

CARBON FOOTPRINT CALCULATOR

PROJECT DESIGN DOCUMENT

PROJECT STATUS: Final Implementation

TABLE OF CONTENTS

1. PROJECT OVERVIEW
2. HIGH-LEVEL ARCHITECTURE
3. DESIGN PRINCIPLES AND PATTERNS
4. CLASS HIERARCHY AND DESCRIPTIONS
5. DESIGN DECISIONS AND JUSTIFICATIONS
6. DATA FLOW AND INTERACTIONS
7. EXTENSIBILITY AND FUTURE ENHANCEMENTS
8. KNOWN LIMITATIONS
9. TESTING CONSIDERATIONS

1. PROJECT OVERVIEW

PURPOSE

The Carbon Footprint Calculator is a command-line application designed to help users track and calculate their annual CO₂ emissions from three primary sources: transportation (vehicles), home energy consumption, and dietary choices.

GOALS

- * Provide accurate CO₂ emission calculations based on user input

- * Support multiple emission sources per user
- * Maintain clean, extensible code architecture
- * Offer intuitive command-line interface
- * Enable easy addition of new emission source types

TARGET USERS

- * Individuals wanting to track personal carbon footprint
- * Environmental awareness programs
- * Educational institutions teaching about climate impact
- * Researchers gathering carbon footprint data

2. HIGH-LEVEL ARCHITECTURE

ARCHITECTURAL PATTERN

LAYER RESPONSIBILITIES

Presentation Layer:

- * Handles user input/output
- * Displays menus and prompts
- * Validates basic input format
- * Controls program flow

Business Logic Layer:

- * Performs calculations
- * Aggregates emission sources
- * Provides reporting capabilities

- * Enforces business rules

Data/Domain Model Layer:

- * Defines emission source abstractions
- * Implements emission calculations
- * Stores emission-related data
- * Provides domain-specific behavior

3. DESIGN PRINCIPLES AND PATTERNS

OBJECT-ORIENTED PRINCIPLES APPLIED

1. ABSTRACTION

- * Emission class provides abstract interface for all emission sources
- * Vehicle provides abstract base for all vehicle types
- * Hides implementation details from upper layers

2. ENCAPSULATION

- * Private fields with public getters/setters
- * Data hidden within objects
- * Controlled access to internal state

3. INHERITANCE

- * Clear hierarchy: Emission → Vehicle → Car/Bus/Motorbike
- * Emission → Home, Diet
- * Code reuse through parent classes

4. POLYMORPHISM

- * calculateEmission() method overridden by each subclass
- * All emission sources treated uniformly through Emission interface
- * Enables flexible collection handling

DESIGN PATTERNS USED

1. TEMPLATE METHOD PATTERN

Location: Emission class

Purpose: Defines skeleton of emission calculation

Implementation: Abstract calculateEmission() method implemented by subclasses

Benefit: Consistent interface for all emission calculations

2. FACADE PATTERN

Location: CalculationEngine class

Purpose: Simplifies complex calculation operations

Implementation: Provides simple methods like computeTotal() and breakdown()

Benefit: UI doesn't need to understand User's internal structure

3. COMPOSITE PATTERN (Partial)

Location: User class

Purpose: User aggregates multiple emission sources

Implementation: User contains collection of Vehicles plus Home and Diet

Benefit: Treat individual sources and collections uniformly

4. STRATEGY PATTERN

Location: Different Emission subclasses

Purpose: Different calculation strategies for different emission types

Implementation: Each subclass implements its own calculateEmission()

Benefit: Easy to add new emission calculation methods

SOLID PRINCIPLES

S - Single Responsibility Principle

- ✓ Each class has one clear purpose
- ✓ App handles UI, User handles data, CalculationEngine handles logic

O - Open/Closed Principle

- ✓ Open for extension (add new Vehicle types)
- ✓ Closed for modification (base classes don't need changes)

L - Liskov Substitution Principle

- ✓ Any Vehicle can replace another Vehicle
- ✓ Any Emission can replace another Emission

I - Interface Segregation Principle

- ✓ Classes only expose methods they need
- ✓ No bloated interfaces

D - Dependency Inversion Principle

- ✓ High-level modules (App) depend on abstractions (User, CalculationEngine)
- ✓ Not dependent on low-level details

4. CLASS HIERARCHY AND DESCRIPTIONS

4.1 ABSTRACT BASE CLASSES

CLASS: Emission

Type: Abstract Base Class

Package: Default (root)

Purpose: Fundamental abstraction for all CO₂-emitting activities

Fields:

- emissionFactor (double): kg CO₂ per unit of activity
- activityData (double): quantity of activity performed

Methods:

- getEmissionFactor(): returns emission factor
- setEmissionFactor(double): sets emission factor
- getActivityData(): returns activity data
- setActivityData(double): sets activity data
- calculateEmission(): abstract method for CO₂ calculation

Design Rationale:

This class embodies the core formula: $\text{emission} = \text{factor} \times \text{activity}$.

By making it abstract, we enforce that all emission sources must provide their own calculation logic while maintaining a consistent interface.

CLASS: Vehicle

Type: Abstract Class (extends Emission)

Package: Default (root)

Purpose: Specialized emission source for transportation

Fields:

- id (String, final): unique vehicle identifier
- fuelType (String, final): fuel type description
- efficiency (double, final): kg CO₂ per km (emission factor)
- activityData (inherited): stores annual distance in km

Methods:

- getId(): returns vehicle ID
- getFuelType(): returns fuel type
- getEfficiency(): returns efficiency
- setDistanceKm(double): updates annual distance
- calculateEmission(): distance × efficiency

4.2 CONCRETE EMISSION CLASSES

CLASS: Car, Bus, Motorbike

Type: Concrete Classes (extend Vehicle)

Package: Default (root)

Purpose: Specific vehicle type implementations

Methods:

- Constructor only (calls super)

CLASS: Home

Type: Concrete Class (extends Emission)

Package: Default (root)

Purpose: Models household electricity emissions

Fields (inherited):

- emissionFactor: kg CO₂ per kWh (grid emission intensity)
- activityData: monthly electricity consumption in kWh

Methods:

- Constructor: Home(kgCo2PerKWh, monthlyKWh)
- calculateEmission(): returns $12 \times \text{monthlyKWh} \times \text{emissionFactor}$

CLASS: Diet

Type: Concrete Class (extends Emission)

Package: Default (root)

Purpose: Models food-related emissions

Fields (inherited):

- emissionFactor: kg CO₂ per dietary unit
- activityData: annual dietary units (e.g., days, meals)

Methods:

- Constructor: Diet(kgCo2PerUnit, annualUnits)
- calculateEmission(): returns $\text{annualUnits} \times \text{emissionFactor}$

Design Rationale:

Uses the base formula directly. The flexibility in "units" allows different dietary tracking methods (daily average, meal counts, etc.) without changing the class structure.

4.3 ALTERNATIVE IMPLEMENTATIONS (UNUSED)

CLASS: DistanceBasedEmission

Type: Concrete Class (extends Emission)

Status: Currently unused

Purpose: Alternative distance-based emission calculation

CLASS: EnergyConsumptionEmission

Type: Concrete Class (extends Emission)

Status: Currently unused

Purpose: Alternative energy-based emission calculation

4.4 DATA AGGREGATION CLASSES

CLASS: User

Type: Concrete Class

Package: Default (root)

Purpose: Aggregates all emission sources for a single person

Fields:

- userId (String, final): unique user identifier
- name (String): display name
- vehicles (List<Vehicle>): collection of user's vehicles
- home (Home): optional home energy profile
- diet (Diet): optional diet profile

Methods:

- getUserId(), getName(), setName()
- getVehicles(), addVehicle(Vehicle)
- getHome(), setHome(Home)
- getDiet(), setDiet(Diet)
- vehiclesEmission(): sum of all vehicle emissions
- homeEmission(): home emission or 0 if null
- dietEmission(): diet emission or 0 if null
- getTotalCarbonFootprint(): sum of all sources

Design Rationale:

User acts as the central data container. It uses:

- List for vehicles (can have 0-n vehicles)
- Single instances for home and diet (one per user)
- Null-safe methods (returns 0 if home/diet not set)

This design allows partial data (e.g., only vehicles, no home/diet) while still producing valid calculations.

4.5 BUSINESS LOGIC CLASSES

CLASS: CalculationEngine

Type: Utility Class (all static methods)

Package: Default (root)

Purpose: Centralizes calculation logic, separates concerns

Methods:

- computeTotal(User): returns total carbon footprint
- breakdown(User): returns Map<String, Double> with categorized emissions

Design Rationale:

This class implements the Facade pattern. Benefits:

1. UI doesn't need to know User's internal structure
2. Calculation logic in one place (easy to test)
3. Can add calculation features without changing User or UI
4. Provides consistent data format for reporting

The breakdown() method returns a LinkedHashMap to preserve insertion order, ensuring consistent report formatting.

4.6 USER INTERFACE CLASSES

CLASS: Main

Type: Entry Point

Package: Default (root)

Purpose: Program entry point

Methods:

- main(String[]): delegates to App.main()

CLASS: App

Type: Controller/CLI Handler

Package: Default (root)

Purpose: Orchestrates user interaction and program flow

Methods:

- main(String[]): main program loop
- safeInt(String): robust integer parsing
- safeDouble(String): robust double parsing

Design Rationale:

App implements the Controller pattern for CLI interaction. It:

- Manages program state and flow
- Handles all user input/output
- Creates and manipulates domain objects
- Delegates calculations to CalculationEngine

The safe parsing methods (safeInt, safeDouble) provide robust error handling, preventing crashes from invalid input. They return default values (0, 0.0) rather than throwing exceptions, allowing the program to continue with defaults when appropriate.

CLASS: AppCLI

Type: Utility Class (all static methods)

Package: Default (root)

Purpose: UI display helpers

Methods:

- banner(): displays program title

- showMenu(): displays menu options