Budget Analyser

Comprehensive Documentation

Budget Analyser Team

August 10, 2025

This document provides comprehensive documentation for the Budget Analyser application, including its architecture, components, functionality, and usage instructions.

Version 1.0
August 10, 2025

Contents

Chapter 1

Introduction

1.1 Overview

Budget Analyser is an application designed to help users analyze and review their financial data by processing bank account statements. The application provides tools for categorizing transactions, generating reports, and visualizing financial data to help users better understand their spending habits and financial health.

1.2 Purpose

The primary purpose of the Budget Analyser is to:

- Process and analyze bank account statements
- Categorize financial transactions automatically
- Generate detailed reports on earnings and expenses
- Provide insights into spending patterns and financial trends
- Help users make informed financial decisions

1.3 Target Audience

The Budget Analyser is designed for:

- Individual users who want to track their personal finances
- Small business owners who need to monitor business expenses
- Financial advisors who assist clients with financial planning
- Anyone interested in gaining better insights into their financial data

Chapter 2

System Architecture

2.1 Overview

The Budget Analyser application follows a layered architecture pattern, specifically implementing the Model-View-Controller (MVC) design pattern with additional framework utilities. This architecture separates the application into distinct components, each with specific responsibilities, making the system more maintainable, extensible, and testable.

2.2 Architectural Layers

2.2.1 View Layer

The View layer is responsible for the user interface components and user interaction. It includes:

- **Ui_Widget**: The login screen UI
- **Ui_MainWindow**: The main dashboard UI
- **InputDisplayApp** and **ExpenseReport**: Test UI components

2.2.2 Model Layer

The Model layer is responsible for data management and business logic. It includes:

- **Statements**: Collects and formats financial transaction data
- **OriginalStatement**: Provides raw transaction data
- **StatementFormatter**: Formats raw statements into a consistent format

2.2.3 Controller Layer

The Controller layer acts as an intermediary between the View and Model layers, processing user input and updating the model and view accordingly. It includes:

- **Report**: Generates various financial reports from transaction data
- **Processor**: Processes raw transactions by adding categorization

2.2.4 Framework Layer

The Framework layer provides utility classes and services used by the other layers. It includes:

- **Logger**: Provides centralized logging (implemented as a Singleton)
- **PandasToolkit**: Utility methods for pandas DataFrame operations
- **JsonHandler**: Handles JSON file loading and parsing

2.3 Component Relationships

The main application flow starts in `main_be.py`, which creates instances of `Statements`, `Processor`, and `Report` to process financial data and generate reports. The relationships between components are as follows:

- **main_be** creates instances of **Statements**, **Processor**, and **Report**
- **Ui_Widget** opens the dashboard (**Ui_MainWindow**)
- **Statements** uses **OriginalStatement** and **StatementFormatter**
- **Statements**, **Processor**, and **Report** use **PandasToolkit** for data operations
- **Statements** and **Processor** use **Logger** for logging
- **Processor** uses **JsonHandler** for mapping
- **InputDisplayApp** creates **ExpenseReport**

These static relationships are complemented by sequence diagrams in Appendix A, which illustrate the dynamic interactions between components over time. The data processing sequence diagram shows how data flows through the system during transaction processing, while the UI flow sequence diagram illustrates how the user interacts with the application's interface components.

Chapter 3

Components

3.1 View Components

3.1.1 Login Screen (Ui_Widget)

The login screen is the entry point of the application's user interface. It provides:

- A password-based authentication system
- A modern, dark-themed interface with visual elements
- Access to the main dashboard upon successful authentication

3.1.2 Dashboard (Ui_MainWindow)

The dashboard is the main interface of the application after login. It features:

- A sidebar with navigation buttons (Home, Earnings, Expenses, Upload, Mapper, Settings, Logout)
- A toggle between icon-only and full-name views for the sidebar
- A month selector for filtering data by time period
- Different pages for different functions (Home, Earnings, Expenses, Settings)
- Tables for displaying financial data

3.2 Model Components

3.2.1 Statements

The `Statements` class is responsible for collecting and managing transaction data. It:

- Uses `OriginalStatement` to get raw transaction data
- Uses `StatementFormatter` to format the data consistently
- Provides a unified view of transactions from different sources

3.2.2 OriginalStatement

The `OriginalStatement` class provides access to raw transaction data from different sources:

- Credit card statements
- Checking account statements

3.2.3 StatementFormatter

The `StatementFormatter` class is responsible for converting raw statement data into a consistent format that can be processed by the application.

3.3 Controller Components

3.3.1 Report

The `Report` class generates various financial reports based on transaction data:

- Earnings reports
- Expenses reports
- Category-based expense reports
- Sub-category-based expense reports

3.3.2 Processor

The `Processor` class enhances raw transaction data with additional information:

- Adds category information to transactions
- Adds sub-category information to transactions
- Identifies transactions as credits or debits
- Uses mapping data from JSON files for categorization

3.4 Framework Components

3.4.1 Logger

The `Logger` class provides centralized logging functionality:

- Implemented as a Singleton to ensure a single logging instance
- Supports different log levels (debug, info, error, critical, exception)
- Provides special logging for tabular data

3.4.2 PandasToolkit

The `PandasToolkit` class provides utility methods for working with pandas DataFrames:

- Filtering rows based on conditions
- Adding columns with derived data
- Concatenating DataFrames

3.4.3 JsonHandler

The `JsonHandler` class provides functionality for working with JSON files:

- Loading JSON data from files
- Parsing JSON data into Python objects

Chapter 4

Functionality

4.1 Data Processing Flow

The main data processing flow in the Budget Analyser application is as follows:

1. Raw transaction data is collected from various sources (credit cards, checking accounts) by the `OriginalStatement` class.
2. The raw data is formatted into a consistent structure by the `StatementFormatter` class.
3. The formatted data is collected and managed by the `Statements` class.
4. The `Processor` class enhances the transaction data with categories and other information.
5. The processed data is grouped by month for analysis.
6. The `Report` class generates various reports based on the processed data.
7. The reports are displayed to the user through the UI or logged for later reference.

This data processing flow is visually represented in the Data Processing Flow sequence diagram in Appendix A. The sequence diagram illustrates the interactions between components and the flow of data through the system, providing a clear visual representation of the process described above.

4.2 User Interface Flow

The user interface flow in the Budget Analyser application is as follows:

1. The user starts the application and is presented with the login screen (`Ui_Widget`).
2. After entering the correct password, the user is taken to the dashboard (`Ui_MainWindow`).
3. The dashboard provides access to different sections of the application through the sidebar navigation.
4. The user can select a specific month to view data for that time period.
5. The user can view different types of reports (earnings, expenses, etc.) by navigating to the corresponding pages.
6. The user can upload new statement data, map transactions to categories, adjust settings, or log out using the sidebar options.

This user interface flow is visually represented in the UI Flow sequence diagram in Appendix A. The sequence diagram illustrates the interactions between the user and the application's interface components, showing how the user navigates through the application and how the different UI components respond to user actions.

4.3 Key Features

4.3.1 Transaction Categorization

The Budget Analyser automatically categorizes transactions based on predefined rules and mappings, helping users understand where their money is going.

4.3.2 Financial Reporting

The application generates various financial reports, including:

- Earnings reports showing sources of income
- Expense reports showing where money is being spent
- Category-based reports showing spending by category
- Sub-category-based reports showing detailed spending patterns

4.3.3 Time-Based Analysis

The application groups data by month, allowing users to analyze their financial patterns over time and identify trends or anomalies.

4.3.4 User-Friendly Interface

The application features a modern, intuitive user interface with:

- A dark theme for reduced eye strain
- Clear navigation options
- Responsive design elements
- Tabular data presentation for easy comprehension

Chapter 5

Installation and Setup

5.1 Prerequisites

Before installing the Budget Analyser, ensure you have the following prerequisites:

- Python 3.9 or higher
- pip (Python package installer)
- Required Python packages (listed in requirements.txt)

5.2 Installation Steps

1. Clone the repository:

   ```
       git clone https://github.com/username/Analyser.git
       cd Analyser
   ```

2. Install the required dependencies:

   ```
       pip install -r requirements.txt
   ```

3. Set up any necessary configuration files (if applicable).

5.3 Running the Application

5.3.1 Running the Backend

To run the backend processing without the GUI:

```
python source/main_be.py
```

5.3.2 Running the GUI

To run the application with the graphical user interface:

```
python source/view/login.py
```

Chapter 6

Usage Guide

6.1 Logging In

To log in to the Budget Analyser:

1. Launch the application.
2. Enter the password in the password field (default: "password").
3. Click the "Login" button.

6.2 Navigating the Dashboard

The dashboard provides several navigation options:

- **Home**: View the main dashboard overview.
- **Earnings**: View reports on income sources.
- **Expenses**: View reports on spending.
- **Upload**: Upload new statement data.
- **Mapper**: Configure transaction categorization rules.
- **Settings**: Adjust application settings.
- **Logout**: Exit the application.

6.3 Viewing Reports

To view financial reports:

1. Navigate to the desired report section (Earnings, Expenses, etc.).
2. Select the desired month from the month selector.
3. Review the tabular data presented in the report.

6.4 Uploading Statements

To upload new statement data:

1. Navigate to the Upload section.
2. Follow the prompts to select and upload statement files.
3. The application will process the new data and update the reports.

Chapter 7

Development Guide

7.1 Project Structure

The Budget Analyser project follows a structured organization:

- **documentation/**: Contains project documentation, including UML diagrams.
- **resources/**: Contains resources needed for documentation.
- **source/**: Contains the application source code.
  - **controller/**: Contains controller components.
  - **database/**: Contains database-related components.
  - **framework/**: Contains utility and framework components.
  - **model/**: Contains data model components.
  - **view/**: Contains UI components.

7.2 Adding New Features

When adding new features to the Budget Analyser, follow these guidelines:

1. Identify the appropriate layer for the new feature (View, Model, Controller, or Framework).
2. Create new classes or extend existing ones as needed, following the established patterns.
3. Update the UML diagrams to reflect the new components and relationships.
4. Add appropriate logging using the Logger class.
5. Update the documentation to include the new feature.

7.3 Coding Standards

When contributing to the Budget Analyser, follow these coding standards:

- Use PEP 8 style guidelines for Python code.
- Use meaningful variable and function names.
- Add docstrings to all classes and methods.

- Include appropriate logging statements.
- Write unit tests for new functionality.

Chapter 8

Future Enhancements

Based on the planning document, several potential enhancements could be added to the Budget Analyser in future versions:

8.1 Planned Features

8.1.1 Predictive Budgeting

Implement predictive models to forecast future spending based on historical data and create alerts when expenses are projected to exceed the budget.

8.1.2 Savings Goal Tracker

Add functionality to set and track long-term savings goals, visualize progress, and suggest monthly savings contributions based on current spending patterns.

8.1.3 Bill Payment Reminders

Create a reminder system for bill payments with automated notifications and analyze historical data to predict upcoming bills.

8.1.4 Credit Card and Loan Management

Add features to track credit card usage and payments, display interest accrued, and suggest payoff strategies to minimize interest payments.

8.1.5 Investment and Net Worth Tracker

Implement functionality to track investments and calculate overall net worth, with performance comparison to market benchmarks.

8.1.6 Mobile Integration

Develop a mobile app or bot where users can input expenses or view budget summaries on the go, with cloud synchronization.

Chapter 9

Troubleshooting

9.1 Common Issues

9.1.1 Login Issues

If you're having trouble logging in:

- Ensure you're using the correct password (default: "password").
- Restart the application and try again.

9.1.2 Data Processing Issues

If the application is not processing data correctly:

- Ensure your statement files are in the expected format.
- Check the logs for error messages.
- Verify that the mapping files are correctly configured.

9.1.3 UI Issues

If you're experiencing UI problems:

- Ensure you have the required PyQt6 version installed.
- Check that all resource files are in the correct locations.
- Restart the application to refresh the UI.

9.2 Getting Help

If you need additional help with the Budget Analyser:

- Check the documentation for guidance.
- Look for error messages in the application logs.
- Contact the development team for support.

Chapter 10

Conclusion

10.1 Summary

The Budget Analyser is a powerful tool for analyzing financial data, providing insights into spending patterns, and helping users make informed financial decisions. With its intuitive interface, automatic categorization, and detailed reporting, it simplifies the process of managing personal or business finances.

10.2 Feedback and Contributions

The Budget Analyser is an evolving project, and feedback and contributions are welcome. If you have suggestions for improvements or would like to contribute to the development, please contact the development team or submit pull requests to the repository.

Appendix A

UML Diagrams

A.1 Class Diagram

The class diagram shows the structure of the Budget Analyser application, including the main classes, their attributes and methods, and the relationships between them.

The class diagram is defined in the `class_diagram.puml` file located in the `documentation/uml/` directory. To generate a visual representation of this diagram, you can use one of the following methods:

1. **Online PlantUML Server**:
   - Visit `https://www.plantuml.com/plantuml/uml/`
   - Copy and paste the content of `class_diagram.puml` into the text area
   - The diagram will be generated automatically
2. **Using PlantUML locally**:
   - Install PlantUML (requires Java): `https://plantuml.com/starting`
   - Run the command: `java -jar plantuml.jar class_diagram.puml`
   - This will generate a PNG image in the same directory
3. **Using an IDE Plugin**:
   - Many IDEs (IntelliJ, VS Code, etc.) have PlantUML plugins
   - Install the plugin and open the `.puml` file to view the diagram

The diagram illustrates the following key components and their relationships:

- **View Layer**: Ui_Widget, Ui_MainWindow, InputDisplayApp, ExpenseReport
- **Model Layer**: Statements, OriginalStatement, StatementFormatter
- **Controller Layer**: Report, Processor
- **Framework Layer**: Logger, PandasToolkit, JsonHandler

A.2 Sequence Diagrams

The sequence diagrams show the dynamic behavior of the Budget Analyser application, illustrating how objects interact with each other over time to accomplish specific tasks. Two sequence diagrams have been created:

1. **Data Processing Flow**: Shows how data flows through the system during the processing of financial transactions.
2. **UI Flow**: Shows how the user interacts with the application's user interface components.

The sequence diagrams are defined in the `data_processing_sequence.puml` and `ui_flow_sequence.puml` files located in the `documentation/uml/` directory. To generate visual representations of these diagrams, you can use the same methods as for the class diagram:

1. **Online PlantUML Server**:
   - Visit `https://www.plantuml.com/plantuml/uml/`
   - Copy and paste the content of the `.puml` file into the text area
   - The diagram will be generated automatically
2. **Using PlantUML locally**:
   - Install PlantUML (requires Java): `https://plantuml.com/starting`
   - Run the command: `java -jar plantuml.jar filename.puml`
   - This will generate a PNG image in the same directory
3. **Using an IDE Plugin**:
   - Many IDEs (IntelliJ, VS Code, etc.) have PlantUML plugins
   - Install the plugin and open the `.puml` file to view the diagram

A.2.1 Data Processing Flow Sequence Diagram

The Data Processing Flow sequence diagram illustrates:

- How the application processes financial transaction data
- The interactions between main_be, Statements, Processor, and Report components
- How raw transaction data is collected, processed, and used to generate reports
- The role of the Logger in recording operations throughout the process
- The monthly grouping and reporting process

A.2.2 UI Flow Sequence Diagram

The UI Flow sequence diagram illustrates:

- The login process and authentication flow
- How the dashboard is displayed after successful login
- Navigation between different pages of the application
- The user's interaction with various UI components
- The logout process

Appendix B

API Reference

B.1 Model API

B.1.1 Statements

```python
class Statements:
    def __init__(self):
        # Initialize the Statements object

    def collect_transactions(self):
        # Collect transactions from various sources

    @staticmethod
    def future_method():
        # Placeholder for future functionality
```

B.1.2 OriginalStatement

```python
class OriginalStatement:
    def __init__(self):
        # Initialize the OriginalStatement object
        self.from_credit_cards = ...
        self.from_checking_accounts = ...
```

B.1.3 StatementFormatter

```python
class StatementFormatter:
    def __init__(self, account_name, statement):
        # Initialize the StatementFormatter object
        self.account_name = account_name
        self.statement = statement

    def get_desired_format(self):
        # Format the statement into the desired format
```

B.2 Controller API

B.2.1 Report

```
class Report:
    def __init__(self, statement):
        # Initialize the Report object
        self.statement = statement

    def earnings(self):
        # Generate earnings report

    def expenses(self):
        # Generate expenses report

    def expenses_category(self):
        # Generate expenses by category report

    def expenses_sub_category(self):
        # Generate expenses by sub-category report
```

B.2.2 Processor

```
class Processor:
    def __init__(self, raw_transactions):
        # Initialize the Processor object
        self.__raw_transactions = raw_transactions
        self.processed_transactions = None

    def process_transactions(self):
        # Process the raw transactions

    def _add_sub_category(self):
        # Add sub-category to transactions

    def _add_category(self):
        # Add category to transactions

    def _add_c_or_d(self):
        # Add credit or debit indicator

    @staticmethod
    def mapper(content, hash_map):
        # Map content using the hash map
```

B.3 Framework API

B.3.1 Logger

```
class Logger:
    _instance = None
    _initialized = False

    @staticmethod
    def debug(message):
        # Log debug message

    @staticmethod
    def info(message):
        # Log info message

    @staticmethod
    def error(message):
        # Log error message

    @staticmethod
    def critical(message):
        # Log critical message

    @staticmethod
    def exception(message):
        # Log exception message

    @staticmethod
    def table(table, header):
        # Log tabular data

    @staticmethod
    def __get_logger():
        # Get the logger instance

    @staticmethod
    def __log(level, tag, message):
        # Log a message with the specified level and tag

    @staticmethod
    def get_tag_info(frame_index):
        # Get tag information from the call stack
```

B.3.2 PandasToolkit

```
class PandasToolkit:
    @staticmethod
    def filter_rows(df, column_name, condition):
        # Filter DataFrame rows based on condition

    @staticmethod
    def add_column(df, column_name, source_column, func):
        # Add a new column to the DataFrame

    @staticmethod
    def concat_dataframes(df1, df2):
        # Concatenate two DataFrames
```

B.3.3 JsonHandler

```
class JsonHandler:
    def __init__(self, file_path):
        # Initialize the JsonHandler object
        self.file_path = file_path
        self.data = None
```