCalories = 0;

foreach (var meal in meals)

{

totalCalories += meal.Calories;

}

return $"Total Meals: {meals.Count}, Total Calories Consumed: {totalCalories} calories.";

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

internal abstract class Workout

{

public double Duration { get; set; } // Duration in minutes

public double CaloriesBurned { get; set; }

public DateTime Date { get; set; }

// Constructor

protected Workout(double duration, double caloriesBurned, DateTime date)

{

Duration = duration;

CaloriesBurned = caloriesBurned;

Date = date;

}

// Abstract Method

public abstract double CalculateCalories();

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

internal class WorkoutLogger

{

public List<Workout> WorkoutList { get; private set; } = new List<Workout>();

// Method to Log a Workout

public void LogWorkout(Workout workout)

{

if (workout == null)

throw new ArgumentNullException(nameof(workout), "Workout cannot be null.");

WorkoutList.Add(workout);

Console.WriteLine($"Workout logged: {workout.GetType().Name}, Duration: {workout.Duration} minutes, Calories Burned: {workout.CaloriesBurned}");

}

// Method to Get All Workouts

public List<Workout> GetWorkouts()

{

return WorkoutList;

}

// Method to Display Workouts

public void DisplayWorkouts()

{

Console.WriteLine("Logged Workouts:");

foreach (var workout in WorkoutList)

{

Console.WriteLine($"Type: {workout.GetType().Name}, Duration: {workout.Duration} minutes, Calories: {workout.CaloriesBurned}, Date: {workout.Date.ToShortDateString()}");

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

public enum MealType

{

Breakfast,

Lunch,

Dinner,

Snack

}

internal class Meal

{

public string FoodName { get; set; }

public double Calories { get; set; }

public MealType MealType { get; set; }

public DateTime Date { get; set; }

// Constructor

public Meal(string foodName, double calories, MealType mealType, DateTime date)

{

FoodName = foodName;

Calories = calories;

MealType = mealType;

Date = date;

}

// Method to Log Meal

public string LogMeal()

{

return $"Meal: {FoodName}, Calories: {Calories}, Type: {MealType}, Date: {Date.ToShortDateString()}";

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

internal class StrengthWorkout : Workout

{

public int Sets { get; set; }

public int Reps { get; set; }

public double WeightLifted { get; set; } // Weight lifted in kilograms or pounds

// Constructor

public StrengthWorkout(double duration, int sets, int reps, double weightLifted, DateTime date)

: base(duration, 0, date) // Initializing base class (calories default to 0 initially)

{

Sets = sets;

Reps = reps;

WeightLifted = weightLifted;

}

// Overriding CalculateCalories Method

public override double CalculateCalories()

{

// Example logic: calories burned is proportional to sets, reps, and weight lifted

CaloriesBurned = Sets \* Reps \* WeightLifted \* 0.05;

return CaloriesBurned;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

internal class CardioWorkout : Workout

{

public double Distance { get; set; } // Distance in kilometers or miles

public int HeartRate { get; set; } // Average heart rate during workout

// Constructor

public CardioWorkout(double duration, double distance, int heartRate, DateTime date)

: base(duration, 0, date) // Initializing base class (calories default to 0 initially)

{

Distance = distance;

HeartRate = heartRate;

}

// Overriding CalculateCalories Method

public override double CalculateCalories()

{

// Example logic: calories burned is proportional to duration and heart rate

CaloriesBurned = Duration \* HeartRate \* 0.1;

return CaloriesBurned;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

internal class MealLogger

{

public List<Meal> MealList { get; private set; } = new List<Meal>();

// Method to Log a Meal

public void LogMeal(Meal meal)

{

if (meal == null)

throw new ArgumentNullException(nameof(meal), "Meal cannot be null.");

MealList.Add(meal);

Console.WriteLine($"Meal logged: {meal.FoodName}, Calories: {meal.Calories}, Type: {meal.MealType}, Date: {meal.Date.ToShortDateString()}");

}

// Method to Get All Meals

public List<Meal> GetMeals()

{

return MealList;

}

// Method to Display Meals

public void DisplayMeals()

{

Console.WriteLine("Logged Meals:");

foreach (var meal in MealList)

{

Console.WriteLine($"Food: {meal.FoodName}, Calories: {meal.Calories}, Type: {meal.MealType}, Date: {meal.Date.ToShortDateString()}");

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

internal class NotificationService

{

public void SendNotification(string message)

{

if (string.IsNullOrWhiteSpace(message))

{

throw new ArgumentException("Notification message cannot be empty.", nameof(message));

}

// Simulate sending a notification (e.g., display a console message)

Console.WriteLine($"Notification: {message}");

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

internal class Goal

{

public double TargetWeight { get; set; }

public DateTime Deadline { get; set; }

public string Status { get; private set; } // e.g., "In Progress", "Completed"

// Constructor

public Goal(double targetWeight, DateTime deadline)

{

TargetWeight = targetWeight;

Deadline = deadline;

Status = "In Progress";

}

// Method to Set Goal

public void SetGoal(double targetWeight, DateTime deadline)

{

TargetWeight = targetWeight;

Deadline = deadline;

Status = "In Progress";

}

// Method to Check Progress

public string CheckProgress(double currentWeight)

{

if (currentWeight <= TargetWeight)

{

Status = "Completed";

return "Congratulations! You have achieved your goal.";

}

else if (DateTime.Now > Deadline)

{

Status = "Failed";

return "Deadline passed. Goal not achieved.";

}

else

{

return $"Keep going! Current weight: {currentWeight}, Target weight: {TargetWeight}, Days remaining: {(Deadline - DateTime.Now).Days}";

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using ConsoleApp1.Interfaces;

namespace ConsoleApp1.Models

{

internal class ProgressEvaluator

{

private readonly IGoalTrackingStrategy \_strategy;

// Constructor

public ProgressEvaluator(IGoalTrackingStrategy strategy)

{

\_strategy = strategy ?? throw new ArgumentNullException(nameof(strategy), "Strategy cannot be null.");

}

// Method to Evaluate Progress

public string EvaluateProgress(User user, Goal goal)

{

if (user == null || goal == null)

{

throw new ArgumentNullException("User or Goal cannot be null.");

}

return \_strategy.EvaluateGoal(user, goal);

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Models

{

internal class MealLogger

{

public List<Meal> MealList { get; private set; } = new List<Meal>();

// Method to Log a Meal

public void LogMeal(Meal meal)

{

if (meal == null)

throw new ArgumentNullException(nameof(meal), "Meal cannot be null.");

MealList.Add(meal);

Console.WriteLine($"Meal logged: {meal.FoodName}, Calories: {meal.Calories}, Type: {meal.MealType}, Date: {meal.Date.ToShortDateString()}");

}

// Method to Get All Meals

public List<Meal> GetMeals()

{

return MealList;

}

// Method to Display Meals

public void DisplayMeals()

{

Console.WriteLine("Logged Meals:");

foreach (var meal in MealList)

{

Console.WriteLine($"Food: {meal.FoodName}, Calories: {meal.Calories}, Type: {meal.MealType}, Date: {meal.Date.ToShortDateString()}");

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using ConsoleApp1.Models;

namespace ConsoleApp1.Interfaces

{

internal interface IGoalTrackingStrategy

{

string EvaluateGoal(User user, Goal goal);

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Interfaces

{

internal interface ITrackable

{

string TrackProgress();

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1.Interfaces

{

internal interface IFactory<T>

{

T Create();

}

}

**Include as many figures as you need to show the different parts of your source code.**

**Descriptions of how and where you have used key concepts**

|  |  |
| --- | --- |
| **Concept** | **Description** (around 30 words each) refer to the class diagram and source code figures as you see fit |
| Abstraction (abstract classes and/or interfaces) | We used abstraction in the Workout class, which defines common attributes (Duration, CaloriesBurned) and the abstract method CalculateCalories (). Interfaces like ITrackable abstract tracking functionality for Workout and Meal. |
| Concrete classes | Concrete classes like CardioWorkout, StrengthWorkout, and Meal implement specific functionalities. For instance, CardioWorkout overrides CalculateCalories () from Workout to provide calorie calculation logic specific to cardio exercises. |
| Name spaces | Namespaces logically group related classes and interfaces, such as HealthTracker.Users for User, HealthTracker.Workouts for Workout and its subclasses, and HealthTracker.Interfaces for ITrackable, ensuring code organization and modularity. |
| Methods | Methods like LogWorkout () in WorkoutLogger and CalculateCalories () in CardioWorkout encapsulate actions. Abstract methods in Workout and overridden implementations in subclasses demonstrate reusable and specific functionality. |
| Properties | Properties like Duration and CaloriesBurned in Workout provide controlled access to class data. They encapsulate attributes while ensuring validation and flexibility, supporting abstraction and encapsulation across classes like Meal and Goal. |
| Overriding of methods and/or properties | The CardioWorkout and StrengthWorkout classes override the abstract method CalculateCalories () from Workout, providing specific implementations for calorie calculations based on cardio and strength exercises, respectively. |
| Constructor overloading | Constructor overloading is used in the User class to allow multiple ways of creating a user. For example, one constructor initializes Name and Email, while another includes Weight. |
| Enumeration (Enum) | The MealType enum in the Meal class categorizes meals into Breakfast, Lunch, and Dinner, ensuring consistency, simplifying comparisons, and avoiding hardcoding of values throughout the application. |
| Collections (e.g., Lists) | Collections like List<Workout> in WorkoutLogger and List<Meal> in MealLogger store multiple entries efficiently, enabling iteration, aggregation, and retrieval of workouts and meals for user tracking. |
| Loops (e.g., foreach) | The foreach loop is used in WorkoutLogger and MealLogger to iterate through List<Workout> and List<Meal> collections, enabling operations like generating summaries or displaying logged entries. |

|  |  |
| --- | --- |
| **SOLID Principle** | **Description of where the principle is realized and for what purpose** (around 50 Words) |
| 1) Single Responsibility | The WorkoutLogger and MealLogger classes follow SRP by focusing solely on managing workouts and meals, respectively. This separation ensures maintainability, as changes to meal or workout logic don’t affect unrelated parts of the application. |
| 2) Open-Closed | The Workout class adheres to OCP by allowing extension through subclasses like CardioWorkout and StrengthWorkout without modifying the base class. This ensures flexibility when adding new workout types in the future. |
| 3) Liskov Substitution | Subclasses like CardioWorkout and StrengthWorkout can replace the abstract Workout class in any context. This ensures consistent behavior while allowing specific calorie calculation logic, maintaining compatibility and extensibility throughout the system. |
| 4) Interface Segregation | Interfaces like ITrackable and IGoalTrackingStrategy focus on specific functionalities, ensuring classes like Workout and ProgressEvaluator implement only the methods they need, reducing unnecessary dependencies and promoting modular design. |
| 5) Dependency Inversion | The ProgressEvaluator class depends on the IGoalTrackingStrategy interface instead of concrete implementations. This allows flexibility in defining goal evaluation strategies, ensuring high-level modules remain independent of low-level implementations. |

|  |  |
| --- | --- |
| **GOF Design Pattern** | **Description of where the pattern is realized and for what purpose** (around 50 Words) |
| 1. Factory Pattern | The IFactory interface and its implementations, like WorkoutFactory, create instances of CardioWorkout or StrengthWorkout. This centralizes object creation, promoting flexibility and scalability by decoupling client code from specific workout instantiation logic. |
| 1. Builder Pattern | The Goal class uses the Builder Pattern to construct complex goal objects with optional parameters like TargetWeight, Deadline, and Status. This ensures flexibility, readability, and scalability when creating customized fitness goals for users. |
| 1. Strategy Pattern | The ProgressEvaluator class uses the IGoalTrackingStrategy interface to apply different strategies for evaluating user goals, such as weight-based or calorie-based tracking. This promotes flexibility by allowing interchangeable evaluation algorithms without modifying the core logic. |

**User Interface (Windows Forms)**

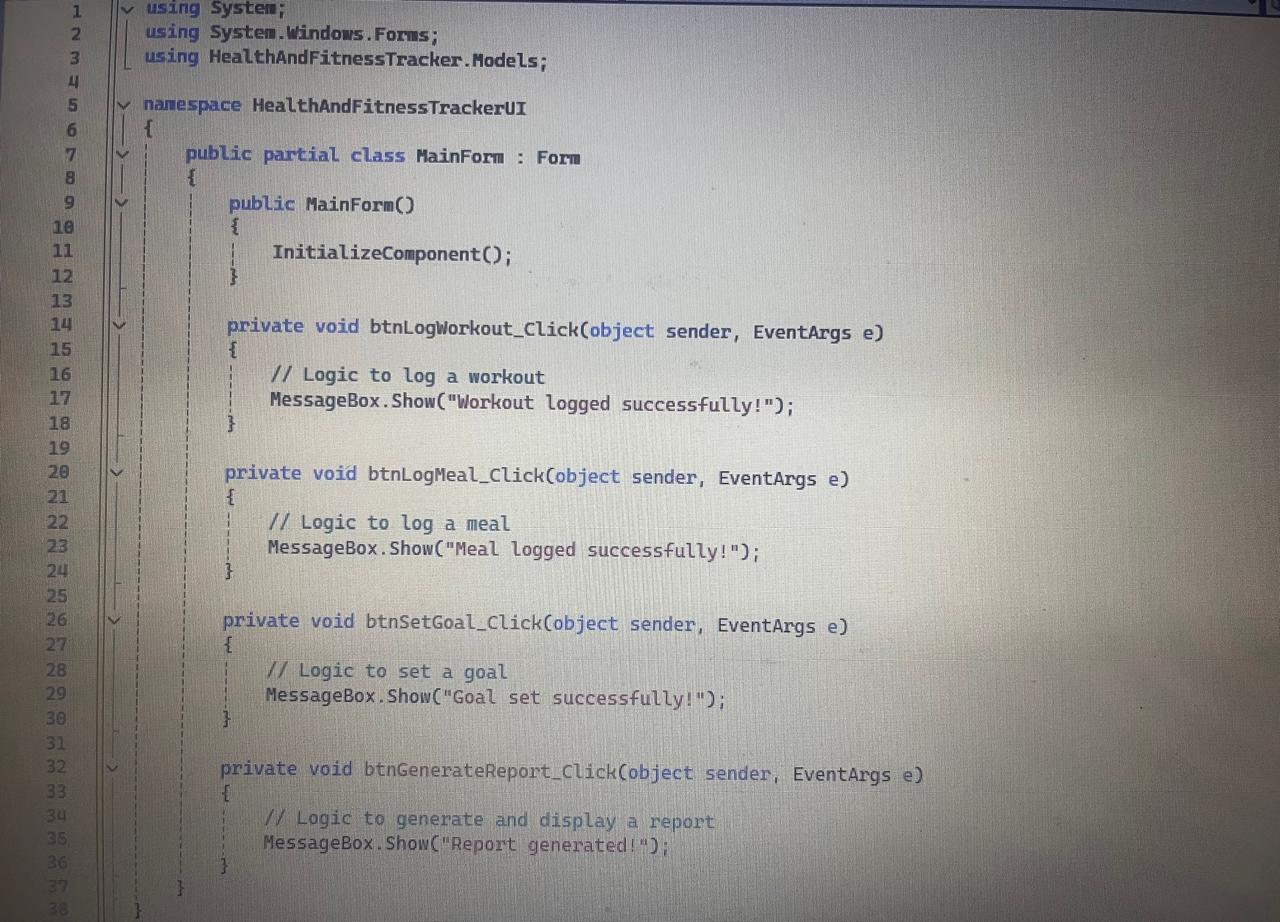


Figure n. User Interface caption

|  |
| --- |
| **Brief Description of UI** (around 50 words) |
| The user interface is a single form with buttons for logging workouts and meals, setting fitness goals, and generating progress reports. Input fields allow users to enter workout details, meal information, and goals, while a display area shows summaries, ensuring simple, functional interaction with the application's core features. |