UNIVERSITÀ DEGLI STUDI DI FIRENZE
Dipartimento di Matematica e Informatica

Metodologie di Programmazione

Modellazione di Gestione dei Prestiti in Banca

Docente:

Prof. Lorenzo Bettini

Autore:

Jacob Angeles
matricola: 7024541 | jacob.angeles@stud.unifi.it

Descrizione del progetto

L'obiettivo del progetto è di implementare un gestore di prestiti in banca, in particolare la modellazione delle entità che ne gestiscono. Le entità, in questo caso sono gli agenti e i loro manager, sono a strutture ad albero. Ogn'uno delle entità possono autorizzare un prestito in base ai seguenti criteri:

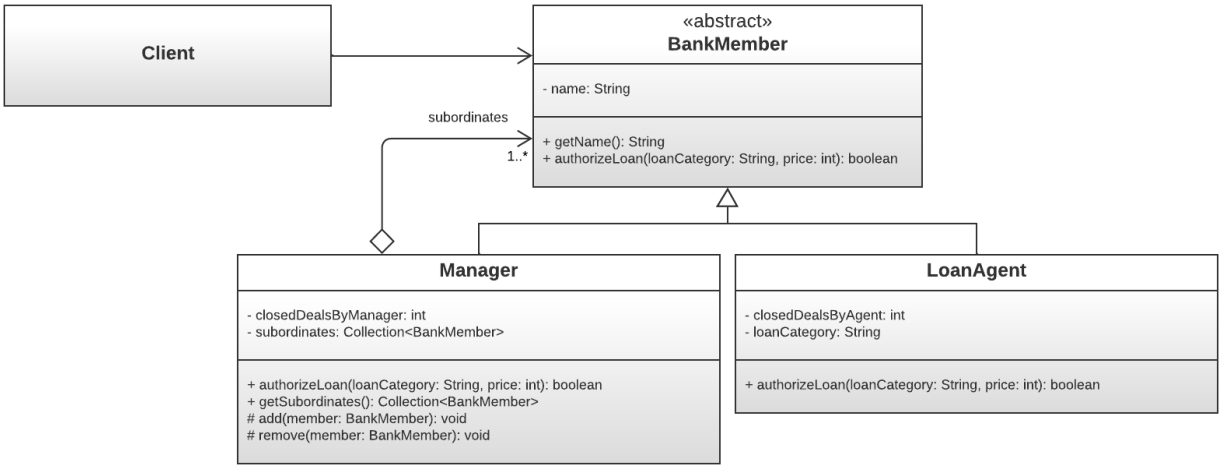
- Gli ***agenti*** possono autorizzare un prestito non superiore a 10.000 e a seconda della categoria a loro assegnate.
- I ***manager*** possono autorizzare tra 10.001 e 200.000 compresi, indifferente dalla categoria del prestito.

Infine, si modellano delle reportistiche per calcolare gli stipendi di ogni personale e i prestiti che hanno autorizzato.

Scelte di Design

Composite

Per la struttura delle entità, sapendo che è ad albero, si è scelto ad utilizzare il pattern ***Composite***, un pattern strutturale, che permette di trattare allo stesso modo oggetti interi e i loro componenti. La variante scelta è quella denominata "***Design for type safety***" per assicurare che le operazioni sui figli(*add, remove, etc.)* fanno parte solo dalle classe composite(*Manager*) e non alla classe astratto(*BankMember*), evitando così che anche le foglie(*LoanAgent*) fossero obbligate ad implementare questi metodi.

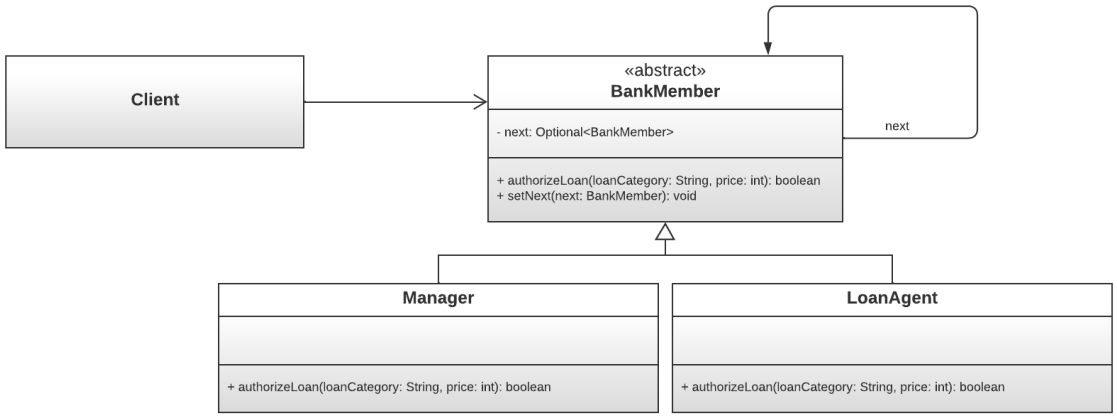


Il **component** in questo caso è la **BankMember**, che è una classe astratta che caratterizza il comportamento comune a tutte le sue sottoclassi. La classe **Manager** è il **composite** che esegue l'operazione di default e gestisce il comportamento dei componenti figli(le foglie e eventuali sotto

composte), come i metodi *getSubordinate*, *add* e *remove*. Il **leap** invece è la classe **LoanAgent** che rappresenta la foglia della struttura ad albero, e fa l'operazione di default.

Chain of Responsibility

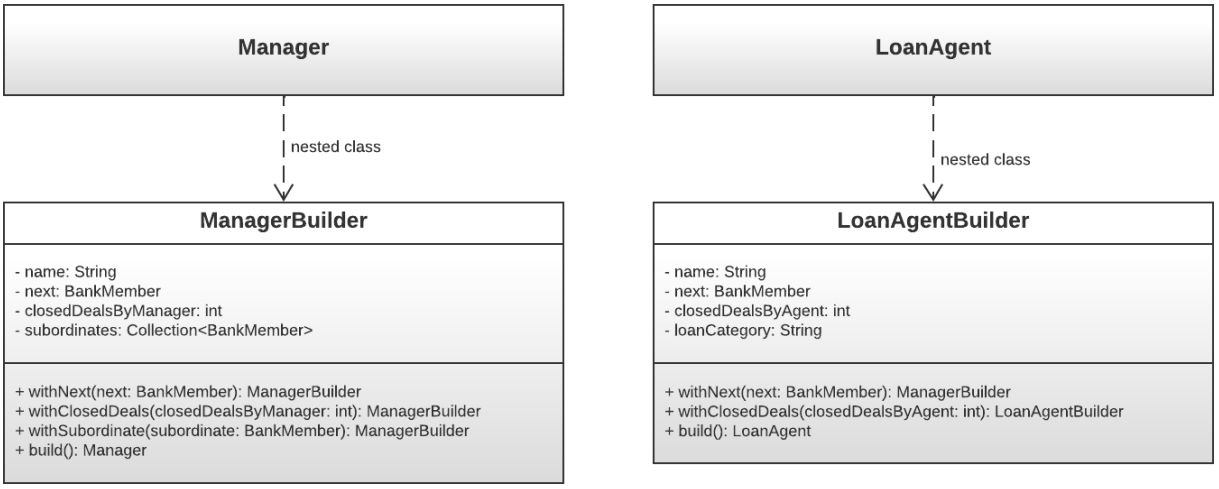
Per l'operazione per gestire i prestiti, chiamata *authorizeLoan*, si è pensato di utilizzare il pattern **Chain of Responsibility**, un pattern comportamentale basato su oggetti, in modo da disaccoppiare il mittente di una richiesta dal destinatario. Il client farà la richiesta alla classe astratta **BankMember** che ne gestirà chi effettuerà operazione tra gli oggetti(*tra Manager e LoanAgent*) nella catena. Durante l'esecuzione del metodo *authorizeLoan*, *BankMember* chiamerà il prossimo elemento della catena tramite il campo *next*, che a sua volta proverà ad effettuare l'operazione, e cosi via finche l'operazione venga fatta oppure che non siano più degli elementi nella catena.



Builder

Per la creazione delle entità, in particolare per *Manager* e *LoanAgent*, si è scelto ad utilizzare il pattern **Builder**, con l'obiettivo di permettere la costruzione di un oggetto utilizzando una *classe helper* invece che con i soliti costruttori. Questo migliora sia la sicurezza durante il processo di costruzione che la leggibilità del codice del programma. Per queste classi, i costruttori sono resi privati, evitando il loro uso.

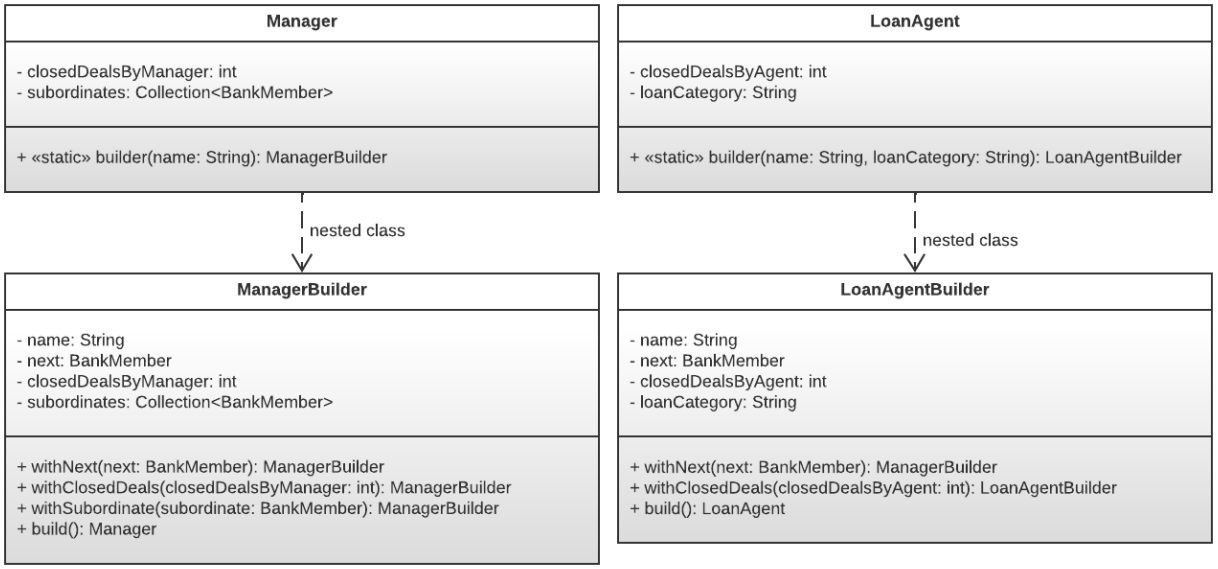
I campi che caratterizzano le classi, Manager e LoanAgent, vengono prima salvati all'interno della classe helper, chiamato ManagerBuilder e LoanAgentBuilder. La classe helper ne verificherà se sono validi e creerà l'oggetto usando il metodo *build*.



Ho sfruttato il fatto che i builder possono validare i campi prima della creazione dell'oggetto per assicurare che il campo ***next*** è sempre gestito, sia con oggetto di tipo BankMember oppure con un null. Dentro la classe helper, viene inizializzato il campo next con un null. Con il metodo interno ***withNext*** invece, il client può decidere di inserire un elemento successivo, sovrascrivendo il valore di default del campo.

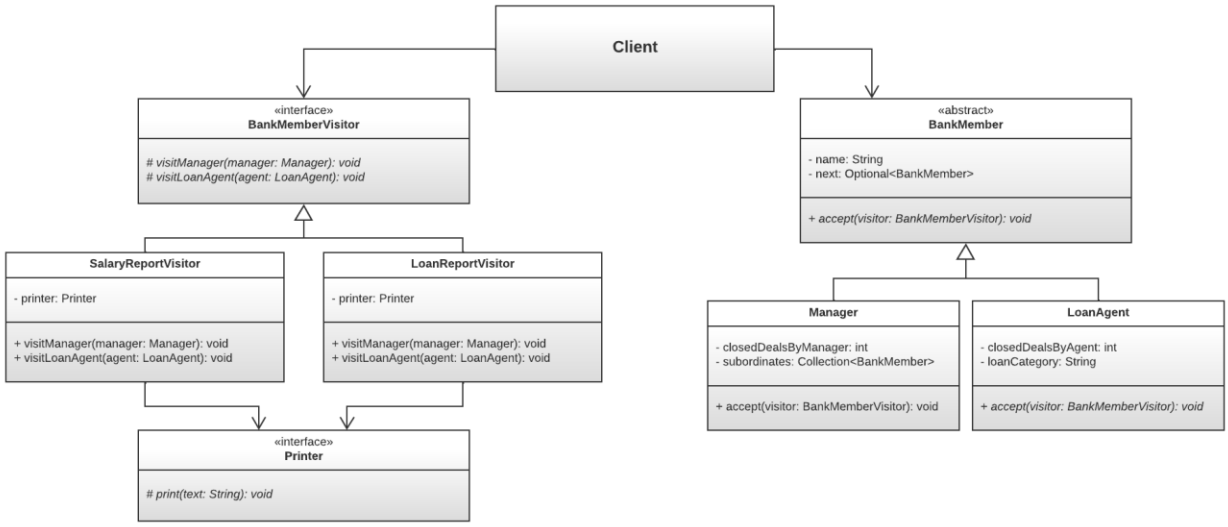
Static Factory Method

Insieme al pattern builder, si è scelto di utilizzare il pattern Static Factory Method per evitare l'uso dei costruttori. Questo permette ancora di avere un codice più pulito e leggibile. Invece di chiamare il costruttore della classe principale(Manager e LoanAgent) e il costruttore degli eventuali classi helper(ManagerBuilder e LoanAgentBuilder), si utilizzerà il *metodo statico* "***builder***" per creare gli oggetti.



Visitor

Per inserire nuove operazioni(in questo caso, per creare report degli stipendi e dei prestiti chiusi) in strutture di oggetti preesistenti, senza che queste strutture debbano essere modificate per questo, si usa il pattern Visitor, che è un pattern comportamentale. Questo mi permetterà di aggiungere in futuro altre operazioni su questi oggetti senza dover modificare il codice ma solo estenderlo(rispettando così l' "*Open Close Principle*" che raccomanda di preferire l'estensione del codice alla sua modifica diretta). In particolare, uso la versione "generica" del Visitor, ovvero non "impongo" un tipo di ritorno specifico ai metodi della classe Visitor ma uso i tipi generici per lasciare libertà di scelta del tipo di ritorno al Client. Questo permetterà di avere un'unica interfaccia per tutti i tipi di ritorno possibili invece di dover creare una nuova interfaccia (e di conseguenza anche un nuovo metodo accept)per ogni tipo di ritorno necessario.



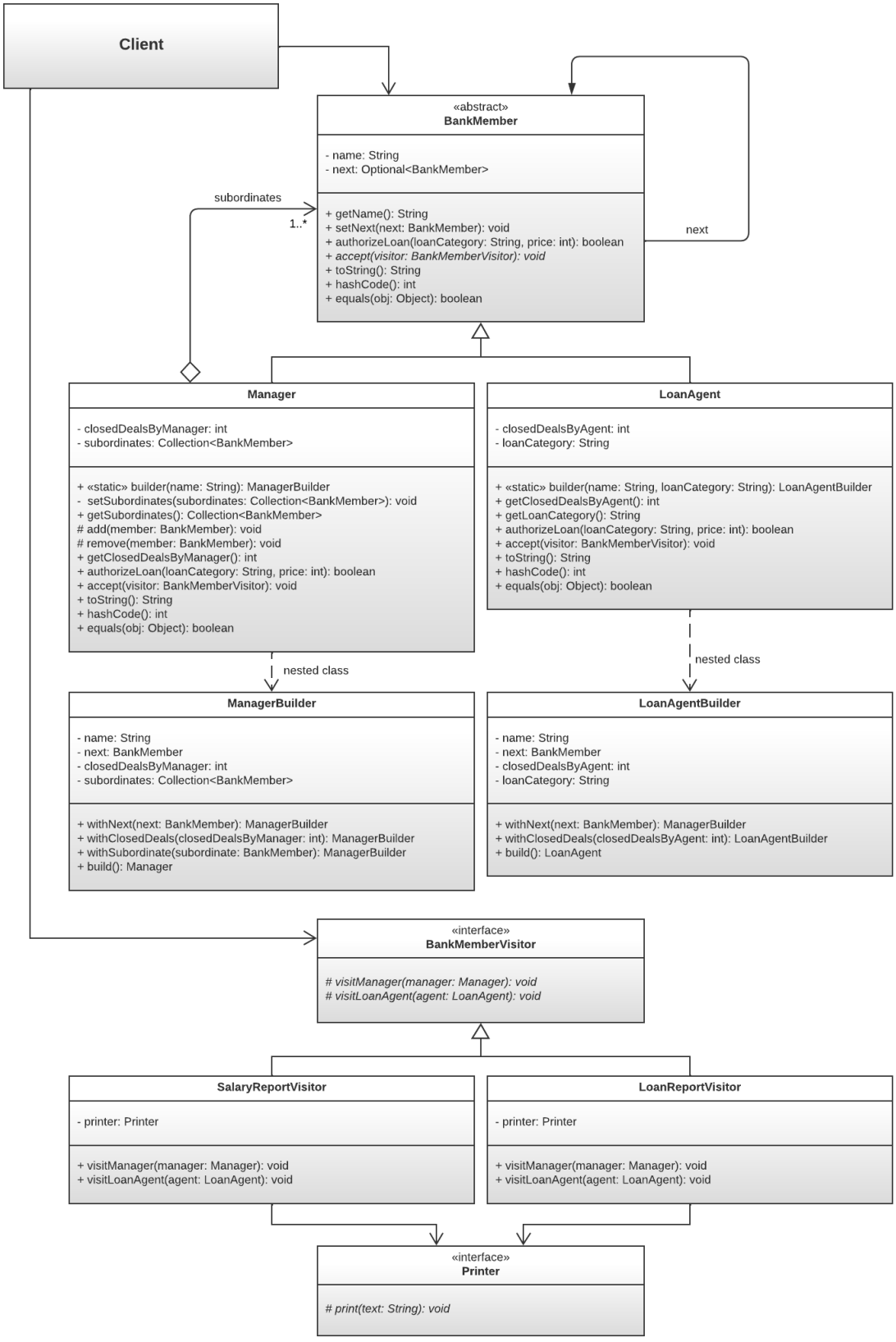
Strategie per testare

Per testare le mie classi concrete ho usato la libreria "AssertJ" in quanto grazie alla sua interfaccia fluente risulta di più facile comprensione ed utilizzo rispetto ai metodi standard di Junit che al contrario risultano, in alcuni casi, di difficile interpretazione e comprensione.

Per le classi Manager, LoanAgent e gli eventuali builder, ho testato semplicemente tramite i metodi verificando se i risultati sono uguali a quelli desiderati. Per testare il Builder della classe LoanAgent ho testato i casi in cui se vengono istanziati tutti i campi(sia quelli neccessari per istanziare l'oggetto che quelli opzionali) o vengono istanziati solo i campi richiesti e non quelli opzionali. Ho testato i casi di eventuali errori sui parametri, catturandone le eccessioni.

Ho creato un file di test separato per testare i comportamenti vari del pattern chain of responsibility, cercando di verificare tutti i possibili casi.

Per testare le classi coinvolti al pattern visitor, ho creato una classe "mock" per le implementazioni fittizie dell'interfaccia Printer. Ho verificato tutti i possibili scenari, confrontando i valori attesi al valore dei risultati. Non ho testato i metodi generato dall'IDE.


Client
«abstract»
BankMember
- name: String
- next: Optional<BankMember>
+ getName(): String
+ setNext(next: BankMember): void
+ authorizeLoan(loanCategory: String, price: int): boolean
+ accept(visitor: BankMemberVisitor): void
+ toString(): String
+ hashCode(): int
+ equals(obj: Object): boolean
subordinates
1..*
next
Manager
- closedDealsByManager: int
- subordinates: Collection<BankMember>
+ «static» builder(name: String): ManagerBuilder
- setSubordinates(subordinates: Collection<BankMember>): void
+ getSubordinates(): Collection<BankMember>
# add(member: BankMember): void
# remove(member: BankMember): void
+ getClosedDealsByManager(): int
+ authorizeLoan(loanCategory: String, price: int): boolean
+ accept(visitor: BankMemberVisitor): void
+ toString(): String
+ hashCode(): int
+ equals(obj: Object): boolean
LoanAgent
- closedDealsByAgent: int
- loanCategory: String
+ «static» builder(name: String, loanCategory: String): LoanAgentBuilder
+ getClosedDealsByAgent(): int
+ getLoanCategory(): String
+ authorizeLoan(loanCategory: String, price: int): boolean
+ accept(visitor: BankMemberVisitor): void
+ toString(): String
+ hashCode(): int
+ equals(obj: Object): boolean
nested class
nested class
ManagerBuilder
- name: String
- next: BankMember
- closedDealsByManager: int
- subordinates: Collection<BankMember>
+ withNext(next: BankMember): ManagerBuilder
+ withClosedDeals(closedDealsByManager: int): ManagerBuilder
+ withSubordinate(subordinate: BankMember): ManagerBuilder
+ build(): Manager
LoanAgentBuilder
- name: String
- next: BankMember
- closedDealsByAgent: int
- loanCategory: String
+ withNext(next: BankMember): ManagerBuilder
+ withClosedDeals(closedDealsByAgent: int): LoanAgentBuilder
+ build(): LoanAgent
«interface»
BankMemberVisitor
visitManager(manager: Manager): void
visitLoanAgent(agent: LoanAgent): void
SalaryReportVisitor
- printer: Printer
+ visitManager(manager: Manager): void
+ visitLoanAgent(agent: LoanAgent): void
LoanReportVisitor
- printer: Printer
+ visitManager(manager: Manager): void
+ visitLoanAgent(agent: LoanAgent): void
«interface»
Printer
print(text: String): void