ARASH ZIAEE - 7029880

Progetto del Corso Metodologie di Programmazione

Descrizione del progetto

Questo progetto modella la struttura di un'azienda di sviluppo software,
Racchiudendo vari ruoli all'interno dell'azienda, concentrandosi in particolare sui membri dello staff: Developers e ProjectManager.
La struttura è progettata per offrire un quadro chiaro ed estensibile per rappresentare i diversi ruoli del personale e le loro interazioni, si basa su una gerarchia a struttura ad albero dei lavoratori all'interno dell'azienda, dove sono state implementate tre categorie principali che sono: Staff, Developers e Project Manager.

Nel progetto vengono implementate tre funzionalità:

1. Calcolo del salario: Implementa un sistema per calcolare il salario di ogni dipendente tenendo conto di diversi fattori come l'esperienza e i bonus per i project manager in base al numero del developer gestiti.
2. Assegnare di progetto: permette di assegnare il progetto ai Project manager con capacità di estendere questa assegnazione ai Developers
3. Dettagli sui membri dello Staff: attraverso l'uso di metodi dedicati il sistema può fornire informazioni dettagliate su ogni membro dello staff incluse anche Developers e ProjectManager

Design Pattern Applicati

I pattern applicati in questo progetto sono:

- Builder Pattern
- Static Factory Method
- Visitor Pattern
- Composite Pattern

Le descrizione di ogni pattern sono riportate come seguito:

1. **Builder** e **Static Factory Method**:

   Poiché questi due pattern possono essere applicati insieme, nella documentazione li spieghiamo insieme.

   Il Builder Pattern e Static Factory Method sono stati implementati come strategie di design per la creazione di oggetti complessi all'interno della struttura aziendale. Queste tecniche sono state scelte per la loro efficacia nel migliorare la leggibilità, la flessibilità e la manutenibilità del codice.

   Abbiamo una classe statica interna sia nella **Classe Developers** che nella **Classe ProjectManager**.

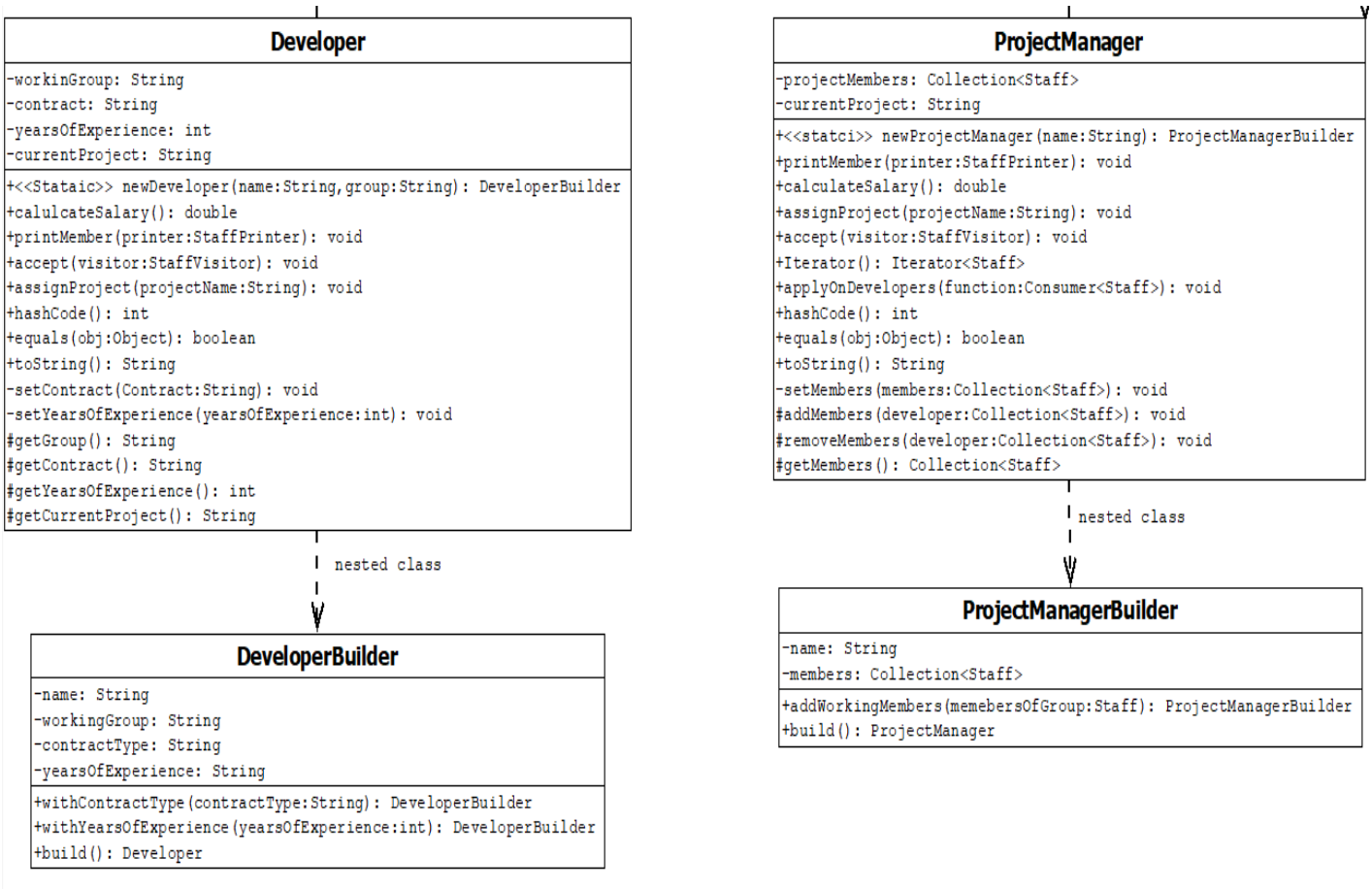
   In queste classi abbiamo il costruttore che prende come input i campi richiesti e pio definiamo i metodi con parametro di input opzionali all'interno della Classe Builder di ciascun componente del progetto, La costruzione si conclude con l'invocazione del metodo ***build()*** che crea l'oggetto utilizzando gli argomenti raccolti per i vari campi.

Un campo come tra le due Classi **Developers** e **ProjectManager** è il campo Name dichiarato nella Superclass **Staff.**

L'oggetto **Developers** contiene un altro campo richiesto ***workingGroup***, all'interno del costruttore della Classe Builder di questa Classe Developer abbiamo il costruttore con due parametri di input ***DeveloperBuilder(name, group)*** E altri due campi opzionali che sono ***(contractType, YearsOfExperience)***, questi due campi vengono istanziati con i metodi all'interno della Classe Builder e abbiamo deciso che quando questi due campi opzionali non vengono istanziati da Client, vengono considerati i valori di default che sono **contractType = "Not Mentioned", YearsOfExperience = "0"**.

Per garantire una creazione degli oggetti robusta e controllata, abbiamo reso il costruttore della classe privato, così il Builder è l'unico modo della creazione di un oggetto.

L'implementazione del Builder Pattern e Static Factory Method in questo progetto ha migliorato l'architettura del codice, rendendo il processo di creazione degli oggetti più intuitivo e mantenibile.L'uso dei setter accanto all'utilizzo del Builder Pattern è una cosa inevitabile, pertanto li rendiamo privati, e accessibili solo all'interno della classe ed alla Classe Builder.



2. **Composite Pattern**:

   Il Pattern Composite viene utilizzato per modellare e gestire la gerarchia e la struttura ad albero tra diversi tipi di oggetti dello staff, in particolare tra ProjectManager e Developers. Questo approccio è ottimale per rappresentare strutture dove i nodi possono avere sia ruoli di “foglia” (come Developers) sia ruoli di “compositi”(come ProjectManager).

   Staff:

Superclasse astratta che definisce l'interfaccia comune per tutti i componenti, sia compositi sia foglia. Fornisce metodi come 'printMember()', 'calculateSalary()', 'assignProject()'che devono essere implementati sia dai **Class Developers** sia dai **Class ProjectManager**.

Developers:

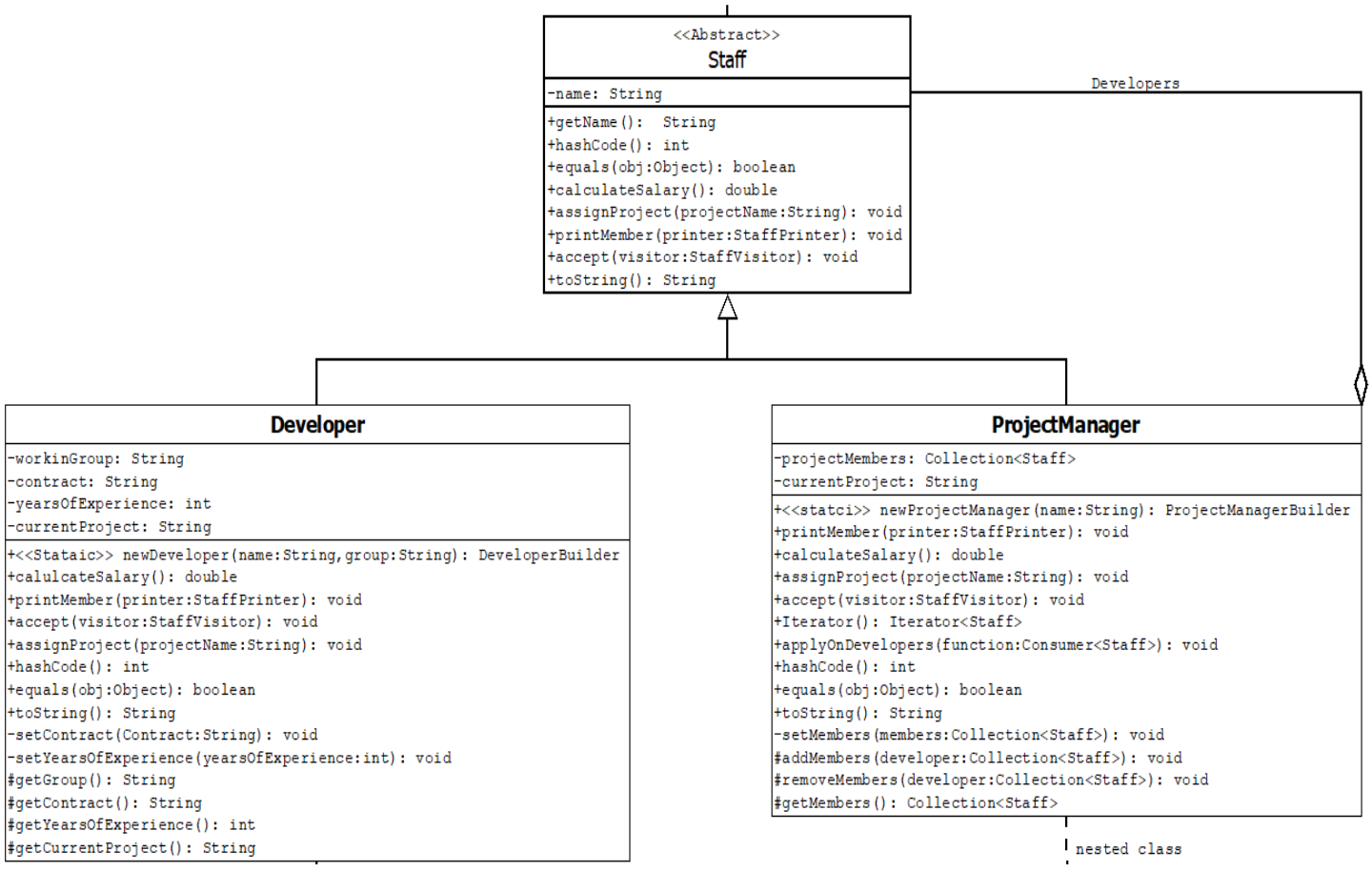
Sono i componenti della foglia. Non hanno sottocomponenti che rappresentano gli elementi base della struttura. In questa Classe Developers, abbiamo un metodo di stampa il nome, e altri dettagli di un Developer prendendo come input un ***printer*** di tipo **StaffPrinter** → Il Dichiariamo questa interfaccia (***StaffPrinter Interface***), solo per delegare la funzionalità di Print e quindi non dichiariamo nessuna classe concreta come la sua implementazione. Solo per poter testare il programma si è creata una classe **MockStaffPrinter** sarebbe una implementazione fittizia.
Esiste un metodo ***calculateSalary()***, contiene la logica del calcolo del salario di una foglia.

ProjectManager:

Agisce come una componente composita. Ogni oggetto ProjectManager può avere sotto di sé altri oggetti Staff, che possono essere sia Developers sia altri ProjectManager. Questo permette di costruire una struttura gerarchia flessibile.
Esiste anche operazione di stampa:
Stampa del Project Manager: Il metodo stampa i dettagli dell'oggetto ProjectManager. Questo è implementato nella prima chiamata di ***printer.print***, dove vengono stampati il nome del project manager e il progetto assegnato.

Membri del Team: Il Metodo stampa anche i dettagli dei membri del team sotto il controllo del ProjectManager. Viene stampato solo il nome di ogni membro del team e non altri dettagli specifici. La chiamata a ***projectMembers.stream()*** seguita da ***.map*** e ***.forEach*** serve per iterare su ogni membro del team, estrarne il nome e stamparlo.

Esiste anche il metodo **calculateSalary()** che ha la sua logica di implementazione diversa dalla Classe Developers.



3. **Visitor Pattern**

Il progetto, che presenta una struttura di oggetti appartenenti a diverse classi con una superclass comune, abbiamo deciso di applicare due funzionalità principali: il calcolo del salario di ogni dipendente e la stampa delle informazioni del gruppo con il progetto assegnato.

Il Visitor Pattern è stato implementato per consentire l'esecuzione di operazioni sugli elementi della struttura oggetti senza la necessità di modificare le lassi di questi elementi. Questo approccio offre la flessibilità di aggiungere nuove operazioni all'albero in futuro, in base alle esigenze e alle richieste dei client. Inoltre, il pattern visitor supporta l'estensibilità del codice e facilità la manutenzione, poiché le modifiche possono essere fatte senza avere nessun danno all'architettura esistente.

Abbiamo dichiarato un'interfaccia comune StaffVisitor con due metodi ***visitDevelopers(), visitProjectManager()*** e anche nella superclass Staff si dichiara un metodo ***accept()*** per predisporre le sottoclassi e essere visitata tramite questo metodo. Siccome i visitor sono applicati come tipo void, ogni visitor concreto mantiene anche uno stato per accumulare i risultati.

implementazione **StaffListVisitor** manteniamo uno stato attraverso un campo di tipo **StaffPrinter**, questo ci permette di accumulare i risultati delle operazioni di stampa eseguite sui vari membri dello staff.

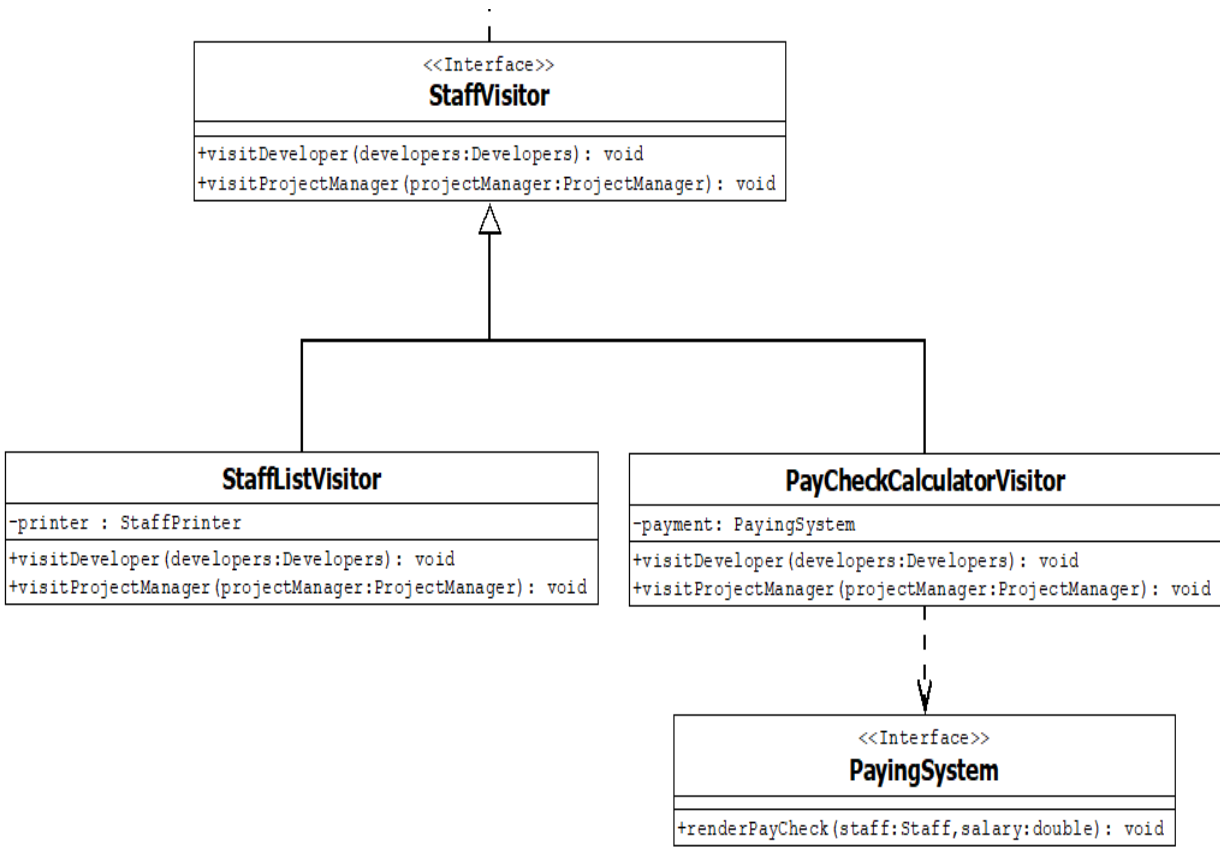
Implementazione **PayCheckCaculatorVisitor**, similmente nella classe concreta PayCheckCalculatorVisitor, utilizziamo un campo di tipo **PayingSystem,** per

accumulare e gestire i calcoli dei vari membri dello staff.

Classi Concreti del Pattern Visitor:

+ ***StaffListVisitor***: ha la responsabilità per la stampa delle informazioni relative ai membri dello staff, inclusi "**Developers**"e sia "**ProjectManager**". Considerando che stiamo usando i visitor void, dobbiamo avere uno stato per poter accumulare i risultati passo per passo e quindi esiste un campo di tipo **StaffPrinter,** come abbiamo visto prima, questa interfaccia è senza implementazione concreta ha solo un metodo ***print().***

+ ***PayCheckCalculaterVisitor***: questa classe concreta si focalizza sul calcolo e la gestione dei salari dei membri dello staff. Si occupa del calcolo di calcolare il salario di ogni dipendente. Per questa operazione abbiamo dichiarato un'interfaccia **PayingSystem**, sempre lasciando la senza implementazione con un unico metodo renderPayCheck() che ci fornisce di delegare l'operazione di generare il salario che viene dopo implementata da un client e di questa classe abbiamo anche un'implementazione fittizia solo per essere in grado di testare la classe.



Ci sono i metodi ***CalculateSalary()***, e ***applyOnDevelopers()***:

CalculateSalary(): è utilizzato nelle classi "**Developers**" e "**ProjectManager**" per calcolare il salario dei membri dello staff, questo metodo è un esempio di come la funzionalità specifiche di un classe possano essere utilizzate per eseguire calcoli interni in modo efficiente.

Sebbene il calcolo del salario in una classe esterna come "**PayCheckCalculator**" sembra una violazione del principio di Single Responsibility Principle, in realtà rientra nell'ambito delle responsabilità specifiche di queste classi.

Il calcolo salario si fa in due modi diversi a seconda della loro classe concreta.

applyOnDevelopers(): è presente nella classe "**ProjectManager**" e consente di applicare un'azione specifica a tutti gli developers o projectManager sotto la supervisione di un determinato project manager. Prendendo un funzione di input, visita tutti i figli e applica questa funzione su tutti questi. Come parametro di input, nella classe **PayCheckCalculatorVisitor** passiamo il metodo **CalculateSalary**.

Strategie per testare

Per verificare la corretta implementazione del programma ci sono le classi dei test tutti scritti utilizzando dei test JUnit la versione Junit4, importando le librerie necessarie per scrivere dei test.

Assicurarsi che ogni classe e metodo funzionino come previsto, conformemente alle specifiche del progetto, garantire che le diverse parti del sistema interagiscono correttamente, rilevare e risolvere problemi.

Strategie Specifiche di Test:

Test del Builder Pattern → Verifica che gli oggetti "Developers" e "ProjectManager" siano creati correttamente attraverso il loro Builder,con tutti i parametri obbligatori e opzionali gestiti in modo appropriato.

Test del Visitor Pattern → Assicura che "StaffListVisitor" e "PayCheckCalculatorVisitor" interagiscono correttamente con gli oggetti "Staff", chiamando i metodi adeguati e producendo l'output e il calcolo previsto.

Test di Funzionalità Ricorsive → Verifica che la funzione “assignProject” in “ProjectManager” assegni correttamente i progetti a tutti i membri del team, inclusi altri project manager e developer sotto la loro supervisione.

Spiegazione di Ogni Classe di test

- **DevelopersTest**

  Durante la fase SetUp, viene creato un nuovo developer utilizzando il Builder Pattern, implementato nella classe “Developer”, questo consente di configurare l'istanza con vari attributi, dimostrando l'efficacia e la flessibilità del pattern per la creazione di oggetti complessi.

  Abbiamo testato varie casi, Creazione con Builder, viene testata la correttezza della creazione di un’istanza “Developers” tramite il Builder Pattern, verificando l’assegnazione corretta di tutti i parametri, inclusi quelli opzionali.

  Creazione senza parametri opzionali, si testa la creazione di un “Developers” senza specificare i parametri opzionali, verificando che i valori default, “Not Mentioned” e “0” vengono correttamente applicati. Questo test evidenza la flessibilità del Builder Pattern nel gestire scenari con parametri mancanti.

  Stampa dettagli, attraverso l’utilizzo del mock “MockStaffPrinter” si verifica l'accuratezza delle rappresentazione testuale di “Developers”, assicurando che tutte le informazioni siano correttamente visualizzate.

Calcolo del salario, Viene testata la logica di calcolo del salario, basata sull'esperienza e altri criteri specifici di "Developers", valida la corretta implementazione della formula di calcolo del salario.|

- **ProjectManagerTest**

  Nella fase di SetUp, viene configurato un ambiente di test che include la creazione di un "ProjectManager" con un team composto da un altro ProjectManager e due Developers, ciascuno con caratteristiche specifiche. Questo setup riflette una struttura gerarchica tipica all'interno del progetto.

  Test il Builder Pattern, verifica che l'oggetto creato tramite il Builder Pattern sia correttamente inizializzato con i parametri forniti, inclusi i membri del team.

  Test ProjectManager print, controlla la corretta rappresentazione testuale del "ProjectManager" e dei membri del suo team, utilizzando un MockStaffPrinter per catturare l'output e confrontarlo con il risultato atteso.

  Test del calcolo del salario, valuta la correttezza del calcolo del salario per il "ProjectManager", che deve includere i bonus basati sul numero di membri nel suo team.

  Test Assegnazione del Project, verifica assegnazione di un progetto al ProjectManager e la corretta propagazione di questa assegnazione a tutti i membri del team.

  Test Iterator Project Manager, testa la capacità di iterare sui membri del

team di un “ProjectManager”, confermando che l’iteratore esponga tutti i membri attesi.

Test di aggiungere e rimuovere membri, questi test verificano rispettivamente la capacità di aggiungere nuovi membri al team di un ProjectManager, e di rimuoverli, validando le modifiche alla composizione del team.

- Diagramma UML del Programma