

# Jbudget

Francesco Scotti

June 2025

# Contents

<b>1</b>	<b>Introduzione a JBudget</b>	<b>3</b>
<b>2</b>	<b>Funzionalità Disponibili</b>	<b>4</b>
2.1	Gestione delle Transazioni Finanziarie . . . . .	4
2.2	Sistema di Etichettatura e Categorizzazione . . . . .	5
2.3	Gestione delle Ricorrenze . . . . .	5
2.4	Calcolo e Analisi del Bilancio . . . . .	6
2.5	Filtraggio e Ricerca . . . . .	7
<b>3</b>	<b>Responsabilità Individuate</b>	<b>8</b>
3.1	Responsabilità del Livello di Modello . . . . .	8
3.2	Responsabilità del Livello di Vista . . . . .	9
3.3	Responsabilità del Livello di Controllo . . . . .	9
3.4	Responsabilità del Livello di Persistenza . . . . .	9
3.5	Responsabilità del Filtraggio . . . . .	10
<b>4</b>	<b>Architettura Classi ed Interfacce</b>	<b>11</b>
4.1	Livello di Modello dei Dati . . . . .	11
4.2	Livello di Persistenza . . . . .	12
4.3	Livello di Controllo . . . . .	12
4.4	Sistema di Filtraggio . . . . .	13
4.5	Livello di vista . . . . .	13
4.6	Diagramma UML . . . . .	14
<b>5</b>	<b>Organizzazione dati e persistenza</b>	<b>15</b>
5.1	Modello Logico dei Dati . . . . .	15
5.2	Architettura della Persistenza . . . . .	16
<b>6</b>	<b>Meccanismi di Estendibilità</b>	<b>17</b>
6.1	Estendibilità del Sistema di Persistenza . . . . .	17
6.2	Estendibilità del Bilancio . . . . .	18
6.3	Estendibilità dell'Interfaccia <code>IFinancialEntitiesAdder</code> . . . . .	18
6.4	Indipendenza dall'Interfaccia Utente . . . . .	18
6.5	Estendibilità del Sistema di Filtraggio . . . . .	19

6.6	Estendibilità del Modello di Dati . . . . .	19
6.7	Estendibilità per la sincronizzazione . . . . .	19
6.7.1	Introduzione del Sistema di Account e Autenticazione	19
6.7.2	Modello di Entità Utente . . . . .	20
6.7.3	Architettura di Autenticazione . . . . .	20
6.7.4	Isolamento dei Dati per Utente . . . . .	21
<b>7</b>	<b>Conclusioni</b>	<b>22</b>

# Chapter 1

## Introduzione a JBudget

Il sistema **JBudget** rappresenta un'applicazione pensata per la gestione finanziaria personale e/o familiare. Dal momento che si tratta di un prototipo di quella che in futuro sarà un'applicazione completa, l'obiettivo principale del progetto è stato quello di garantire una chiara separazione delle responsabilità, favorire l'estendibilità e mantenere un'elevata manutenibilità del codice.

L'intera applicazione è stata realizzata in linguaggio **Java**, adottando un approccio modulare che consente di gestire in maniera efficace le transazioni finanziarie, i tag associati e i calcoli relativi al bilancio.

L'architettura del sistema è stata strutturata in modo da distinguere nettamente tra il livello di *modello dei dati*, il livello di *persistenza* e il livello di *controllo applicativo*. Questa organizzazione rende il sistema completamente indipendente da qualsiasi interfaccia utente specifica, permettendo così l'implementazione di diverse tipologie di frontend (desktop, web, mobile) senza dover modificare la logica di business principale. Nel progetto viene fornita un'implementazione desktop che mostri il funzionamento dell'applicazione.

Grazie a un utilizzo attento di interfacce ben definite e classi astratte, il sistema è facilmente evolvibile e consente l'integrazione di nuove funzionalità senza compromettere quanto già realizzato. Questo rende JBudget particolarmente adatto a scenari di sviluppo incrementale e a lungo termine.

## Chapter 2

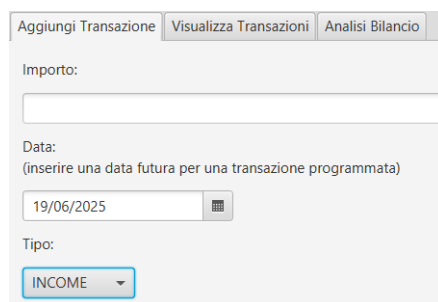
# Funzionalità Disponibili

### 2.1 Gestione delle Transazioni Finanziarie

La funzionalità centrale del sistema è rappresentata dalla gestione delle transazioni finanziarie, basata su un modello dati che supporta sia movimenti in entrata che in uscita. Ogni transazione può essere registrata specificando dettagli quali: importo, tipologia (entrata o spesa), data di esecuzione e uno o più tag (impostato con un massimo di 3, ma facilmente modificabile) per la categorizzazione.

Le transazioni sono identificate univocamente tramite un identificatore numerico che assicura l'integrità referenziale. Gli importi sono gestiti come valori in virgola mobile a doppia precisione, per garantire una rappresentazione accurata dei dati monetari. La tipologia di transazione è definita da un'enumerazione che comprende attualmente due categorie principali: **INCOME** (entrata) e **EXPENSE** (spesa).

Dal momento che è possibile scegliere la data di esecuzione della transazione è anche possibile impostarla come una transazione futura e che verrà considerata come se fosse una scadenza.



The screenshot shows a web form titled 'Aggiungi Transazione' (Add Transaction). At the top, there are three tabs: 'Aggiungi Transazione' (selected), 'Visualizza Transazioni' (View Transactions), and 'Analisi Bilancio' (Balance Sheet Analysis). The form contains the following fields:

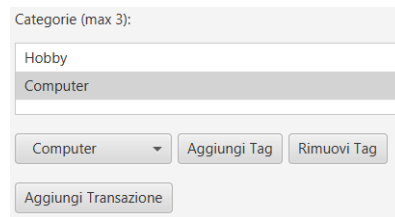
- Importo:** A text input field for the amount.
- Data:** A text input field with the value '19/06/2025'. Below it, a small note says '(inserire una data futura per una transazione programmata)' (insert a future date for a scheduled transaction). To the right of the date field is a small calendar icon.
- Tipo:** A dropdown menu with 'INCOME' selected.

## 2.2 Sistema di Etichettatura e Categorizzazione

Il sistema di tagging rappresenta una componente dell'applicazione progettata per supportare la categorizzazione gerarchica delle transazioni. I tag sono entità strutturate in una relazione padre-figlio, con la possibilità di creare tassonomie articolate.

Ogni tag è definito da un identificatore univoco, un nome descrittivo e, opzionalmente, un riferimento al tag padre. Questa struttura ad albero consente una categorizzazione flessibile. Un esempio tipico è l'organizzazione di una categoria principale "Alimentazione" con sottocategorie quali "Ristoranti", "Supermercato" e "Delivery", utile per un'analisi dettagliata delle abitudini di spesa.

È importante sottolineare che la creazione, rimozione o modifica dei tag viene lasciata allo sviluppatore. Il motivo di questa scelta è stato fatto in modo da poter limitare errori da parte dell'utente: ad esempio ipotizziamo che la creazione di un tag fosse lasciata all'utente. Questo potrebbe portare a situazioni in cui un'utente si sia dimenticato di aver già creato un tag e quindi ne crei un duplicato, come magari creare il tag spesa, oppure SPESA, che verrebbero considerati come due tag diversi.



## 2.3 Gestione delle Ricorrenze

Il sistema offre supporto alla gestione di transazioni ricorrenti mediante l'enumerazione **Recurrence**, che consente di definire pattern di ripetizione automatica. I pattern disponibili includono frequenze giornaliere, settimanali, mensili, annuali e un'opzione "NONE" per le transazioni singole, ma è possibile inserire altri pattern in futuro.

La logica delle ricorrenze è implementata tramite periodi fissi espressi in giorni. Il metodo **addRecurrence** della classe **TransactionAdder** consente di generare automaticamente una serie di transazioni partendo da un modello iniziale, incrementando progressivamente la data fino a una scadenza predefinita.

Tale funzionalità è particolarmente utile nella gestione di spese o entrate periodiche, come affitti, bollette o stipendi, contribuendo a ridurre significativamente il carico manuale di inserimento dati.

Ricorrenza:

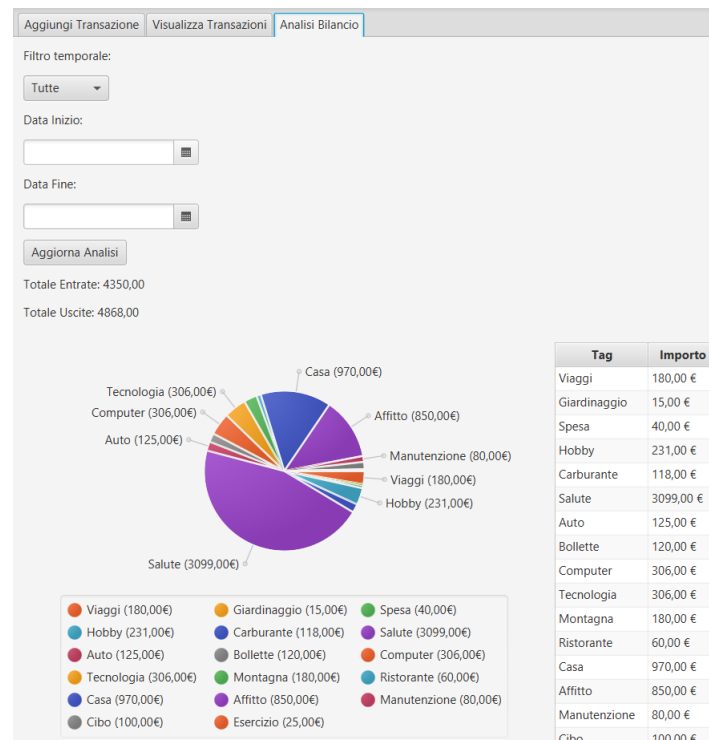
WEEKLY

Fine Ricorrenza (se applicabile):

## 2.4 Calcolo e Analisi del Bilancio

Le funzionalità di calcolo e analisi del bilancio sono implementate tramite la classe **TransactionBalance**, che estende l'astrazione fornita da **AbstractBalanceCalculator**. Questo componente offre metodi per calcolare il totale delle entrate, delle spese e il bilancio netto, sfruttando lo stream processing per garantire performance elevate anche su dataset di grandi dimensioni.

Tra le funzionalità più rilevanti c'è la possibilità di generare analisi aggregate per tag, al fine di visualizzare la distribuzione delle spese per categoria. Tali analisi si basano su operazioni di raggruppamento e aggregazione, che trasformano l'elenco delle transazioni in una mappa di tag e importi totali. Il bilancio viene calcolato come differenza tra entrate e spese, restituendo un indicatore immediato dello stato finanziario. Le operazioni aritmetiche coinvolte sono progettate per essere atomiche, garantendo consistenza anche in contesti di accesso concorrente.



## 2.5 Filtraggio e Ricerca

JBudget integra un meccanismo di filtraggio delle transazioni, basato sull'interfaccia **Filtered** e la relativa implementazione **FilteredTransactions**. Questo componente consente all'applicazione di ottenere viste personalizzate del dataset.

Tra i criteri di filtraggio supportati figurano la tipologia di transazione, l'intervallo temporale e la distinzione tra transazioni passate, future o tutte. Il filtraggio temporale si avvale del **Record Period**, che consente di definire intervalli con date di inizio e fine opzionali, rendendo possibili query flessibili quali “tutte le spese del mese corrente” o “tutte le entrate dell'anno precedente”.

L'implementazione si avvale di *stream processing* e *lambda expressions*, offrendo una combinazione di elevate prestazioni e leggibilità del codice. L'approccio funzionale adottato facilita la costruzione di filtri complessi e la loro applicazione in pipeline di elaborazione dati.

Aggiungi Transazione Visualizza Transazioni Analisi Bilancio				
Tipo transazione:				
EXPENSE				
Filtro temporale:				
Tutte				
Data inizio (opzionale):				
Data fine (opzionale):				
Applica Filtro				
Transazioni:				
Data	Tipo	Importo	Tags	
2025-06-01	EXPENSE	40,00	Spesa, Cibo	
2025-06-02	EXPENSE	60,00	Ristorante, Cibo	
2025-06-03	EXPENSE	850,00	Affitto, Casa	
2025-06-04	EXPENSE	120,00	Bollette, Casa	
2025-06-05	EXPENSE	45,00	Carburante, Auto	
2025-06-06	EXPENSE	80,00	Manutenzione, Auto	
2025-06-07	EXPENSE	25,00	Esercizio, Salute	
2025-06-08	EXPENSE	90,00	Computer, Tecnologia	
2025-06-09	EXPENSE	15,00	Giardinaggio, Hobby	
2025-06-10	EXPENSE	180,00	Montagna, Viaggi	
2025-06-08	EXPENSE	54,00	Tecnologia, Computer, Hobby	
2025-07-08	EXPENSE	54,00	Tecnologia, Computer, Hobby	
2025-08-07	EXPENSE	54,00	Tecnologia, Computer, Hobby	
2025-09-06	EXPENSE	54,00	Tecnologia, Computer, Hobby	
2025-06-11	EXPENSE	58,00	Salute	
2025-06-18	EXPENSE	58,00	Salute	



## Chapter 3

# Responsabilità Individuate

L'architettura del sistema **JBudget** presenta una distribuzione chiara e modulare delle responsabilità tra i diversi componenti, in conformità con i principi di progettazione software consolidati, come il *Single Responsibility Principle*. Viene inoltre sfruttato il pattern architetturale **MVC** (Model, View, Controller).

### 3.1 Responsabilità del Livello di Modello

Il livello di modello si occupa della definizione delle strutture dati e delle regole fondamentali del sistema. Classi **record** come **Transaction**, **Tag** e **Period** incapsulano la rappresentazione dei dati, garantendo immutabilità e sicurezza anche per un eventuale concorrente (*thread-safety*), garantendo estendibilità quindi anche per un'implementazione che sfrutta il multi-threading.

La classe **Period** è responsabile della gestione degli intervalli temporali e fornisce metodi ausiliari per determinare se una data specifica rientri in un dato intervallo. I metodi `getDataStartOrNow` e `getDataEndOrNow` implementano una logica di fallback che utilizza la data corrente in assenza di valori definiti, migliorando la robustezza nelle operazioni temporali.

L'enumerazione **TransactionType** centralizza la definizione delle tipologie di transazione, semplificando estensioni future. Allo stesso modo, l'enumerazione **Recurrence** incapsula la logica di calcolo dei periodi di ricorrenza, fornendo un metodo `period()` che traduce i pattern in valori numerici utili per i calcoli temporali.

### 3.2 Responsabilità del Livello di Vista

Il livello *Vista* ha il compito di gestire l'interazione tra l'utente e il sistema, occupandosi esclusivamente della presentazione delle informazioni e della raccolta degli input. In particolare, le sue responsabilità principali includono:

- **Visualizzazione dei dati finanziari:** mostrare in modo chiaro e organizzato le transazioni, i bilanci e altri elementi rilevanti per l'utente.
- **Raccolta degli input utente:** gestire l'inserimento di nuove transazioni, tag e altre entità finanziarie tramite apposite interfacce grafiche.
- **Aggiornamento dinamico dell'interfaccia:** risponde ai cambiamenti nei dati aggiornando dinamicamente le schermate o le componenti grafiche (es. ricalcolo bilanci, aggiornamento tabelle).
- **Flessibilità e intercambiabilità:** l'interfaccia è progettata per essere facilmente sostituibile, ad esempio passando da una realizzazione JavaFX a una basata su FXML o su altri framework, senza impattare il core dell'applicazione.

### 3.3 Responsabilità del Livello di Controllo

Il livello di controllo coordina la logica e le interazioni tra i vari componenti del sistema, delegando a ciascuna classe una responsabilità funzionale specifica.

La classe `TransactionAdder` è incaricata esclusivamente dell'aggiunta di nuove transazioni, incluse quelle ricorrenti. La concentrazione di responsabilità consente un'implementazione mirata e facilmente manutenibile. La generazione di transazioni ricorrenti è integrata come estensione naturale di tale responsabilità.

La classe astratta `AbstractBalanceCalculator` definisce un modello comune per il calcolo dei bilanci, adottando il *template method pattern* per favorire estensioni coerenti.

La classe `TransactionBalance` fornisce un'implementazione concreta per il dominio delle transazioni.

### 3.4 Responsabilità del Livello di Persistenza

Il livello di persistenza è responsabile della gestione completa del ciclo di vita dei dati, inclusi salvataggio, caricamento e mantenimento della durabilità delle informazioni oltre l'esecuzione dell'applicazione.

L'interfaccia `FinancialEntitiesPersistency` definisce il contratto per le operazioni di persistenza generiche su entità finanziarie. Questa astrazione consente l'intercambiabilità del meccanismo di storage, rendendo possibile l'adozione di differenti tecnologie senza necessità di modifiche al livello applicativo.

L'interfaccia `TransactionPersistency` estende il contratto di base con operazioni specifiche per transazioni e tag. In particolare, il metodo `loadTags()` riconosce i tag come entità distinte che richiedono un caricamento dedicato. La classe `XmlPersistency` rappresenta l'implementazione concreta del livello di persistenza, adottando il formato XML per la memorizzazione. Essa si occupa della conversione tra oggetti Java e rappresentazioni XML (e viceversa), includendo meccanismi di parsing, serializzazione, gestione degli errori e creazione automatica dei file qualora questi non vengano trovati.

### 3.5 Responsabilità del Filtraggio

Il sottosistema di filtraggio si occupa della selezione e visualizzazione di sottoinsiemi di dati in base a criteri personalizzabili.

L'interfaccia `Filtered` stabilisce il contratto per i componenti incaricati della produzione di viste filtrate su collezioni di dati. La classe `FilteredTransactions` implementa questa interfaccia nel dominio delle transazioni, incapsulando la logica per l'applicazione di criteri multipli e combinabili.

La separazione del sottosistema di filtraggio dal resto dell'architettura consente un'evoluzione indipendente dei criteri di selezione e semplifica l'introduzione di nuove funzionalità senza impatti collaterali sugli altri moduli.

## Chapter 4

# Architettura Classi ed Interfacce

L'architettura del sistema **JBudget** adotta una struttura modulare e stratificata, nella quale le classi e le interfacce sono organizzate in livelli logici distinti, ciascuno con responsabilità chiare e ben delimitate.

### 4.1 Livello di Modello dei Dati

Il livello di modello rappresenta il fondamento concettuale del sistema, questo contiene le entità principali per la gestione finanziaria.

**Classe Transaction** La classe **Transaction** è implementata come **record** Java, garantendo l'immutabilità e la generazione automatica di metodi **equals()**, **hashCode()** e **toString()**. Essa incapsula cinque attributi: un identificatore univoco di tipo **long**, l'importo (**double**), il tipo di transazione (**TransactionType**), la data (**LocalDate**) e una lista di tag associati.

**Classe Tag** La classe **Tag** consente la definizione di strutture gerarchiche tramite un riferimento ricorsivo al tag padre. Essa include un identificatore univoco, un nome descrittivo e un riferimento opzionale al genitore. Tale struttura supporta query complesse e aggregazioni lungo la gerarchia. L'implementazione come **record** assicura l'immutabilità della gerarchia.

**Classe Period** La classe **Period** modella intervalli temporali con date opzionali di inizio e fine. I metodi **getDataStartOrNow()** e **getDataEndOrNow()** applicano una strategia di fallback che ricorre alla data corrente quando necessario. Il metodo **contains()** verifica l'inclusione di una data nell'intervallo, sfruttando una logica a estremi inclusivi.

**Enumerazioni di Supporto** L'enumerazione `TransactionType` definisce le categorie di transazione (`INCOME`, `EXPENSE`) centralizzando la logica di tipizzazione. L'enumerazione `Recurrence` modella i pattern temporali ricorrenti e fornisce il metodo `period()` per convertire tali pattern in incrementi numerici che potranno essere usati per calcolare il numero di transazioni da inserire durante la ricorrenza.

## 4.2 Livello di Persistenza

Il livello di persistenza adotta un'architettura a inversione di dipendenza, separando l'interfaccia dalla logica di implementazione, in linea con il *Dependency Inversion Principle*.

**Interfaccia `FinancialEntitiesPersistency`** Definisce un contratto generico per il salvataggio e il caricamento di entità finanziarie. L'uso di generics consente l'astrazione rispetto al tipo di entità, mantenendo coerenza e sicurezza nei tipi.

**Interfaccia `TransactionPersistency`** Estende `FinancialEntitiesPersistency` con il metodo `loadTags()`, riconoscendo i tag come entità distinte che richiedono logiche di caricamento separate.

**Classe `XmlPersistency`** Implementa la persistenza su file utilizzando le API DOM di Java per la manipolazione di XML. Include logiche avanzate per il parsing ricorsivo dei tag, la serializzazione degli oggetti Java e la gestione automatica dei file mancanti. L'utilizzo di costanti per i nomi dei file facilita la manutenzione.

## 4.3 Livello di Controllo

Il livello di controllo gestisce le varie operazioni disponibili, delegando responsabilità a componenti dedicati.

**Interfaccia `IFinancialEntitiesAdder`** Definisce un contratto generico per componenti che si occupano dell'aggiunta di entità finanziarie, mantenendo un'interfaccia coerente attraverso l'uso di generics.

**Classe `TransactionAdder`** Gestisce l'aggiunta di transazioni, incluse quelle ricorrenti. Fa uso di `TransactionPersistency` per ricevere istanze della stessa, favorendo testabilità e configurabilità. Il metodo `addRecurrence()` implementa un algoritmo iterativo per la generazione di transazioni ricorrenti basato su `Recurrence`. Gli identificatori vengono generati con `System.currentTimeMillis()`, garantendo unicità temporale.

**Classe astratta `AbstractBalanceCalculator`** Definisce la struttura generale del calcolo dei bilanci. Fornisce il metodo concreto `getBalance()` che si basa sui metodi astratti `getTotalIncome()` e `getTotalExpense()`. Una lista protetta di elementi è resa disponibile per le sottoclassi. Il metodo `getTagsAmountMap()` supporta analisi aggregate per tag.

**Classe `TransactionBalance`** Specializza `AbstractBalanceCalculator` per il dominio delle transazioni, utilizzando *stream processing* e *lambda expressions* per l'implementazione dei calcoli. Il metodo `getTotals()` consente il filtraggio e l'aggregazione, mentre `getTagsAmountMap()` applica `flatMap` e `groupingBy` per generare mappe di aggregazione per tag.

## 4.4 Sistema di Filtraggio

Il sistema di filtraggio implementa una logica flessibile per la selezione di dati secondo criteri configurabili dall'utente.

**Interfaccia `Filtered`** Stabilisce un contratto minimale per la fornitura di viste filtrate su collezioni di entità. L'uso di generics consente l'applicazione dei filtri a diverse tipologie di tipi ed è stata principalmente pensata per l'introduzione di nuove entità finanziarie.

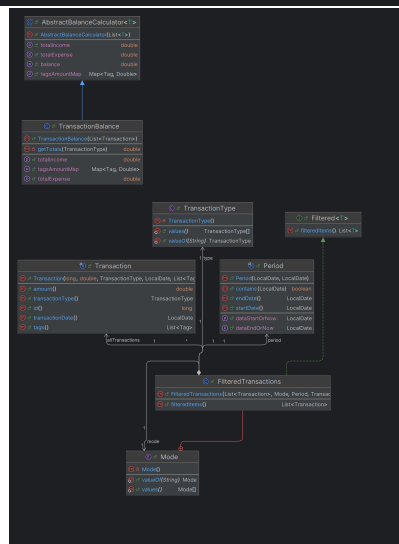
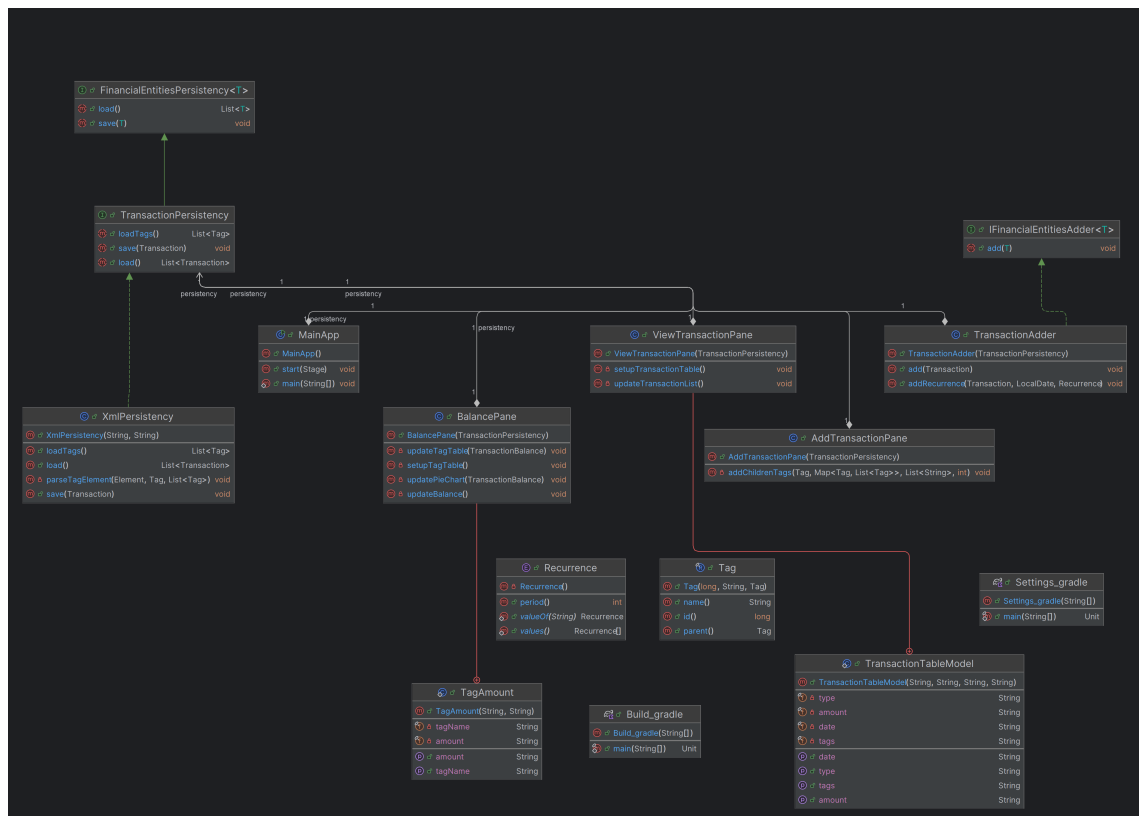
**Classe `FilteredTransactions`** Implementa il filtraggio delle transazioni mediante l'uso di criteri multipli, configurabili tramite un'enumerazione interna `Mode` e parametri opzionali relativi a periodo e tipo. L'implementazione sfrutta *stream processing* e *switch expressions* per l'applicazione efficiente dei filtri e una maggiore leggibilità.

## 4.5 Livello di vista

Il livello *Vista* dell'applicazione `JBudget` è pensato per essere completamente indipendente dalla logica interna del programma. Questo significa che, volendo, si potrebbe sostituire l'interfaccia grafica attuale con una diversa senza dover modificare il cuore del sistema. Ad esempio, l'interfaccia utente attuale è costruita usando `JavaFX`, ma nulla vieta di riscriverla utilizzando `FXML` per una gestione più ordinata e dichiarativa delle componenti grafiche, oppure creare una versione web o mobile sfruttando `API REST`.

Questo approccio flessibile rende il sistema molto più semplice da adattare a nuove esigenze in futuro. Che si voglia aggiungere nuove funzionalità, cambiare completamente il layout, o creare versioni dell'app per altre piattaforme, il livello *Vista* può essere aggiornato in autonomia, lasciando intatto tutto il resto dell'applicativo.

## 4.6 Diagramma UML



## Chapter 5

# Organizzazione dati e persistenza

L'architettura del sistema **JBudget** adotta un modello che separa in maniera netta la rappresentazione logica dei dati dai meccanismi di persistenza fisica, favorendo estendibilità e manutenibilità.

### 5.1 Modello Logico dei Dati

Il modello è basato su tre entità principali: **Transaction**, **Tag** e **Period**, ciascuna modellata per rappresentare con precisione dei concetti finanziari. Prima di parlare in dettaglio nell'effettiva persistenza dei dati è bene ribadire i concetti di organizzazione dei dati già visti in precedenza.

**Transazioni** Le transazioni costituiscono il fulcro del sistema. Sono strutture immutabili che contengono:

- un identificatore univoco basato su timestamp;
- un importo a precisione doppia (`double`);
- una tipologia (`TransactionType`) che distingue tra entrate e spese;
- una data (`LocalDate`);
- una lista di **Tag** associati.

L'enumerazione della tipologia è progettata per estensioni future (es. trasferimenti, investimenti). L'immutabilità garantisce consistenza e tracciabilità.

**Tag Gerarchici** Il sistema di etichettatura adotta una struttura ad albero. Ogni **Tag** mantiene un riferimento al proprio padre, consentendo:

- aggregazioni per categorie e sottocategorie;



- navigazione verso l'alto (*parent*) e verso il basso (via *query*);

**Intervalli Temporali** La classe `Period` modella intervalli tramite date di inizio/fine opzionali. I metodi ausiliari garantiscono:

- fallback sulla data corrente se non specificata;
- flessibilità per intervalli aperti;
- semplicità d'uso nel codice client.

## 5.2 Architettura della Persistenza

Il sistema va a separare logica di dominio e logica di storage.

**Astrazione della Persistenza** L'interfaccia generica `FinancialEntitiesPersistency<T>` definisce le operazioni base di `load()` e `save()`. È estesa da:

- `TransactionPersistency`, che aggiunge `loadTags()` per gestire tag separatamente.

**Persistenza XML** L'implementazione `XmlPersistency` si basa su file XML e API DOM. I vantaggi includono:

- leggibilità umana;
- struttura adatta a dati gerarchici;
- robustezza nel parsing.

Due file separati gestiscono transazioni e tag, data la loro diversa natura e frequenza d'accesso.

### Serializzazione e Deserializzazione

- Mappatura diretta tra attributi Java e XML.
- Gestione di file assenti o corrotti senza interruzione del sistema.
- Creazione automatica dei file all'occorrenza.

## Chapter 6

# Meccanismi di Estendibilità

Il sistema **JBudget** è stato concepito fin dalle prime fasi progettuali per favorire l'estendibilità, adottando pattern architetturali che facilitano l'integrazione di nuove funzionalità senza intaccare la stabilità del nucleo esistente.

### 6.1 Estendibilità del Sistema di Persistenza

Un aspetto fondamentale dell'estendibilità riguarda la possibilità di sostituire completamente il backend di storage senza influire sulle componenti applicative.

#### **Intercambiabilità dei Backend di Storage**

Grazie alle interfacce `FinancialEntitiesPersistency` e `TransactionPersistency`, la logica di business è completamente isolata dai dettagli implementativi della persistenza. È quindi possibile rimpiazzare l'implementazione basata su XML con:

- database relazionali (PostgreSQL, MySQL),
- database NoSQL (MongoDB, CouchDB),
- sistemi cloud (AWS S3, Google Cloud Storage).

La transizione tra backend richiede solo la sostituzione dell'implementazione concreta e una modifica alla configurazione tramite *dependency injection*, senza necessità di riscrivere la logica applicativa.

#### **Supporto per Storage Ibridi**

L'architettura supporta anche l'integrazione di più sistemi di storage. Ad esempio, le transazioni potrebbero risiedere in un database relazionale, mentre i tag in un altro tipo di sistema.

## 6.2 Estendibilità del Bilancio

La classe astratta `AbstractBalanceCalculator` fornisce un punto di estensione robusto per nuovi meccanismi di calcolo finanziario.

### Nuovi Tipi di Entità Finanziarie

Tramite l'uso dei *generics*, è possibile implementare calcolatori per diversi domini:

- `DonationBalance`,
- `InvoiceBalance`,
- `LoanAmortizationBalance`,
- `InvestmentBalance`.

Queste implementazioni mantengono coerenza strutturale e integrabilità con il resto del sistema.

## 6.3 Estendibilità dell'Interfaccia `IFinancialEntitiesAdder`

Questa interfaccia astratta permette di gestire l'aggiunta di nuove entità finanziarie. Implementazioni possibili potrebbero essere:

- `BudgetAdder`,
- `GoalAdder`,
- `SubscriptionAdder`.

Ogni entità può implementare logiche di validazione e inserimento specifiche, mantenendo compatibilità con l'infrastruttura esistente.

## 6.4 Indipendenza dall'Interfaccia Utente

Il sistema è progettato per essere completamente disaccoppiato dalla tecnologia di presentazione, permettendo flessibilità nell'adozione di diverse UI.

Ne è una prova la possibilità di poter rendere l'applicazione multi-platform. Il backend può essere consumato da:

- applicazioni desktop (`JavaFX`, `Swing`, `Electron`),
- applicazioni web (via `REST`, `GraphQL`),
- applicazioni mobile (native o con `Flutter`, `React Native`).

## 6.5 Estendibilità del Sistema di Filtraggio

Il sistema di filtraggio è basato su interfacce e pattern compositivi. L'interfaccia `Filtered` può essere estesa per criteri aggiuntivi:

- filtraggio per importo (soglia),
- filtraggio per Categorie (Tag),
- filtraggio per metodo di pagamento.

## 6.6 Estendibilità del Modello di Dati

Il modello dati è stato progettato per evolversi nel tempo. È quindi possibile anche applicare un'estensione delle varie entità. L'uso di `record` Java facilita l'aggiunta di campi opzionali o versioni estese.

Sarebbe anche possibile aggiungere nuovi tipi di transazione grazie all'enumerazione `TransactionType` può essere estesa con:

- `TRANSFER`,
- `INVESTMENT`,
- `REFUND`.

e qualsiasi altra tipologia possa venire in mente.

## 6.7 Estendibilità per la sincronizzazione

L'architettura attuale del sistema `JBudget`, pur essendo progettata per scenari single-user, presenta caratteristiche strutturali che facilitano l'evoluzione verso un sistema multi-utente, con capacità di sincronizzazione e gestione concorrente.

### 6.7.1 Introduzione del Sistema di Account e Autenticazione

Il primo passo verso un sistema multi-utente consiste nell'introduzione di un sistema di account. La separazione delle responsabilità e l'uso di interfacce come cuore del progetto permettono questa evoluzione in modo incrementale e modulare.

### 6.7.2 Modello di Entità Utente

Come prima cosa, per un'eventuale implementazione multi utente, c'è da introdurre il concetto di Utente. L'introduzione di un'entità utente richiede l'estensione del modello dati con nuove classi **record**, mantenendo l'immutabilità e la coerenza con lo stile esistente.

Dovrebbe essere implementato qualcosa di questo tipo:

```
public record User(  
    long userId ,  
    String username ,  
    String email ,  
    String passwordHash ,  
    LocalDateTime createdAt ,  
    LocalDateTime lastLoginAt ,  
) {}
```

### 6.7.3 Architettura di Autenticazione

Bisognerebbe anche implementare un sistema che vada a gestire i vari utenti. Un sistema di autenticazione flessibile può essere progettato mediante l'uso di un'interfaccia **AuthenticationProvider** con più implementazioni concrete, come ad esempio:

- **LocalAuthenticationProvider**: autenticazione su database locale;
- **OAuthProvider**: integrazione con provider esterni (es. Google, Microsoft);
- **JWTAuthenticationProvider**: autenticazione basata su token.

Per mantenere la coerenza architetturale, si propone la creazione di una nuova interfaccia dedicata alla persistenza degli utenti, che potrebbe essere costruita in questo modo:

Listing 6.1: Interfaccia UserPersistency

```
public interface UserPersistency extends FinancialEntitiesPersistency<User>  
    Optional<User> findByUsername(String username);  
    Optional<User> findByEmail(String email);  
    boolean existsByUsername(String username);  
    void updateLastLogin(long userId , LocalDateTime loginTime);  
}
```

### 6.7.4 Isolamento dei Dati per Utente

L'estensione multi-utente richiede l'isolamento dei dati a livello di persistenza. L'attuale implementazione `XmlPersistency` può essere estesa in due direzioni:

- gestione di file XML separati per utente (`userId.xml`);
- struttura XML unica con nodi principali separati per utente.

Per supportare questi scenari, le interfacce esistenti possono essere estese includendo un contesto utente, ad esempio:

```
public interface TransactionPersistency extends ... {  
    List<Transaction> loadByUser(long userId);  
    List<Tag> loadTagsByUser(long userId);  
    void saveForUser(Transaction transaction, long userId);  
}
```

Queste estensioni mantengono la logica applicativa invariata e consentono un'evoluzione fluida verso un sistema multi-utente robusto e sicuro, con isolamento dei dati e possibilità di personalizzazione per ogni profilo.

## Chapter 7

# Conclusioni

Il progetto **JBudget** è nato con l'obiettivo di creare un sistema semplice ma potente per gestire in modo efficace le proprie finanze. Fin dall'inizio si è scelto di puntare su un'architettura chiara, modulare e facilmente estendibile, così da poter aggiungere nuove funzionalità nel tempo senza dover stravolgere il codice esistente. Grazie alla separazione tra logica applicativa e persistenza dei dati, all'uso di interfacce ben definite e a un modello dati flessibile, il sistema è pronto per evolvere: dalla gestione multi-utente fino all'integrazione con servizi cloud. In sintesi, JBudget è un progetto solido, pensato per crescere e adattarsi, mantenendo al centro semplicità d'uso e chiarezza strutturale.