Progetto Metodologie di Programmazione
Social Network

Info su autore e data di consegna:

Alessandro Grazioli

Numero di matricola: 7012653

E-mail: alessandro.grazioli@stud.unifi.it

Data di consegna: 22/01/2020

Descrizione del sistema implementato:

L'idea alla base del sistema software implementato è la simulazione di alcune funzionalità che sono presenti nei social network odierni.
Andrò ad esporre di seguito le idee che mi hanno portato a scegliere i pattern da utilizzare, lasciando la loro spiegazione dettagliata ai prossimi paragrafi della relazione.

La prima funzionalità che ho implementato è stata quella della creazione di un profilo tramite il pattern Builder. Tale scelta è parsa naturale, poiché nell'effettiva creazione dei profili, nella vita reale, non tutti i campi devono essere obbligatoriamente compilati. Il Builder offre la funzionalità di poter costruire un oggetto passo dopo passo specificando alcuni campi opzionali.

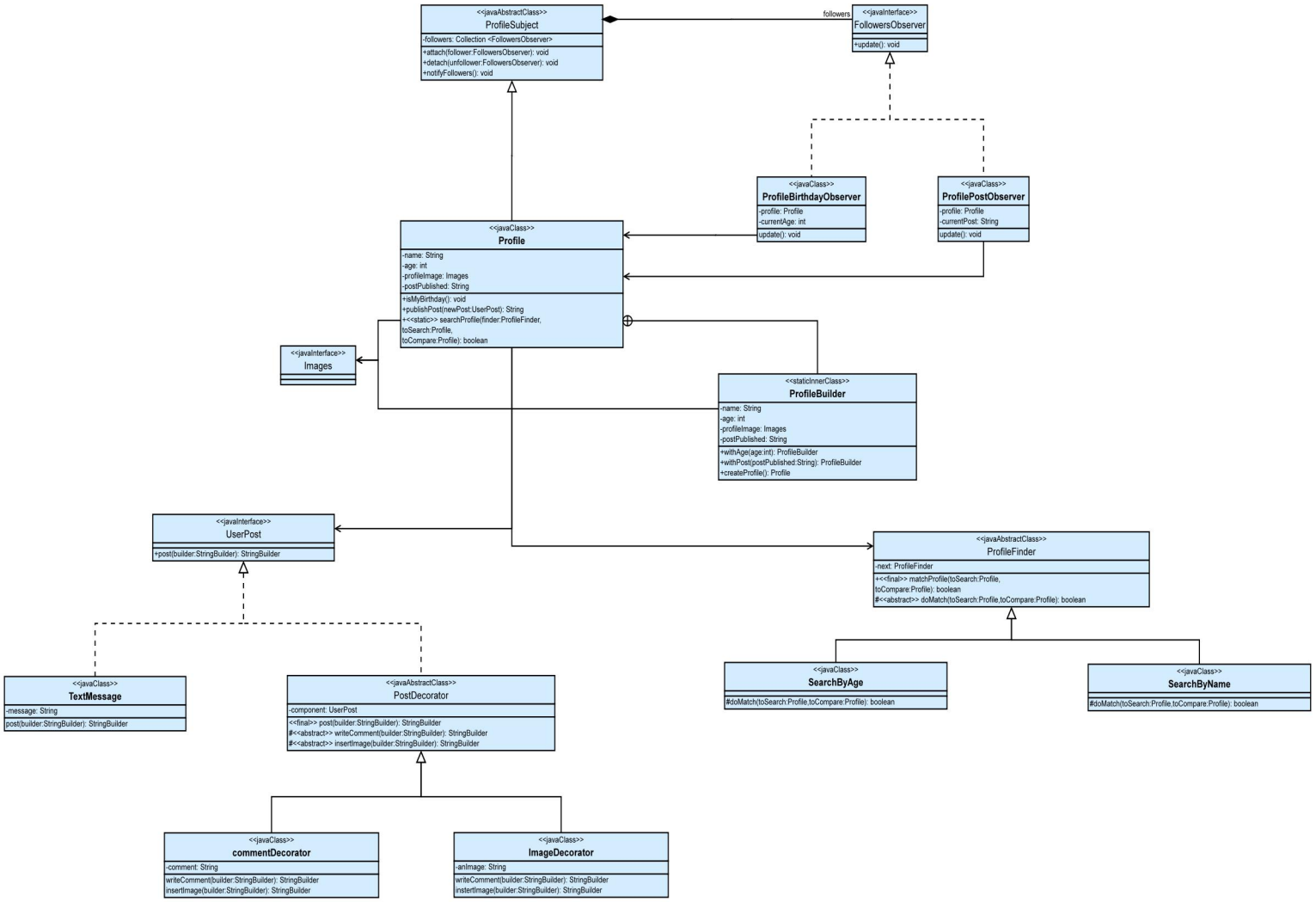
La seconda funzionalità implementata è stata quella della simulazione di un following. Tale funzionalità è stata implementata tramite il pattern Observer, in quanto quest'ultimo permette la notifica al momento del cambiamento di uno stato dell'oggetto osservato. Basandosi su quest'idea, una volta effettuato il follow, i follower (osservatori) del profilo vengono notificati ad ogni nuovo post oppure quando il profilo osservato compie gli anni.

La terza funzionalità implementata è stata quella della simulazione della creazione di un nuovo post. Dopo numerose riflessioni, è stato pensato che potesse essere una buona idea applicare il pattern Decorator per implementare tale funzionalità. Preso come elemento principale il testo di un post, questo viene eventualmente decorato con un'immagine oppure un commento. Essendo che il software simula la creazione di un post, l'immagine è semplicemente un oggetto della classe String.

La quarta ed ultima funzionalità che è stata data al social network, è quella della ricerca di un profilo. L'idea era quella di simulare la ricerca dei profili tramite nome o età. Per implementare tale scelta è stato deciso di utilizzare il pattern Chain of Responsibility. Tale pattern offre la possibilità di delegare a più oggetti la risoluzione di un problema, in questo caso la ricerca, senza che il client sappia da chi è stato gestito tale problema.

Riporto nella pagina seguente il diagramma UML del software Social Network.

STRUTTURA DEL PROGRAMMA ILLUSTRATA TRAMITE DIAGRAMMA UML



Dettagli dell'implementazione:

Il cuore principale del software è racchiuso nel pacchetto **socialNetwork.obeserver.builder** dove risiede la classe concreta **Profile** che gestisce tutte le chiamate ad i metodi implementati. Per questo particolare progetto, essendo destinato ad un uso prettamente universitario, ai fini del superamento dell'esame del corso di Metodologie di Programmazione, non si è fatto utilizzo di nomi di pacchetti secondo la convenzione che vede, come nomi di pacchetti, quelli di un eventuale dominio internet. Per tali fini, dunque, si è racchiuso il codice in diversi pacchetti, cercando di suddividere così i differenti pattern utilizzati e di migliorare la chiarezza della struttura implementata. Durante tutta la scrittura del codice di questo progetto, sono stati tenuti sempre in considerazione i cinque principi S.O.L.I.D. cercando di rispettarli tutti, a meno di conseguenze dovute all'utilizzo di pattern che li violano.

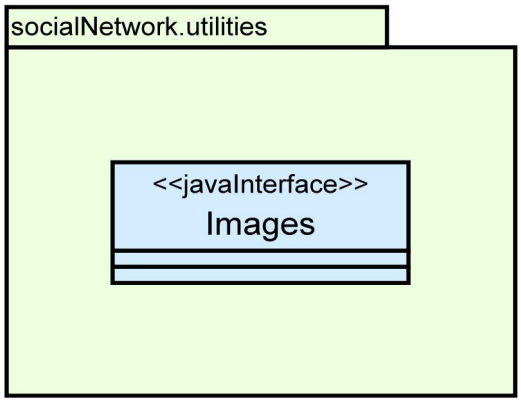
A questo punto, andremo in dettaglio pacchetto per pacchetto, analizzandone i contenuti e esplicando il motivo delle scelte implementative.

socialNetwork.utilities:

Il primo pacchetto che voglio andare ad esplorare, è il pacchetto più piccolo dell'intero software. In pratica si tratta di un semplice pacchetto di utilità dove ho inserito una sola interfaccia che mi serviva per simulare l'immagine del profilo nella classe **Profile.**

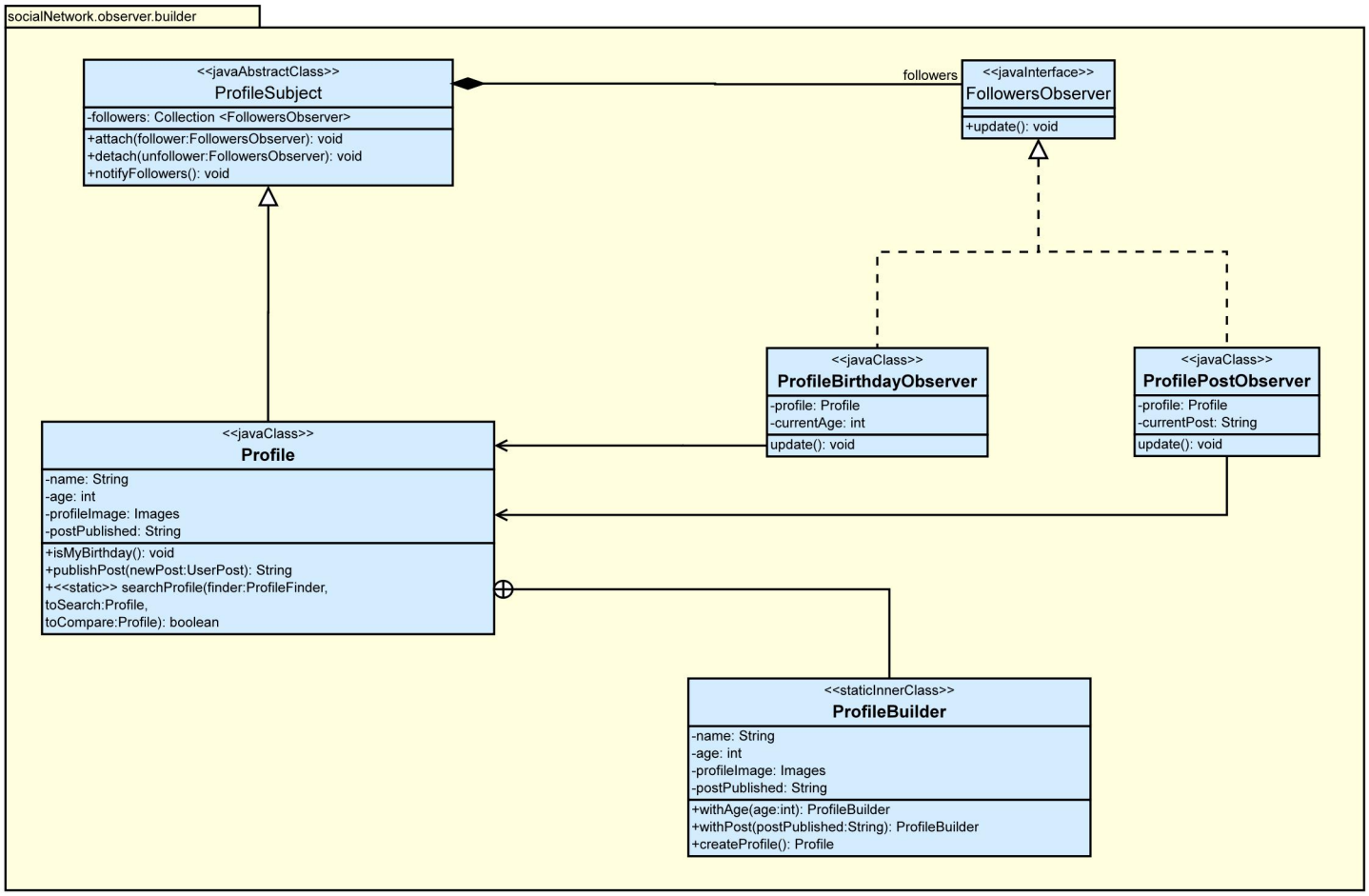
- *Images:*

Interfaccia Java che simula un immagine caricata in un social network. Viene utilizzata dalla classe **Profile** che ne conserva un riferimento in un campo che usa come immagine del profilo. È pensata per essere implementata da futuri sviluppatori. Ai fini dei test è stata implementata una classe **MockImage** nel pacchetto test **socialNetwork.observer.builder.**



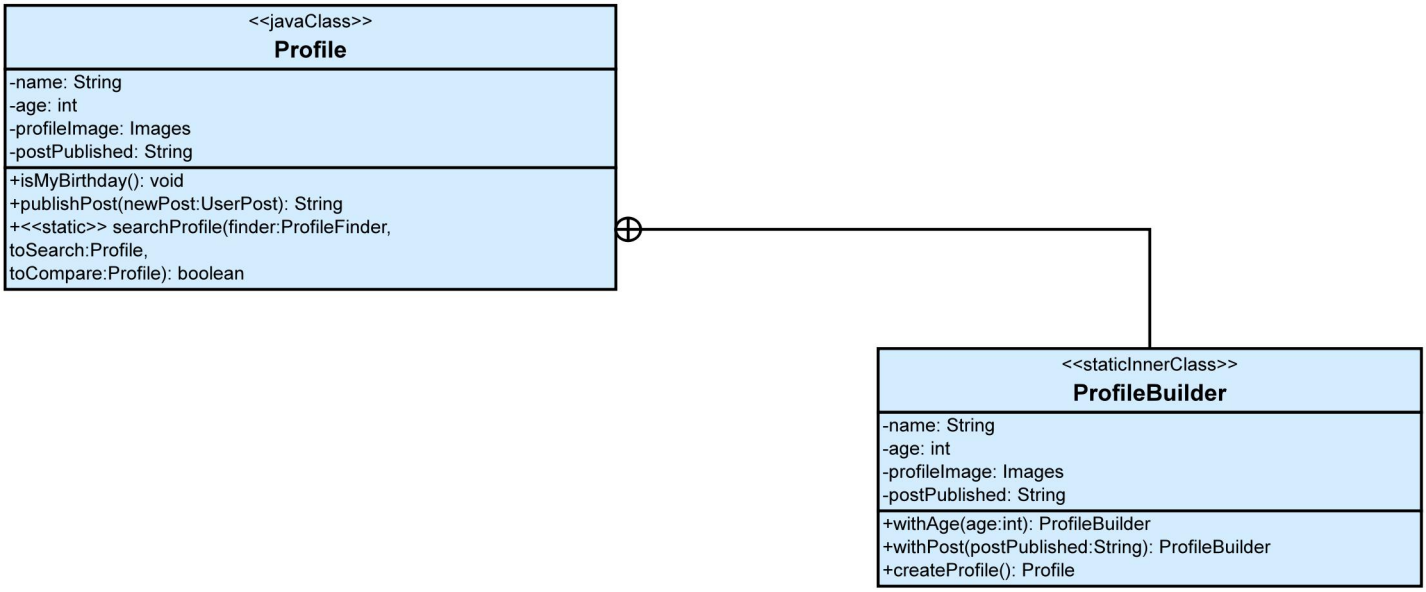
socialNetwork.observer.builder:

Come già detto in precedenza, questo è il pacchetto dove risiede la classe principale di questa modellazione software.
La classe **Profile** è il punto di incontro delle diverse funzionalità di questo progetto. Tale scelta è sembrata sensata, in quanto in un vero social network odierno, nella pagina principale di un profilo, si ha la possibilità di usufruire delle funzionalità messe a disposizione da tale piattaforma. Durante l'implementazione di questa classe, che si arricchiva man mano che nuove funzionalità venivano implementate, ho prestato molta attenzione a non sovraccaricare la classe con responsabilità al di fuori del suo campo, cercando così di rispettare il primo dei cinque principi S.O.L.I.D.
La prima funzionalità implementata in questo pacchetto è quella della creazione del profilo. Per implementare tale funzionalità, non ho avuto dubbi nell'utilizzare il pattern Builder.



<u>Pattern Builder:</u>

Nella classe **Profile** è dunque presente una *static inner class* chiamata per l'appunto **ProfileBuilder.**
Tale classe, è stata implementata seguendo lo schema di implementazione visto a lezione, infatti, nella classe **ProfileBuilder** sono presenti i tutti i campi della classe **Profile,** il cui costruttore è stato dunque dichiarato come *private* e implementato con body vuoto.
Seguendo l'esempio dei veri social network e delle funzionalità offerte dal pattern Builder, due dei quattro campi presenti nella classe **Profile** sono stati resi opzionali, tramite i metodi *withAge()* e *withPost().* Questi due metodi, rendono dunque la creazione di un profilo dinamico, permettendo così al client di dichiarare la propria età e creare un post di benvenuto su questo nuovo social network.



Pattern Observer:

Il secondo pattern presente in questo pacchetto è il pattern Observer. Come già detto in precedenza ho utilizzato questo pattern per implementare un following di un profilo. L'idea alla base di questa parte di programma, è che la classe **Profile** è l'unico *subject* concreto che viene osservato da diverse tipologie di followers concreti, in questo caso due.

La struttura risultate è suddivisa in quattro classi, una astratta e tre concrete, ed un'interfaccia, distribuite in questo modo:

- *ProfileSubject:*

È la classe astratta dei soggetti osservati, conserva una collezione di osservatori ed offre i metodi *attach()* e *detach()* che simulano il follow e l'unfollow. Offre inoltre il metodo *notifyFollowers()* che chiama *update()* su tutti i followers presenti nella collezione.

- *Profile:*

È l'unico soggetto concreto osservabile. Dichiara due metodi relativi al pattern in questione, *isMyBirthday()* che semplicemente aumenta l'età di uno e notifica i followers ed il metodo *publishPost()* che pubblica un nuovo post e notifica i followers.

- *FollowersObserver:*

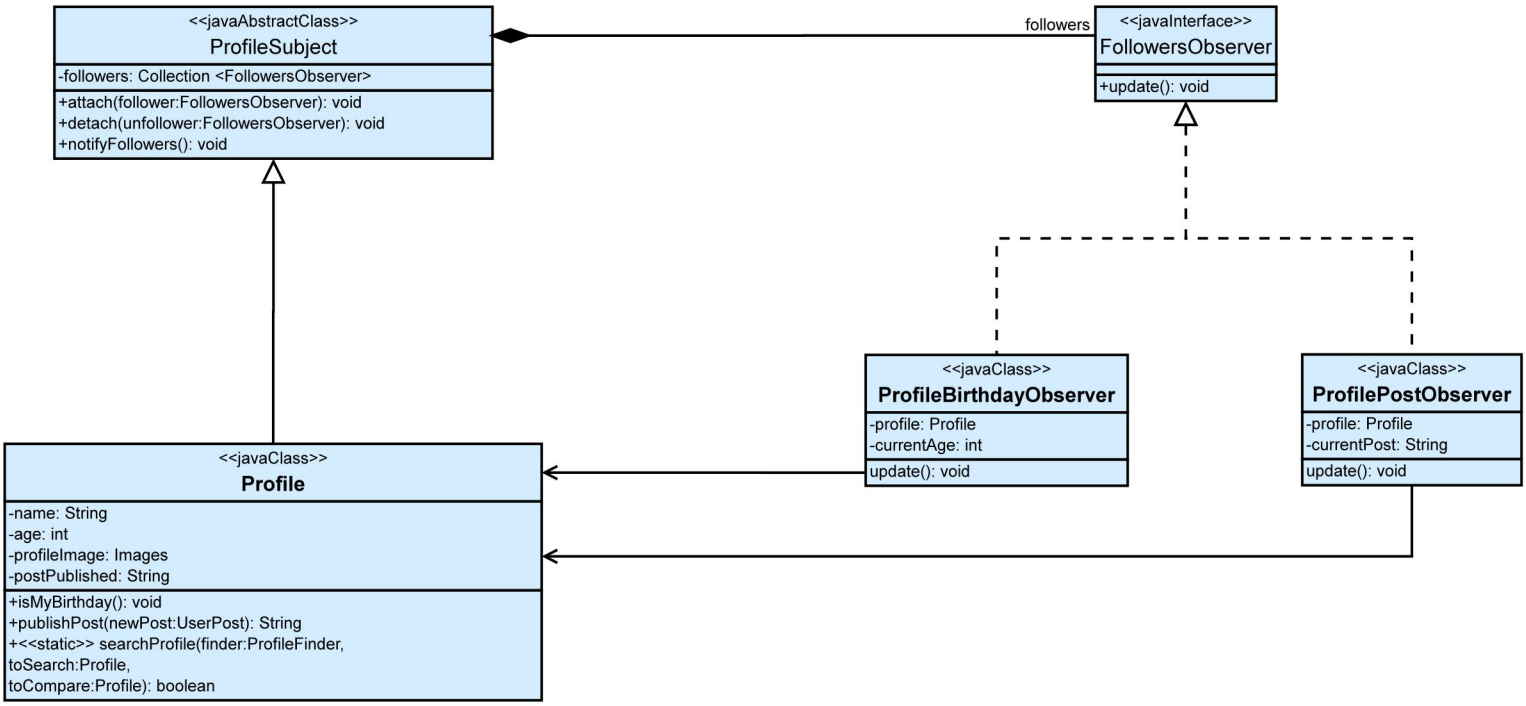
È l'interfaccia java degli osservatori (followers). Dichiara il metodo *update()* che gli osservatori concreti dovranno poi ridefinire.

- *ProfileBirthdayObserver:*

È il follower concreto interessato al compleanno del profilo osservato. Conserva un profilo tra i suoi campi ed un intero. Ridefinisce il metodo *update()* ereditato dall'interfaccia *FollowersObserver,* facendo un confronto tra l'età del profilo osservato ed il campo intero su cui si era memorizzato l'età di quel profilo. Questo confronto serve a capire se la notifica ricevuta sia di suo interesse o meno, evitando inutili riassegnamenti qualora non fosse necessario.

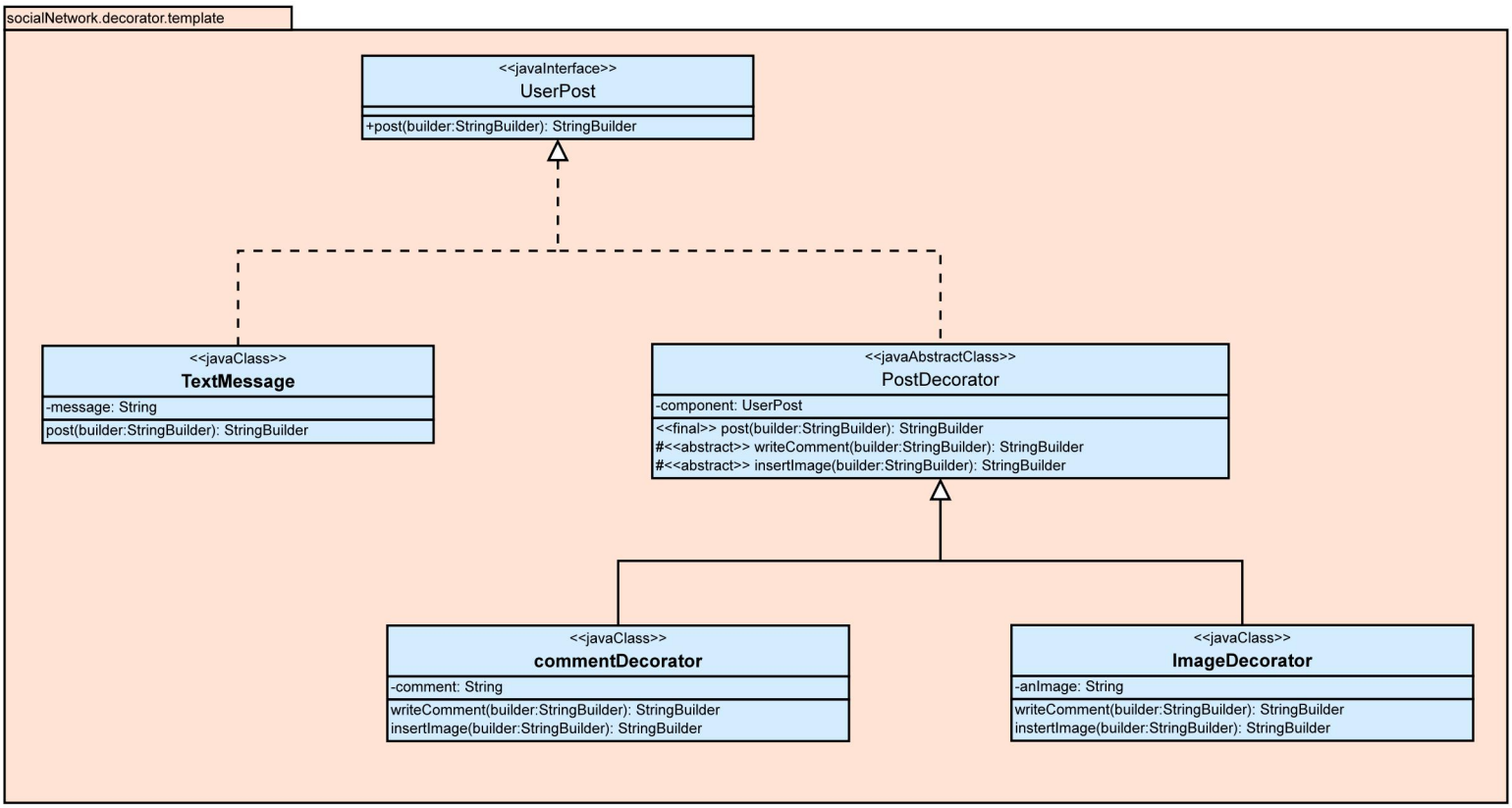
- *ProfilePostObserver:*

Il follower concreto interessato allo stato dei post pubblicati dal profilo seguito. Anch'esso tiene un riferimento del profilo ed un campo di tipo *String* dove si salva lo stato del post attuale. Ridefinisce il metodo *update()* ereditato dall'interfaccia *FollowerObserver* facendo un controllo sullo stato del post ogni volta che riceve una notifica. Anche questo controllo serve per evitare riassegnamenti ogni qual volta si viene notificati per uno stato di non interesse.



socialNetwork.decorator.template:

In questo pacchetto ho racchiuso tutte le classi che ho implementato per far funzionare la scrittura di un nuovo post. Dal nome del pacchetto si intuisce che abbia deciso di utilizzare il pattern Decorator associato al Template. La scelta di utilizzare questo particolare pattern è venuta alquanto spontanea, in quanto durante la scrittura di un post, si possono aggiungere o meno decorazioni, come un'immagine o un commeto. Il pattern Decorator offre la possibilità di poter aggiungere nuove decorazioni, senza rompere le precedenti scritte, ad esempio, il tipo di colore del testo piuttosto che il tipo di carattere da utilizzare; per la scrittura di questo software mi sono limitato a implementare solo due decoratori concreti, in quanto averne di più non era necessaria ai fini del progetto.



Pattern Decorator:

Studiando le diverse implementazioni viste a lezione ho deciso di utilizzare il pattern Decorator che utilizza al suo interno il Template Method. Questa particolare implementazione stabilisce, nella classe astratta, lo schema di decorazione, scaricando le sottoclassi dalla responsabilità di dover inoltrare la chiamata al componente di base prima o dopo la decorazione. Inoltre, questa variante mi è sembrata più sicura, perché in un eventuale ampliamento del software che aggiunge altri decoratori concreti, il rischio di creare un bug dimenticandosi di inoltrare la chiamata al componente principale non è possibile. Inoltre anche se lo schema di decorazione è fissato, ciò non è sembrato un problema, poiché anche nei social network reali, la struttura dei post è definita.

La struttura risultante è suddivisa in tre classi concrete Java, una classe astratta, ed una interfaccia, distribuite in questo modo:

- *UserPost:*

È l'interfaccia Java che permette al client di interfacciarsi con il Decorator. Dichiara il metodo *post()* che le sottoclassi dovranno poi ridefinire.

- *TextMessage:*

È la classe che rappresenta il componente concreto del Decorator, cioè il messaggio principale del post che può essere eventualmente decorato. Ridefinisce il metodo *post()* ereditato dall'interfaccia **UserPost** che implementa.

Il metodo prende come parametro un oggetto di classe **StringBuilder**, che serve per la costruzione del messaggio. Ciò avviene tramite il metodo *.append()* della classe **StringBuilder**.

- *PostDecorator:*

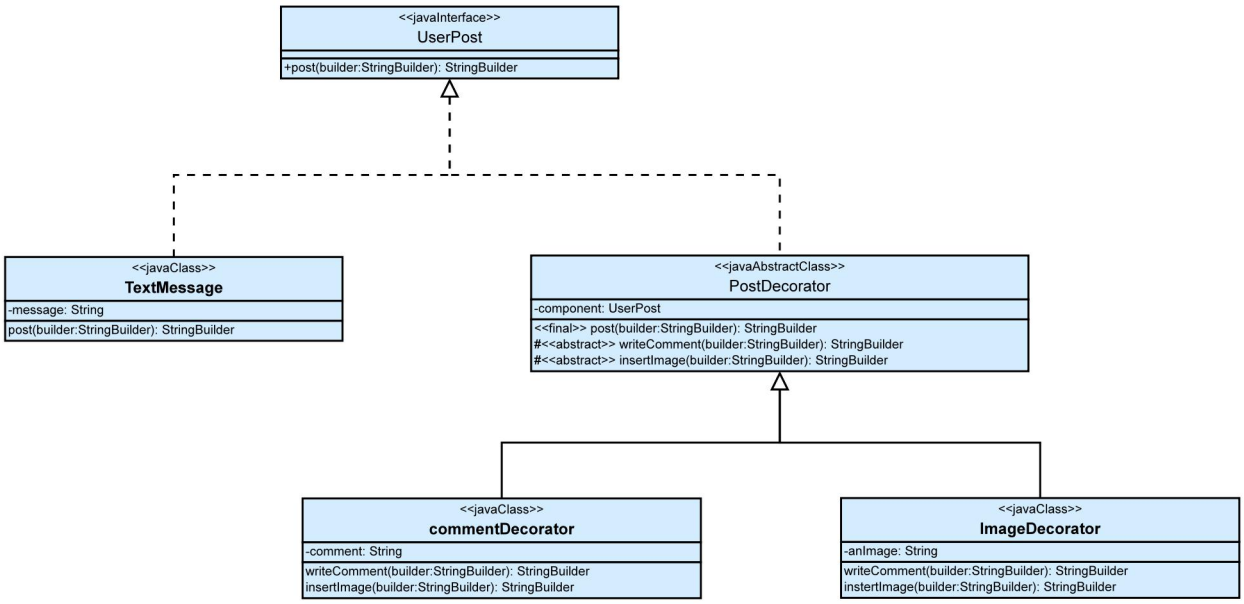
È la classe astratta Java, super-classe dei decoratori concreti, che implementa l'interfaccia **UserPost**. Implementa il metodo *post(),* definendo la struttura di decorazione e la chiamata al componente di base **TextMessage,** e lo rende *final*. Definisce due metodi astratti *writeComment()* e *insertImage()* che le sottoclassi erediteranno e implementeranno.

- *CommentDecorator:*

È uno dei due decoratori concreti, estende la classe astratta **PostDecorator,** ereditandone i metodi astratti. Implementa il metodo con cui intende decorare, in questo caso *writeComment()* e fornisce un'implementazione di base per il metodo *insertImage()*. Il metodo *writeComment()* prende come parametro un oggetto di classe **StringBuilder** che serve per concatenare il commento.

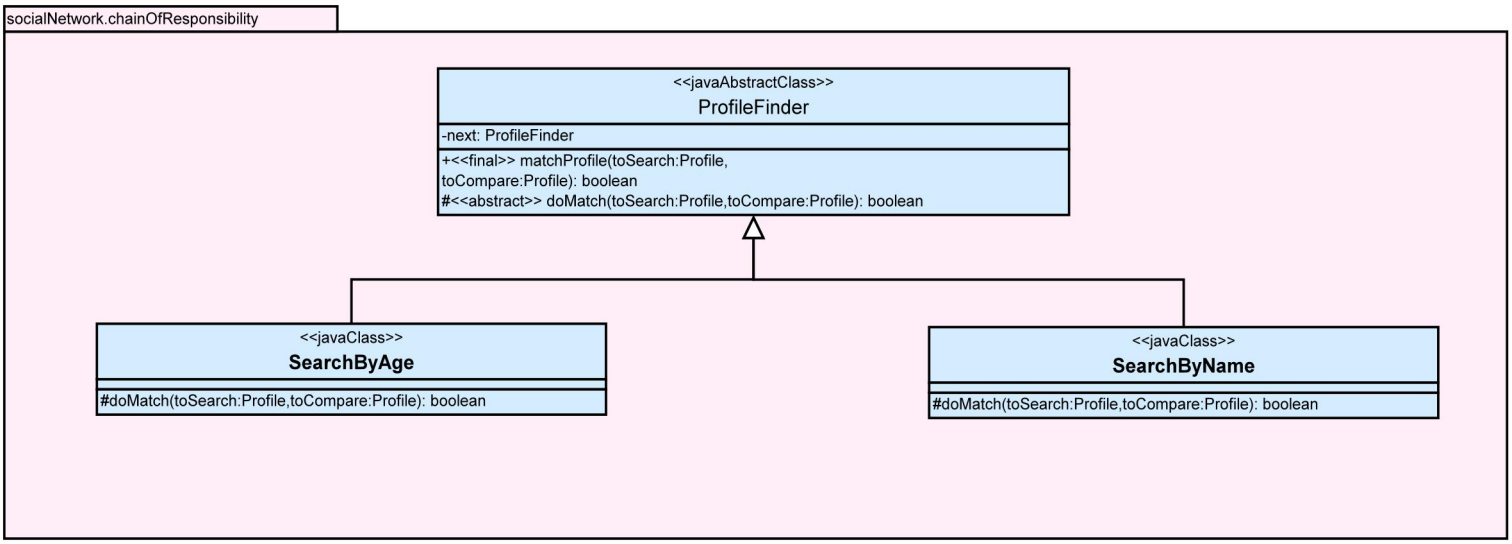
- *ImageDecorator:*

È la classe Java che rappresenta il secondo decoratore concreto implementato. Anch'essa estende la classe astratta **PostDecorator** ereditandone i metodi astratti *writeComment()* e *insertImage()*. Ridefinisce il metodo con cui decora il post, in questo caso *imageDecorator()* prendendo come parametro in ingresso un oggetto della classe **StringBuilder.** Quest'ultimo serve per il concatenamento di un'immagine al post. Lascia una implementazione di base per il metodo *writeComment()*.



socialNetwork.chainOfResponsibility:

L'ultimo pacchetto che andremo ad esplorare è anche l'ultimo pacchetto inserito ed implementato in questa modellazione software. In questo pacchetto sono inserite le classi che implementano la funzione di ricerca di un profilo. Per implementare questa funzionalità ho preso spunto dalla ricerca che si ha nel famosissimo social network scritto da uno studente di Harvard. In pratica scritto qualcosa nella barra di ricerca, l'algoritmo restituisce una lista di tutti i profili che hanno qualche corrispondenza uguale a quella passata come parametro. Nel mio caso ho dovuto simulare tale funzionalità, in quanto non avendo a disposizione un alto numero di profili con cui effettuare la ricerca. Ho pensato che una buona idea era quella di passare due profili ad un metodo, e verificare se alcuni tra i loro campi corrispondessero, e in caso positivo notificare in qualche modo il client che ha effettuato la ricerca. Riflettendo sull'implementazione da effettuare, la scelta del pattern da utilizzare è caduta spontaneamente sul Chain Of Responsibility.



Pattern Chain Of Responsibility:

Mantenendo lo stesso stile di programmazione, tra le diverse implementazioni possibili del pattern, ho deciso di utilizzare quella che si serve di un Template Method. Come già detto in precedenza preferisco queste versioni, in quanto le sottoclassi non hanno la responsabilità dell'inoltro delle chiamate, essendo che tutto viene gestito dalla super-classe, ed inoltre rende più sicuro l'ampliamento delle classi concrete di ricerca. La struttura del pattern risulta comunque abbastanza lineare e semplice ed è composta da una classe astratta e due classi derivate concrete.

- *ProfileFinder:*

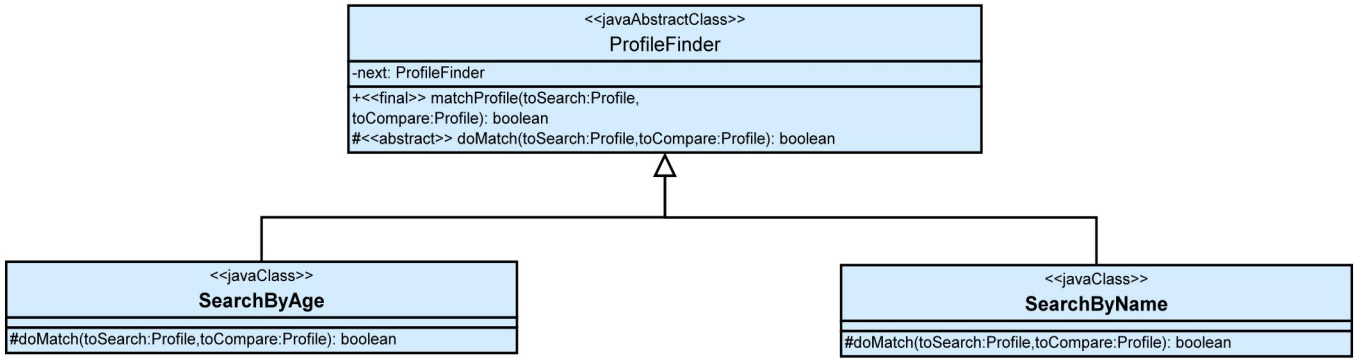
È la super-classe astratta che gestisce lo schema di inoltro delle chiamate, ha un campo del suo stesso tipo che serve a implementare tale meccanismo e ad interrompere le chiamate quando la gestione del problema non è riuscita. Dichiara un metodo *final matchProfile()* in cui lo schema dell'inoltro delle chiamate resta fissato, senza possibilità per le sottoclassi di ridefinirlo. Definisce inoltre il metodo *protected abstract doMatch()* che serve alle sotto-classi concrete per definire la loro logica per la risoluzione del problema.

- *SearchByAge:*

È una delle due classi concrete derivate dalla super-classe **ProfileFinder**. Eredita il metodo *doMatch()* ridefinendolo secondo la propria logica di ricerca. Essendo lo schema di inoltro fissato nella super-classe, la classe **SearchByAge** non si deve preoccupare di continuare gli inoltri con la chiamata alla super-classe tramite il *super*.

- *SearchByName:*

È la seconda ed ultima classe concreta derivata dalla super-classe **ProfileFinder.** Eredita il metodo doMatch() preoccupandosi di ridefinirlo solo secondo la propria logica di ricerca, senza avere la responsabilità di continuare la catena con la chiamata a *super*, essendo questo inoltro fissato nella super-classe.



Analisi dei test effettuati:

Come da specifica richiesta, il progetto è stato completato con la scrittura di test, che sono stati effettuati utilizzando *JUnit 4*. Tutti i test sono inoltre stati scritti con la libreria di asserzioni A*ssertJ*, in quanto la scrittura risulta molto più comprensibile ed immediata, ed in caso di fallimento è molto più semplice e chiaro capire dove risiede l'errore che ha portato al fallimento del test. I test sono stati compilati seguendo attentamente le direttive che sono state proposte a lezione, cercando quindi di testare ogni singolo metodo in maniera esaustiva. Secondo quanto verificato dalla Coverage dei test messa a disposizione de Eclipse, il 98.1% del codice risulta testato, con la sola eccezione, come si evince dall'esecuzione della Coverage in Eclipse, di due test scritti nel pacchetto **socialNetwork.observer.builder,** rispettivamente *testBuilderAgeException()* e *testBuilderPostException().* Questi, testano i metodi che il Builder mette a disposizione per la creazione di un oggetto con campi opzionali, che se creati con argomenti non consentiti, sollevano un'eccezione. Se si trattasse di un progetto più complesso e completo, sarebbe opportuno che tutti i possibili rami del codice siano testati in tutte le loro combinazioni, ma ai fini di questa prova, non risulta necessario, ed inoltre potrebbe portate ad avere test eccessivamente lunghi o complessi.
Si è inoltre prestata molta attenzione a seguire il principio per cui nei test non dovrebbero comparire metodi testati.