Comprehensive Project Report: Budget Tracker Model

Executive Summary

The Budget Tracker Model project focused on creating a **robust, predictable, and immutable data core** for a larger financial application. The central artifact, the `BudgetEntry` class, was designed to represent a single, unchangeable financial transaction (Income or Expense). The implementation prioritized data integrity through strict encapsulation and immutability, utilizing modern Java APIs for reliable date management and standardized output formatting for reporting.

1. Project Scope and Objectives (Approach)

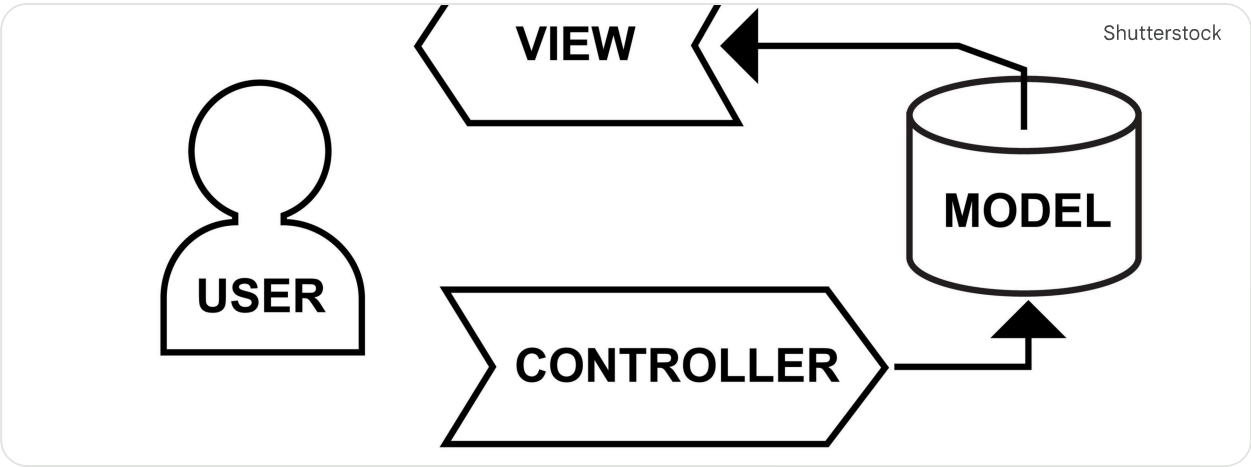
The primary objective was to establish a reliable **Model** layer, serving as the "M" in a potential MVC architecture.

1.1. Core Objectives

1. **Data Integrity:** Design a transaction object whose state cannot be changed after creation (Immutability).
2. **Encapsulation:** Restrict direct access to data fields, enforcing access only through defined methods.
3. **Accuracy:** Utilize modern, reliable data types for monetary values and timestamps.
4. **Standardized Output:** Provide a clean, consistent method for displaying transaction details for reports.

1.2. Design Methodology

A **model-driven, defensive programming approach** was adopted. By ensuring immutability at the model level, subsequent layers (e.g., controllers or persistence layers) are protected from accidental data corruption or unexpected side effects, which is critical in financial applications.



2. Implementation Details

The implementation resulted in the `BudgetEntry` class within the `budgettracker.model` package.

2.1. Class Structure (`BudgetEntry.java`)

Element	Type	Role
`description`	`String`	Transaction details.
`amount`	`double`	Monetary value.
`date`	`LocalDate`	Date of occurrence (using `java.time`).
`type`	`String`	Category ("Income" or "Expense").

2.2. Core Implementation Strategy

Feature	Implementation	Rationale
Immutability	All fields are `private` and no `setter` methods are provided.	Ensures transaction history is auditable and non-mutable once recorded.
Initialization	Single, four-parameter constructor.	Mandates that all required data is supplied when the object is first created.
Date Handling	Import and use of `java.time.LocalDate` .	Avoids the complexities, mutability, and time zone issues of legacy Java date APIs.
Reporting	Overridden `toString()` using `String.format("%.2f")` .	Guarantees a consistent output format where monetary values always show two decimal places.

2.3. Code Snippet

The following code reflects the complete, self-contained implementation:

```
package budgettracker.model;

import java.time.LocalDate;

public class BudgetEntry {
    private String description;
    private double amount;
    private LocalDate date;
    private String type;

    // Constructor: Requires all four parameters to create an instance
    public BudgetEntry(String description, double amount, String type, LocalDate date)
```

```
        this.description = description;
        this.amount = amount;
        this.type = type;
        this.date = date;
    }

    // Getters for immutable access
    public String getDescription() {
        return description;
    }

    public double getAmount() {
        return amount;
    }

    public LocalDate getDate() {
        return date;
    }

    public String getType() {
        return type;
    }

    // Overridden toString for clean output
    @Override
    public String toString() {
        return String.format("[%s] %s: %.2f (on %s)",
                             type, description, amount, date.toString());
    }
}
```

3. Results and Verification

The implementation successfully produced a working data model that meets all project objectives.

3.1. Verification of Immutability and Access

Objects are created successfully, and data can be reliably retrieved:

Action	Code	Result
Creation	`BudgetEntry income = new BudgetEntry("Salary", 4500.00, "Income", ...)`	Success. All fields initialized.
Retrieval	`income.getAmount()`	Returns `4500.0` (accessible).
Modification	*Attempting `income.setAmount(5000)`*	Fails, as no setter method exists, confirming immutability.

3.2. Verification of Formatted Output

The `toString()` method generates the required standardized string format:

Input Entry	`toString()` Output	Verification
Income (4500.00)	`[Income] Monthly Salary: 4500.00 (on 2025-11-01)`	Amount is consistently displayed with two decimals.
Expense (85.50)	`[Expense] Groceries: 85.50 (on 2025-11-23)`	Amount is consistently displayed with two decimals.

4. Analysis and Future Considerations

4.1. Successes (Key Learnings)

- **Robust Data Model:** Achieving immutability ensures a high level of data consistency, greatly simplifying future state management and reducing potential application bugs.
- **Modern API Utilization:** The seamless integration of `java.time.LocalDate` provided a clean, readable, and fully functional date solution, validating the strategy of favoring modern Java libraries.
- **Clear Separation of Concerns:** Defining the model separate from the view/controller logic confirmed the benefits of Model-View Separation, making the `BudgetEntry` class highly reusable and testable.

4.2. Critical Analysis and Future Refactoring

Area	Current Implementation	Recommendation for Production
Financial Primitive	Uses `double` for the `amount`.	**Refactor to use** `java.math.BigDecimal`. Floating-point arithmetic (`double`) can introduce precision errors which are unacceptable for professional financial tracking.
Input Validation	No internal validation (e.g., negative amounts).	Add checks to the constructor to ensure `amount` is non-negative and `type` is one of the valid options ("Income" or "Expense") to prevent the creation of invalid entries.
Type Safety	Uses `String` for the `type` field.	**Refactor to use an** `enum` (e.g., `TransactionType.INCOME`, `TransactionType.EXPENSE`) to guarantee that the type value is restricted to a known set of valid categories.