Nome:Mirko

Cognome:Agresti

Matricola:6360094

Relazione progetto metodologie di programmazione 2019-2020

Specifica del problema:

Per questo progetto ho deciso di realizzare un sistema software che calcola la busta paga per i dipendenti di un’azienda e poi ne gestisce il pagamento con diversi metodi di pagamento.

Analisi del problema:

Per risolvere il problema del calcolo della busta paga sono necessari quattro componenti:

- un lavoratore che può essere di diversi tipi(Es. Lavoratore che riceve un salario mensile, lavoratore a provvigione)
- le ore lavorate o la quantità di prodotti venduti
- il costo orario o la percentuale sui prodotti venduti
- le tasse da dedurre dallo stipendio lordo

Per quanto riguarda, invece, il pagamento dello stipendio è necessario :

- stabilire il metodo di pagamento preferito dal lavoratore
- pagare

Dopo aver analizzato il problema ho pensato di sfruttare il pattern visitor per il calcolo dello stipendio netto, il pattern strategy per ottenere la percentuale di tasse da pagare sullo stipendio, che cambiano in base al tipo di lavoratore, e il builder pattern per istanziare i lavoratori. La parte relativa al pagamento è stata implementata tramite il pattern strategy.

Analisi delle scelte implementative

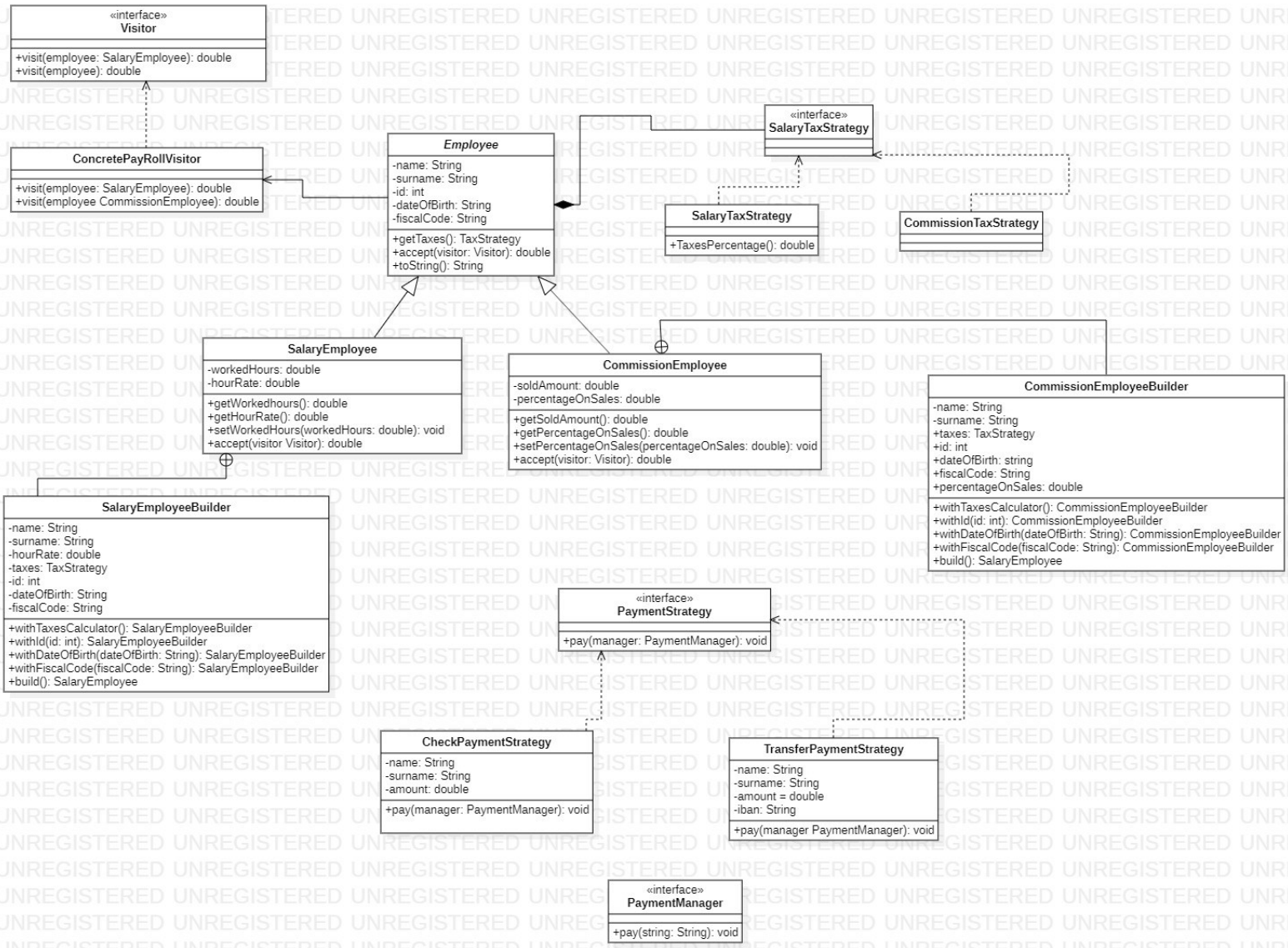
Essendo Employee e le sue sottoclassi la rappresentazione di un lavoratore sono necessari molti campi e quindi un costruttore che richiede molti parametri.

Pertanto ho deciso di implementare all'interno delle classi SalaryEmployee e CommissionEmployee il pattern builder, nella versione descritta da Joshua Bloch, il quale facendo uso di fluent interface permette di migliorare la leggibilità sia durante la lettura del codice che durante la creazione di oggetti. Ho deciso di lasciare fuori dal costruttore l'assegnazione delle ore lavorate dal dipendente in quanto mi pareva più coerente creare prima l'istanza di un lavoratore e poi passare le ore da questo lavorate nel momento in cui si vuole calcolare lo stipendio.

Per calcolare lo stipendio del lavoratore ho deciso di utilizzare il pattern visitor il quale permette di delegare ad una classe esterna l'operazione. Questa scelta porta come benefici l'avere una classe che si limita a rappresentare il lavoratore, la cui struttura non ha bisogno di cambiare spesso, mentre le operazioni, essendo svolte al di fuori della classe, possono essere modificate, aggiunte o rimosse con facilità. Tuttavia questo approccio porta a violare l'incapsulamento in quanto è necessario creare dei getter pubblici e rende la classe Employee rigida.

La percentuale di tasse da pagare sullo stipendio netto viene restituita tramite un pattern strategy. Nella classe Employee è presente un campo taxes di tipo TaxStrategy a cui viene assegnato un oggetto che estende implementa l'interfaccia TaxStrategy tramite il costruttore. Utilizzare questo pattern permette di realizzare l'*open-closed principle* visto che in qualsiasi momento è possibile aggiungere nuove tasse o modificare quelle già esistenti.

Il pagamento dello stipendio è stato realizzato utilizzando il pattern strategy, che ancora una volta permette di sfruttare l'*open-closed principle e si* presta particolarmente bene allo scopo in quanto i metodi di pagamento possono cambiare nel tempo, oppure può rendersi necessario aggiornare i metodi di pagamento(Es. Se l'azienda cambia la banca).


«interface»
Visitor
+visit(employee: SalaryEmployee): double
+visit(employee): double
ConcretePayRollVisitor
+visit(employee: SalaryEmployee): double
+visit(employee CommissionEmployee): double
Employee
-name: String
-surname: String
-id: int
-dateOfBirth: String
-fiscalCode: String
+getTaxes(): TaxStrategy
+accept(visitor: Visitor): double
+toString(): String
«interface»
SalaryTaxStrategy
SalaryTaxStrategy
+TaxesPercentage(): double
CommissionTaxStrategy
SalaryEmployee
-workedHours: double
-hourRate: double
+getWorkedhours(): double
+getHourRate(): double
+setWorkedHours(workedHours: double): void
+accept(visitor Visitor): double
CommissionEmployee
-soldAmount: double
-percentageOnSales: double
+getSoldAmount(): double
+getPercentageOnSales(): double
+setPercentageOnSales(percentageOnSales: double): void
+accept(visitor: Visitor): double
CommissionEmployeeBuilder
-name: String
-surname: String
+taxes: TaxStrategy
+id: int
+dateOfBirth: string
+fiscalCode: String
+percentageOnSales: double
+withTaxesCalculator(): CommissionEmployeeBuilder
+withId(id: int): CommissionEmployeeBuilder
+withDateOfBirth(dateOfBirth: String): CommissionEmployeeBuilder
+withFiscalCode(fiscalCode: String): CommissionEmployeeBuilder
+build(): SalaryEmployee
SalaryEmployeeBuilder
-name: String
-surname: String
-hourRate: double
-taxes: TaxStrategy
-id: int
-dateOfBirth: String
-fiscalCode: String
+withTaxesCalculator(): SalaryEmployeeBuilder
+withId(id: int): SalaryEmployeeBuilder
+withDateOfBirth(dateOfBirth: String): SalaryEmployeeBuilder
+withFiscalCode(fiscalCode: String): SalaryEmployeeBuilder
+build(): SalaryEmployee
«interface»
PaymentStrategy
+pay(manager: PaymentManager): void
CheckPaymentStrategy
-name: String
-surname: String
-amount: double
+pay(manager: PaymentManager): void
TransferPaymentStrategy
-name: String
-surname: String
-amount = double
-iban: String
+pay(manager PaymentManager): void
«interface»
PaymentManager
+pay(string: String): void

Analisi delle classi e delle interfacce

Classe Employee

La classe Employee è una classe astratta che rappresenta un lavoratore generico, quindi sono presenti i campi necessari a identificare il lavoratore, i metodi necessari per il visitor che si occupa di calcolare lo stipendio e il metodo toString che serve per stampare le informazioni relative al lavoratore. In questa classe ho cercato di applicare il più possibile il *single responsibility principle* utilizzando il visitor e creando solo campi relativi al lavoratore.

Classe SalaryEmployee

La classe SalaryEmployee estende la classe Employee e aggiunge campi e metodi necessari per il calcolo di una busta paga mensile.In particolare il campo *workedHours con relativi getter e setter* e il campo *hourRate* a cui viene assegnato un valore tramite il costruttore. Ho scelto di passare le ore lavorate tramite un setter in quanto mi sembrava coerente con il funzionamento reale di questi programmi in cui la creazione di un oggetto lavoratore viene eseguita quando si inserisce nell'anagrafica e poi ogni mese si passano le ore lavorate. Il campo *hourRate* viene inizializzato dal costruttore in quanto il costo orario di un lavoratore viene deciso al momento dell'assunzione secondo determinati principi. All'interno della classe employee è presente una classe annidata. Questa classe è il builder che si occupa di creare oggetti di tipo SalaryEmployee. È stato necessario usare questo pattern a causa della natura di Employee e delle sue sottoclassi, infatti sfrutando una fluent interface è stato possibile rendere più facile la lettura del codice, che appare quasi discorsivo, ed è stato anche possibile rendere più facile la creazione di oggeti. Infatti essendoci molti parametri sarebbe potuto diventare difficile inserire i parametri giusti.Infine in questa classe è presente il metodo *accept* che prendendo come parametro un visitor permette che un oggetto venga visitato dal visitor stesso disaccoppiando il calcolo dalla classe.

Classe CommissionEmployee

La classe CommissionEmployee è molto simile alla classe SalaryEmployee, anche questa estende la classe astratta Employee aggiungendo i campi *soldAmount, con i getter e setter* e *percentageOnSales con il suo getter.* La logica di questa classe è la medesima di SalaryEmployee. Le uniche differenze stanno nel tipo di lavoratore che viene rappresentato.

Interfaccia TaxStrategy

L’interfaccia tax strategy fa parte delle classi e interfacce che realizzano il pattern strategy, dichiara l’operazione necessaria per restituire la percentuale di tasse da pagare. Un campo di questo tipo è presente in tutti gli oggetti di tipo Employee per stabilire le tasse da pagare.

Classe SalaryTaxStrategy

Questa classse è una delle implementazioni dell’interfaccia TaxStrategy e rappresenta l’aliquota da pagare nel caso di un lavoratore dipendente con uno stipendio mensile calcolato sulle ore lavorate.

Classe CommissionTaxStrategy

Questa classe è l’altra implementazione dell’interfaccia TaxStrategy e rappresenta l’aliquota da pagare nel caso di un lavoratore a provvigione.

Interfaccia Visitor

Questa interfaccia fa parte delle classi e interfacce che realizzano il pattern visitor. In particolare definisce i due metodi visit che prendono rispettivamente un SalaryEmployee e un CommissionEmployee.

Classe ConcretePayRollVisitor

Questa classe implementa l’interfaccia visitor e definisce i due metodi visit, grazie ai quali viene calcolato lo stipendio del lavoratore. Nel metodo che prende come parametro un SalaryEmployee vengono chiamati i metodi getWorkedHours e getHourRate oltre al metodo taxesPercentage dell’oggetto taxes presente nel SalaryEmployee passato come parametro.Questi vengono poi usati per fare il calcolo finale. Similmente il metodo che prende come parametro un CommissionEmployee chiama i metodi getSoldAmount e getPercentageOnSales insieme al metodo taxesPercentage dell’employe passato come parametro.

Classe SalaryEmployeeBuilder

Questa classe serve per creare istanze di SalaryEmployee presenta tutti i campi presenti in presenti in SalaryEmployee in maniera da poterli passare al costruttore.

È qui che viene chiamato il costruttore di SalaryEmployee. La classe è stata realizzata all’interno di un’altra classe in quanto così è possibile usare il costruttore di SalaryEmployee che è privato.

Classe CommissionEmployeeBuilder

Questa è simile a SalaryEmployeeBuilder le uniche differenze stanno nel tipo di campi presenti e nei relativi metodi che sono anche loro diversi.

Interfaccia PaymentStrategy

Questa interfaccia appartiene al gruppo di classi/interfacce che implementano il pagamento dello stipendio. In essa compare il metodo *pay()* che viene poi implementato in concreto dalle classi CheckPaymentStrategy e TransferPaymentStrategy.

Classe CheckPaymentStrategy

Questa classe estende l'interfaccia PaymentStrategy e modella il pagamento tramite assegno e quindi contiene i campi necessari per effettuare un assegno quali nome, cognome, e cifra da pagare. Il metodo *pay()* ereditato non è stato implementato concretamente ma è accessibile da un'astrazione a cui viene delegata la gestione del pagamento.

Classe TransferPaymentStrategy

Questa classe estende l'interfaccia PaymentStrategy e modella il pagamento attraverso un bonifico, contiene i campi necessari per questo tipo di pagamento ovvero: nome, cognome, cifra da pagare e codice iban. Anche in questo non è stata realizzata un'implementazione concreta del metodo *pay()* al quale si può accedere tramite l'astrazione del PaymentManager a cui si delega per la gestione del pagamento.

Interfaccia PaymentManager

Questa interfaccia rappresenta l'astrazione attraverso cui viene gestito il pagamento degli stipendi.