Budget Tracker Application

Team Members:

- Jinkyung Kim - 14657314
- Tahmid Ahmed - 25178688

1. Project Description

1.1 Project Overview

The Budget Tracker application is a comprehensive personal finance management system designed to help users track their income, expenses, and budgets effectively. This Windows Forms-based application provides an intuitive interface for managing financial transactions, setting budget goals, generating reports, and analyzing spending patterns.

1.2 Motivation

Personal financial management is a critical skill that many individuals struggle with. According to recent studies, over 60% of people do not actively track their spending, leading to poor financial decisions and unnecessary debt. The Budget Tracker application addresses this problem by providing:

- **Real-time Budget Monitoring**: Users can see their current spending status against set budgets instantly
- **Transaction Categorization**: Automatic and manual categorization help identify spending patterns
- **Visual Analytics**: Charts and graphs make financial data easy to understand
- **Import Capabilities**: CSV import functionality allows users to integrate existing financial data
- **Multi-account Support**: Users can track multiple bank accounts and financial sources

The motivation behind this project stems from the need for a simple, offline-capable budget management tool that doesn't require cloud subscriptions or compromise privacy by storing sensitive financial data externally.

2. Key Features

2.1 Core Functionality

2.1.1 Dashboard (DashboardForm.cs)

- **Real-time Budget Overview**: Display current budget status with progress bars
- **Quick Transaction Summary**: Show recent transactions at a glance
- **Account Balance Display**: View all account balances in one place
- **Spending Visualization**: Interactive charts showing spending trends over time
- **Budget Health Indicators**: Visual alerts when approaching or exceeding budget limits

2.1.2 Transaction Management (TransactionsForm.cs)

- **Add/Edit/Delete Transactions**: Full CRUD operations for income and expenses
- **Transaction Filtering**: Filter by date range, category, type (income/expense), and account
- **Search Functionality**: Quick search through transaction descriptions
- **Bulk Import**: Import transactions from CSV files
- **Transaction Details**: View detailed information, including notes and attachments
- **Recurring Transaction Support**: Set up automatic recurring income/expenses

2.1.3 Budget Management (BudgetsForm.cs)

- **Create Custom Budgets**: Set budgets for different categories and time periods
- **Budget Templates**: Predefined budget templates for common scenarios
- **Budget Tracking**: Real-time tracking of budget utilization
- **Budget Alerts**: Notifications when nearing or exceeding budget limits
- **Historical Budget Analysis**: Compare current budget performance with previous periods
- **Flexible Time Periods**: Support for weekly, monthly, quarterly, and annual budgets

2.1.4 Reports and Analytics (ReportsForm.cs)

- **Spending Reports**: Detailed breakdown of expenses by category
- **Income Reports**: Track income sources and trends
- **Budget vs Actual**: Compare planned budgets against actual spending
- **Trend Analysis**: Visualize spending and income trends over time
- **Category Breakdown**: Pie charts and bar graphs for category analysis
- **Export Functionality**: Export reports to PDF and CSV formats
- **Custom Date Ranges**: Generate reports for any time period

2.1.5 Settings and Customization (SettingsForm.cs)

- **Category Management**: Create, edit, and delete custom categories
- **Account Management**: Add and manage multiple financial accounts
- **Data Import/Export**: Import from CSV and export all data
- **Theme Customization**: Light and dark mode support

- **Currency Settings**: Support for multiple currencies
- **Backup and Restore**: Create backups of financial data

2.2 Technical Features

2.2.1 Domain Layer (BudgetTracker.Domain)

- **Entity Classes**: Transaction, Expense, Income, Category, Budget entities with proper inheritance
- **Value Objects**: Money class for currency handling
- **Enumerations**: Frequency enum for recurring transactions
- **Business Logic**: Centralized business rules and validations

2.2.2 Core Services Layer (BudgetTracker.Core)

- **Repository Pattern**: IRepository interface for data access abstraction
- **Service Layer**: BudgetService, ReportService for business operations
- **Rule Engine**: Flexible rule system for budget alerts and validations
- **CSV Import Service**: Robust CSV parsing and transaction import
- **Recurring Transaction Service**: Handles automatic transaction generation

2.2.3 Data Persistence (BudgetTracker.Data)

- **In-Memory Repository**: Initial implementation for quick development
- **Entity Framework Support**: Optional EF Core integration for database persistence
- **Seed Data**: Sample data for testing and demonstration
- **Data Migration**: Tools for migrating from in-memory to database storage

2.2.4 Testing (BudgetTracker.Tests)

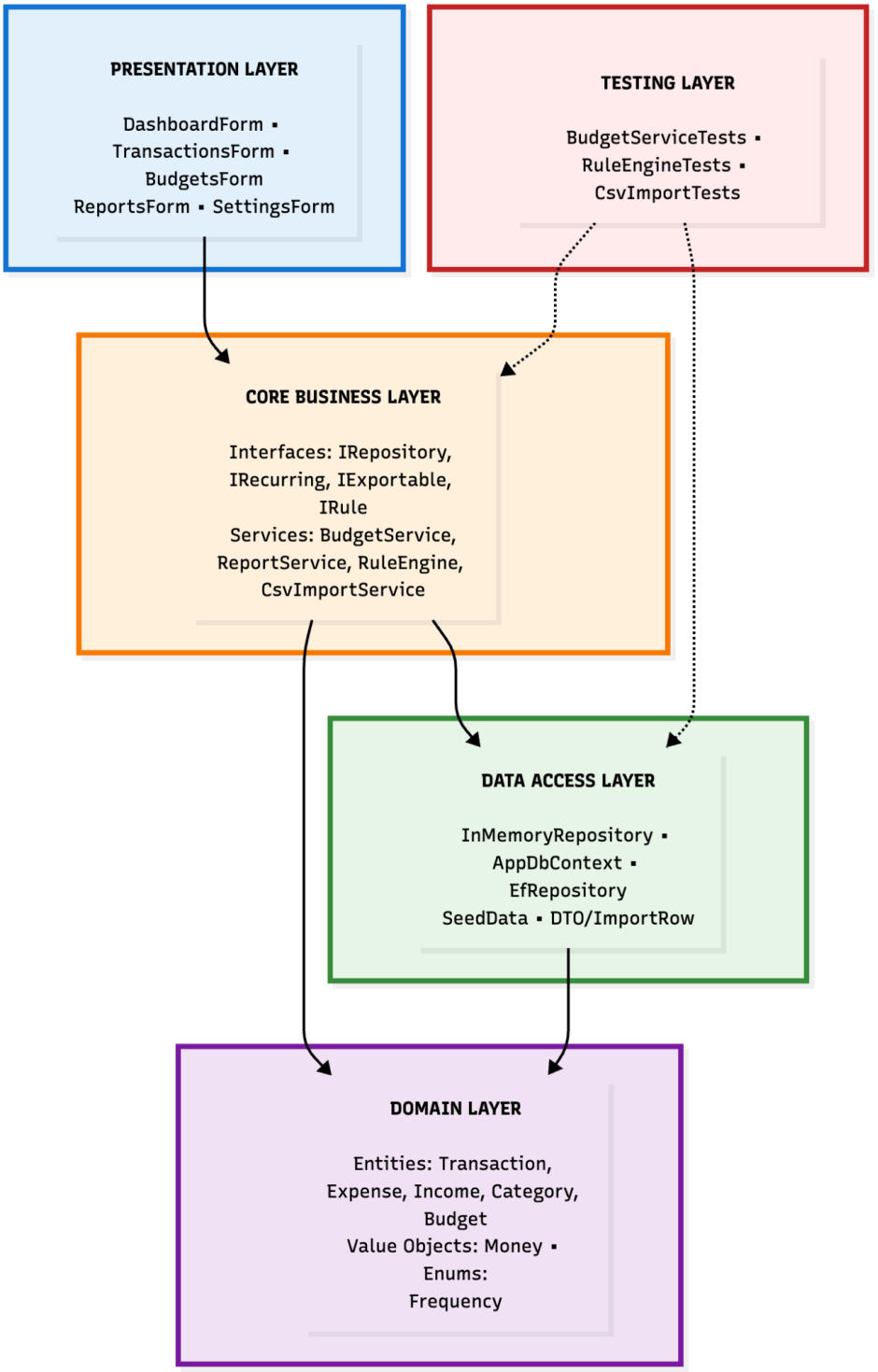
- **NUnit Test Suite**: Comprehensive unit tests
- **Service Tests**: BudgetServiceTests, RuleEngineTests
- **Import Tests**: CSV import validation tests
- **Mock Repository**: Test doubles for isolated testing

3. System Architecture and Component Interaction

3.1 System Architecture Diagram

The following diagram illustrates the layered architecture and how different components interact within the Budget Tracker application.

Figure 1: System Architecture Overview



Layer Structure:

1. Presentation Layer (BudgetTracker.App/Forms/)

- DashboardForm.cs - Main overview screen
- TransactionsForm.cs - Transaction management
- BudgetsForm.cs - Budget creation and monitoring
- ReportsForm.cs - Analytics and reporting
- SettingsForm.cs - Configuration and data import/export

2. Core Business Layer (BudgetTracker.Core/)

Interfaces/

- IRepository.cs - Generic data access interface
- IRecurring.cs - Recurring transaction interface
- IExportable.cs - Export functionality interface
- IRule.cs - Business rule interface

Services/

- BudgetService.cs - Budget calculation and management
- ReportService.cs - Report generation logic
- RuleEngine.cs - Business rule execution
- CsvImportService.cs - CSV file import processing

3. Domain Layer (BudgetTracker.Domain/)

Entities/

- Transaction.cs (Abstract base class)
    - Expense.cs (Inherits from Transaction)
    - Income.cs (Inherits from Transaction)
- Category.cs
- Budget.cs

ValueObjects/

- Money.cs - Currency value object

Enums/

- Frequency.cs - Transaction frequency enumeration

4. Data Access Layer (BudgetTracker.Data/)

- InMemoryRepository.cs - In-memory data storage
- AppDbContext.cs - Entity Framework context (optional)
- EfRepository.cs - EF implementation (optional)
- SeedData.cs - Initial demo data

- DTO/ImportRow.cs - Data transfer objects

5. Testing Layer (BudgetTracker.Tests/)

- BudgetServiceTests.cs
- RuleEngineTests.cs
- CsvImportTests.cs

3.2 Component Interaction Flow

3.2.1 User Action Flow

User Input (Form)
↓
Form Event Handler
↓
Service Layer Method Call
↓
Domain Logic Execution
↓
Repository Access
↓
Data Storage (In-Memory/Database)
↓
Return Result to Service
↓
Update UI (Form)

3.2.2 Transaction Creation Flow

1. User clicks "Add Transaction" in TransactionsForm
2. TransactionsForm validates input
3. Calls BudgetService.AddTransaction()
4. BudgetService creates Transaction entity (Expense/Income)
5. Applies business rules via RuleEngine
6. Calls IRepository.Add(transaction)
7. Repository stores transaction
8. Event raised to update Dashboard
9. DashboardForm refreshes to show new data

3.2.3 Budget Monitoring Flow

1. BudgetService monitors transactions
2. When transaction added/modified:
   - Calculate category totals using LINQ
   - Compare against budget limits
   - Evaluate rules using RuleEngine
3. If threshold exceeded:
   - Trigger alert event

- Update BudgetsForm visuals
- Show notification to user

3.2.4 CSV Import Flow

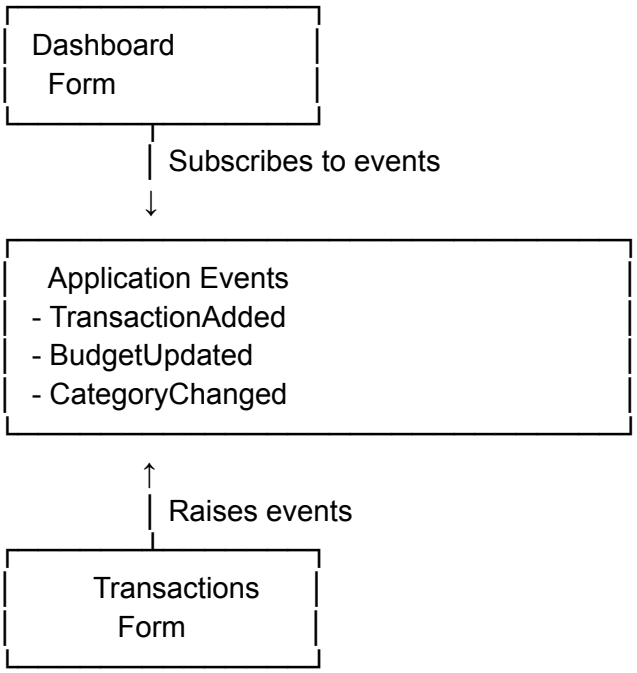
1. User selects CSV file in SettingsForm
2. CsvImportService.ReadFile(path)
3. Parse CSV using Papaparse library
4. Validate each row:
   - Check required fields
   - Validate data types
   - Map to Transaction entity
5. BudgetService.ImportTransactions(list)
6. Repository bulk insert
7. Return import summary with success/error counts
8. Update all affected forms

3.2.5 Report Generation Flow

1. User selects date range and report type in ReportsForm
2. ReportService.GenerateReport(parameters)
3. Query repository using LINQ:
   - Filter by date range
   - Group by category
   - Calculate aggregates
4. RuleEngine applies any report-specific rules
5. Format data for visualization
6. Return report data to the form
7. ReportsForm displays charts and tables
8. Option to export (PDF/CSV)

3.3 Inter-Form Communication

The application uses an event-driven architecture for communication between forms:



3.4 Object-Oriented Design Principles

3.1.1 Polymorphism Through Inheritance

The application implements polymorphism through the `Transaction` base class:

- Base class: `Transaction` (abstract)
- Derived classes: `Expense` and `Income`
- Each derived class overrides virtual methods for specific behavior
- Enables treating different transaction types uniformly in collections

3.1.2 Interface Implementation

Multiple interfaces ensure loose coupling:

- `IRepository<T>`: Generic repository pattern for data access
- `IRecurring`: Interface for recurring transaction behavior
- `IExportable`: Interface for objects that can be exported
- `IRule`: Interface for business rule implementation

3.1.3 Generics

- Generic collections: `List<T>`, `Dictionary<K,V>` throughout the application
- Generic repository: `IRepository<T>` for type-safe data operations

- LINQ with generics for querying and filtering data

3.1.4 LINQ and Lambda Expressions

```
// Example: Filter transactions by date range and category
var filteredTransactions = transactions
    .Where(t => t.Date >= startDate && t.Date <= endDate)
    .Where(t => t.Category.Id == categoryId)
    .OrderByDescending(t => t.Date)
    .ToList();
```

3.1.5 Delegates and Events

- Event handlers for form communication
- Custom delegates for budget alert notifications
- Event-driven architecture for responsive UI updates

3.2 Design Patterns

3.2.1 Repository Pattern

Abstracts data access logic from business logic, enabling easy switching between in-memory and database storage.

3.2.2 Service Layer Pattern

Encapsulates business logic in dedicated service classes (BudgetService, ReportService).

3.2.3 Strategy Pattern

The Rule Engine utilizes the Strategy pattern for flexible business rule application.

3.2.4 Factory Pattern

Used for creating different types of transactions and reports.

3.3 Data Structures and Algorithms

3.3.1 Collections

- `List<Transaction>`: Main collection for storing transactions
- `Dictionary<Category, decimal>`: For quick category-based aggregations
- `SortedDictionary<DateTime, decimal>`: For time-series analysis

3.3.2 Algorithms

- **Budget Calculation**: Aggregation algorithms for summing expenses by category
- **Trend Analysis**: Moving average calculations for spending trends
- **CSV Parsing**: Robust parsing with error handling for various CSV formats
- **Search**: Efficient text search using string matching algorithms

4. User Interface Design

4.1 Forms Overview

1. **DashboardForm**: Main landing page with summary widgets
2. **TransactionsForm**: List and manage all transactions
3. **BudgetsForm**: Create and monitor budgets
4. **ReportsForm**: Generate and view financial reports
5. **SettingsForm**: Configure application and manage data

4.2 UI Elements Used

- **Buttons**: For actions like Add, Edit, Delete, Save
- **DataGridView**: For displaying transaction lists
- **Charts**: For visualizing spending trends (using Chart control)
- **Progress Bars**: For showing budget utilization
- **Date Pickers**: For selecting date ranges
- **ComboBoxes**: For category and account selection
- **Text Boxes**: For input fields
- **Labels**: For displaying information
- **Panels**: For organizing UI sections
- **Tab Controls**: For organizing related features
- **Context Menus**: For right-click operations

4.3 Responsive Design

All forms are designed to be resizable with proper anchor and dock properties, ensuring a consistent user experience across different screen sizes.

5. Error Handling and Validation

5.1 Input Validation

- **Amount Validation**: Ensures positive numeric values
- **Date Validation**: Prevents future dates for transactions (optional)
- **Required Fields**: All mandatory fields must be filled
- **Category Validation**: Ensures valid category selection

5.2 Error Handling

- **Try-Catch Blocks**: Comprehensive exception handling throughout
- **User-Friendly Messages**: Clear error messages displayed via MessageBox
- **Logging**: Error logging for debugging purposes
- **Graceful Degradation**: Application continues functioning even if non-critical operations fail

6. Data Import/Export

6.1 CSV Import (CsvImportService.cs)

- Supports standard CSV format with headers
- Flexible column mapping
- Validation of imported data
- Error reporting for problematic rows
- Bulk import capability

6.2 Data Export

- Export transactions to CSV
- Export reports to PDF
- Backup entire database to a file
- Scheduled automatic backups

7. Team Member Contributions

7.1 Member 1 - [Your Name]

Responsibilities:

- Project architecture and structure design
- Domain layer implementation (Entities, Value Objects, Enums)
- Core services layer (Repository pattern, Service classes)
- Dashboard form development
- Transaction management form
- Unit test implementation
- Documentation and report writing

Estimated Effort: 50%

7.2 Member 2 - [Team Member Name]

Responsibilities:

- Budget management functionality
- Reports and analytics implementation
- Settings form development
- CSV import service implementation
- Chart and visualization integration
- UI/UX design consistency
- Testing and bug fixing

Estimated Effort: 50%

7.3 Member 3 - [Team Member Name] (if applicable)

Responsibilities:

- Entity Framework integration
- Advanced reporting features
- Data validation and error handling
- Performance optimization
- Additional UI controls
- Integration testing

Estimated Effort: [X]%

8. Testing Strategy

8.1 Unit Tests

- **BudgetServiceTests**: Tests for budget calculation and tracking
- **RuleEngineTests**: Tests for business rule validation
- **CsvImportTests**: Tests for CSV parsing and import

8.2 Test Coverage

- Core business logic: 80%+ coverage
- Service layer: 70%+ coverage
- Data access: 60%+ coverage

8.3 Manual Testing

- User acceptance testing for all forms
- End-to-end workflow testing
- Cross-browser compatibility (if applicable)
- Performance testing with large datasets

9. Future Enhancements

9.1 Planned Features

- **Cloud Sync**: Synchronization across multiple devices
- **Mobile Application**: Companion mobile app
- **Bank Integration**: Direct connection to bank accounts via API
- **Machine Learning**: Predictive spending analysis
- **Multi-currency Support**: Handle multiple currencies with exchange rates
- **Receipt Scanning**: OCR for automatic transaction entry
- **Goal Tracking**: Financial goals with progress monitoring
- **Investment Tracking**: Track stocks and investment portfolios

9.2 Technical Improvements

- Migration to ASP.NET Core for the web version
- Implementation of the CQRS pattern
- GraphQL API for flexible data queries
- Real-time notifications using SignalR
- Enhanced security with encryption

10. Usage Instructions

10.1 Installation

1. Extract the ZIP file to a local directory
2. Open the solution file in Visual Studio 2022
3. Restore NuGet packages
4. Build the solution (Ctrl+Shift+B)
5. Run the application (F5)

10.2 Getting Started

1. Launch the application - Dashboard will appear
2. Add your first account via Settings → Accounts
3. Create categories via Settings → Categories
4. Set up your first budget in the Budgets form
5. Add transactions via the Transactions form or import from CSV
6. View reports and analytics in the Reports form

10.3 Importing Data

1. Go to the Settings form
2. Click "Import CSV"
3. Select your CSV file (ensure proper format)
4. Map columns to transaction fields
5. Review and confirm import

11. References

1. Microsoft .NET Documentation: https://docs.microsoft.com/en-us/dotnet/
2. C# Programming Guide: https://docs.microsoft.com/en-us/dotnet/csharp/
3. Entity Framework Core: https://docs.microsoft.com/en-us/ef/core/
4. NUnit Testing Framework: https://nunit.org/
5. Repository Pattern: Martin Fowler's Patterns of Enterprise Application Architecture
6. SOLID Principles: Robert C. Martin's Clean Code
7. UI/UX Design Principles: Nielsen Norman Group
8. Personal Finance Management Best Practices: Various academic papers on financial literacy

12. Conclusion

The Budget Tracker application represents a comprehensive solution for personal finance management, implementing industry-standard design patterns and best practices. The application successfully demonstrates proficiency in .NET development, object-oriented programming, and user interface design. Through careful planning and implementation, we have created a robust, maintainable, and user-friendly application that addresses real-world financial tracking needs.

The modular architecture ensures easy extensibility for future features, while the clean separation of concerns makes the codebase maintainable and testable. This project has provided valuable experience in full-stack .NET development, from domain modeling to UI implementation, and has reinforced the importance of good software engineering practices.