Progetto: Metodologie di programmazione

A.A. 2020-2021

Autore

Francesco Marchini

7029444 francesco.marchini1@stud.unifi.it

Menu

Descrizione

Il sistema implementato si pone come obiettivo il modellamento software di un menu opzioni, suddiviso in varie tab (o schede) all'interno delle quali possono essere inserite altre schede o opzioni. Il progetto viene implementato usando Java 11.

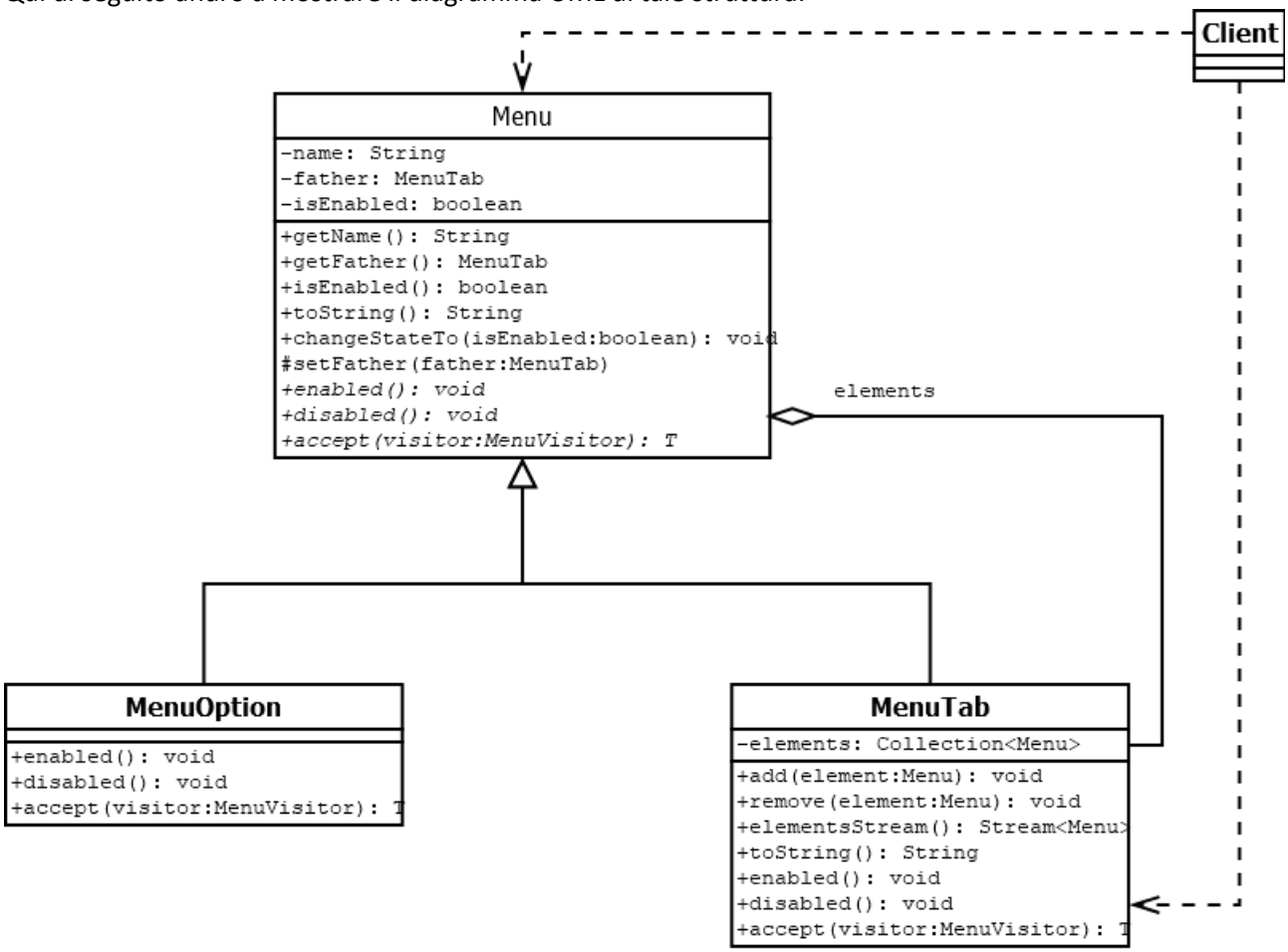
Il nucleo dell'applicazione che rappresenta gli oggetti su cui andremo ad eseguire delle operazioni è stato implementato con l'ausilio del pattern strutturale Composite in variante Type Safety ed è composto dalle seguenti classi:

- Menu;
- MenuTab;
- MenuOption.

Sono state implementate le seguenti operazioni su questa struttura:

- possibilità di attivare o disattivare una singola opzione o una tab intera;
- possibilità di interrogare un oggetto per sapere se è attivato o meno;
- accesso al nome di ogni elemento;
- accesso al riferimento al padre di ogni elemento;
- stampa a video di una stringa contenente informazioni su un oggetto;
- ricerca tramite nome di un'opzione o una scheda;

Qui di seguito andrò a mostrare il diagramma UML di tale struttura.



Come detto precedentemente la classe Menu presenta due metodi fondamentali per interagire con il menu:

- enabled();
- disabled().

Questi vengono implementati in modi differenti nelle varie sottoclassi concrete, in MenuOption il metodo *enabled()* si preoccupa di porre a "true" il valore del booleano *isEnabled*, campo della classe, mentre in MenuTab non solo viene abilitato il campo dell'oggetto su cui è chiamato ma anche quello di tutti gli elementi che ne fanno parte e che si trovano all'interno della struttura dati *elements*. Stessa cosa vale per il metodo *disabled()* che, come è facile immaginare, avrà il compito di portare il valore di *isEnabled* a "false" con le stesse modalità di *enabled()*.

Questo meccanismo offre all'utente la possibilità di abilitare o disabilitare in una volta sola tutte le opzioni e sottomenu di una scheda o di poter anche abilitare una singola opzione.

In ogni istante del programma una scheda saprà sempre se tutti i suoi elementi sono disattivati o attivati e in tal caso anch'essa si attiverà automaticamente, viceversa basta un solo elemento disattivato per far disattivare anche il valore della tab. Questo automatismo è affidato al pattern Observer che andrò successivamente a trattare.

MenuTab

Fino ad ora abbiamo visto le operazioni comuni a tutta la struttura, adesso andrò a descrivere le operazioni che sono proprie della classe MenuTab, in quanto è stata scelta la variante Type Safety del pattern Composite in modo da non dover inserire metodi non necessari all'interno della classe MenuOption.

Tale classe concreta presenta le seguenti operazioni:

- aggiunta di un elemento alla scheda;
- rimozione di un elemento dalla scheda;
- vista di tutti gli elementi presenti in una scheda tramite uno Stream.

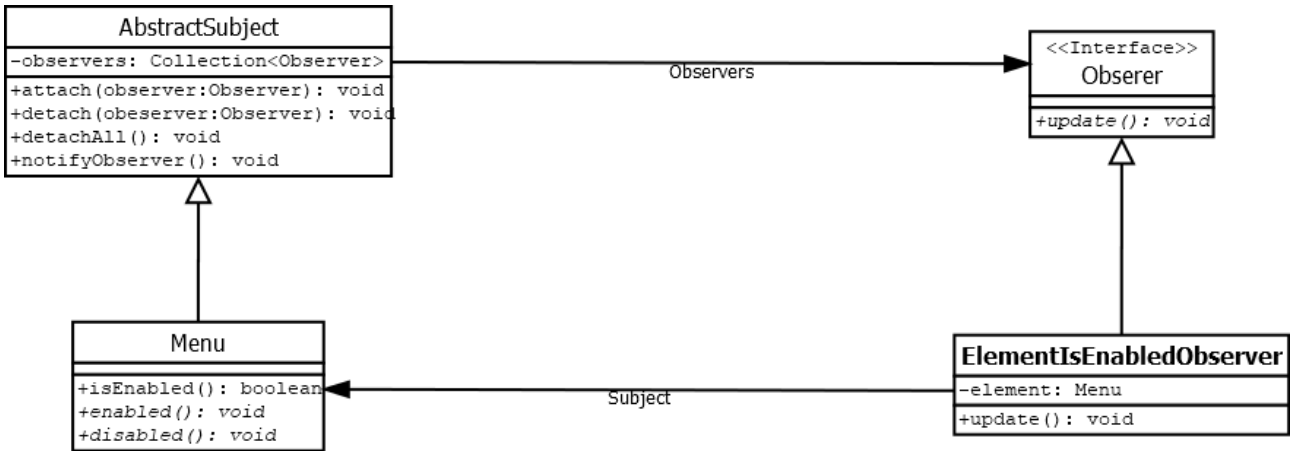
I rispettivi metodi, visibili nel grafico UML, servono a gestire gli elementi che vengono aggiunti alla tab e quindi sarebbe stato errato inserirli nella classe astratta Menu (come nella versione Design for Uniformity) costringendo MenuOption ad implementarli.

Il metodo *add()* quando richiamato non solo inserisce l'elemento passato come argomento alla lista *elements* ma esegue anche altre due operazioni:

- setta il padre di questo elemento, che sarà la tab entro la quale viene inserito;
- esegue l'attach di un nuovo Observer all'elemento passato.

Il metodo *remove()* si comporta in maniera analoga ma inversa a *add()*, rimuove l'elemento passato dalla lista, ne rimuove il padre e l'Observer.

Observer



Come abbiamo visto l'esigenza di implementare un Observer nasce dalla necessità di sapere quando cambia di stato l'elemento di una tab, ovvero quando viene attivato o disattivato, in modo che la scheda possa adeguarsi a tale cambiamento.

In questa implementazione l'Observer concreto (*ElementIsEnabledObserver*) si preoccupa di modificare lo stato della tab ogni qual volta un suo elemento viene modificato.

AbstractSubject

Al suo interno viene dichiarata una collezione di observer dentro la quale andranno ad essere inseriti tutti gli observers di un elemento. Il metodo *attach()* svolge proprio il compito di aggiungere un observer a tale lista e viene richiamato quando andiamo ad eseguire la *add()* di un elemento ad una scheda, in modo che la procedura sia del tutto invisibile ai clients. La *detach()* invece svolge la funzione opposta andando a rimuovere l'observer passato come argomento dalla lista degli observer, in questa implementazione andremo però a fare uso del metodo *detachAll()* che rimuove tutti gli observers all'interno della lista, senza comunque andare a sopprimere il precedente metodo che viene lasciato in caso di futuri aggiornamenti in cui si rendano necessari più Observer per uno stesso elemento. Come per *attach()*, *detachAll()* viene automaticamente invocato su un oggetto quando andiamo ad invocare la *remove()* di esso all'interno della sua scheda di pertinenza.

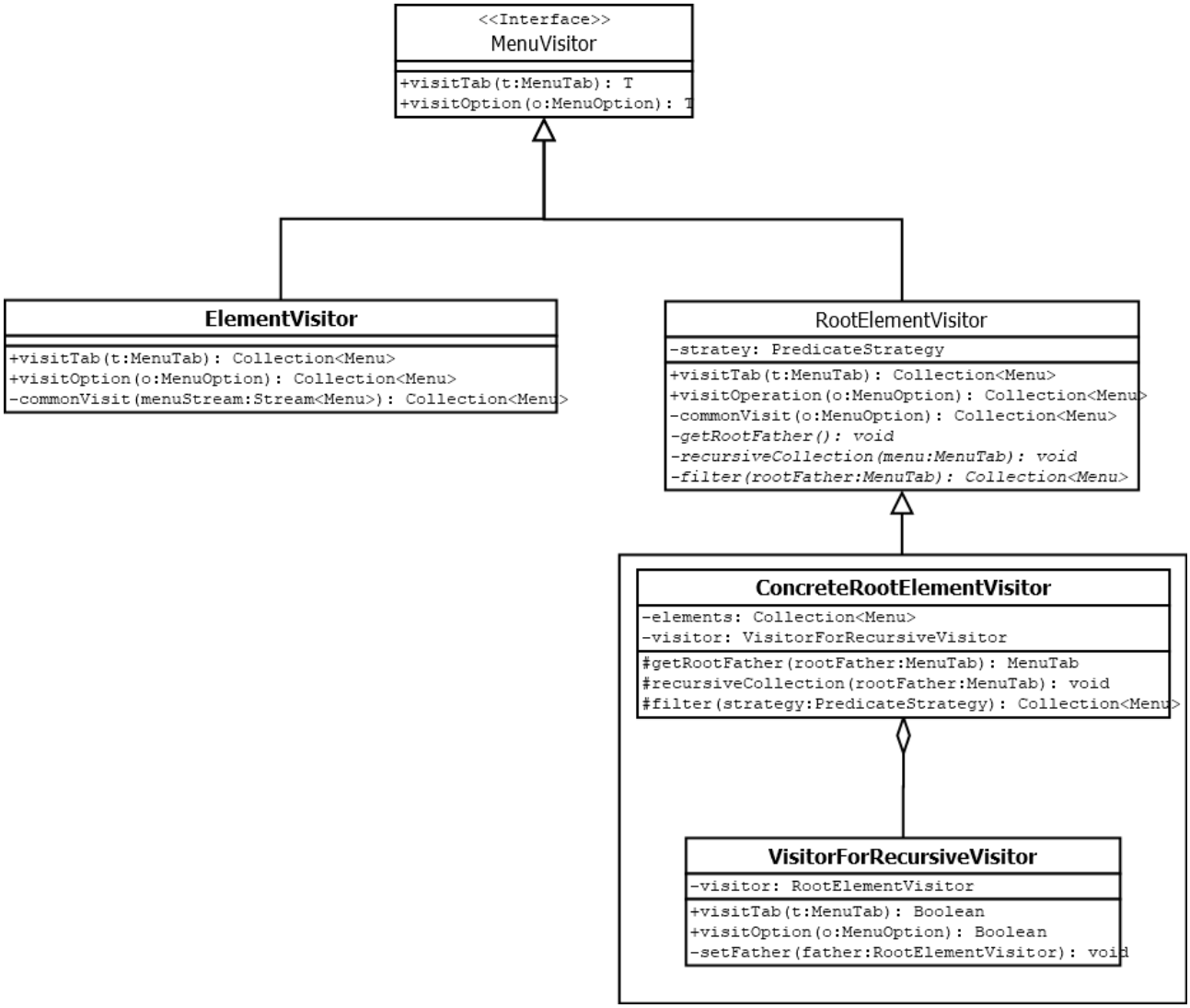
La *notifyObservers()* viene richiamata all'interno di un metodo particolare della classe Menu: *changeStateTo()*. Questo metodo ha il compito di gestire il parametro isEnabled dell'oggetto e di notificare gli Observers. La soluzione più ovvia poteva essere quella di inserire la notifica all'interno di ogni metodo *enabled()/disabled()* ma così facendo rischiamo una cascata di chiamate ricorsive che finiscono con il saturare la stack della Java Virtual Machine. Il problema sorge nel momento in cui MenuTab esegue uno di questi due metodi i quali ricorsivamente richiamano sé stessi su tutti gli elementi presenti all'interno della lista *elements* andando a notificare anche tutti i sottomenu e opzioni di una tab, quando ciò che interessa a noi è che la notifica prosegua verso l'alto andando a modificare chi sta sopra l'elemento modificato, non chi sta sotto.

ElementIsEnabledObserver

L'unico metodo presente nella classe è *update()* il quale una volta invocato risale alla tab di pertinenza dell'elemento invocante e va a controllare lo stato di tutti gli elementi di cui è composta, se essi sono tutti abilitati allora verrà abilitata anche la scheda se invece verrà trovato anche un solo elemento disabilitato la scheda rimarrà disabilitata (o se non lo era verrà disabilitata senza interferire con i suoi elementi interni).

Il metodo *update()* non richiama i metodi *enabled()* e *disabled()* per modificare lo stato di una tab, ma richiama un metodo usato da essi di cui trattato in precedenza, *changeStateTo()*. Questo prende come parametro un booleano e in base ad esso viene cambiato o stato della scheda notificando il proprio Observer, nel fare ciò non vengono intaccati gli elementi contenuti nella tab, non vi è alcuna chiamata ricorsiva ai figli.

Visitor



Questa implementazione del Visitor ci permetterà di navigare all'interno delle impostazioni secondo due tipologie di ricerche:

- ricerca per nome;
- ricerca per elemento attivato/disattivato.

Le due tipologie di ricerca sono definite da uno Strategy il quale sarà successivamente a trattato. I due visitors definiscono due attraversamenti differenti, il primo si limita a ricercare all'interno della tab su cui è chiamato, il secondo scansiona tutta la struttura creata.

ElementVisitor

Visitor concreto che si limita alla visita dell'elemento sul quale viene eseguita la *accept()*.

Ha come campo privato uno Strategy che viene inizializzato nel costruttore, questo ha il compito di definire la strategia di visita da adottare.

Dettaglio dei due metodi del visitor:

- *visitOption()*: richiama il metodo privato *commonVisit()* passandogli come argomento uno stream contenente l'elemento su cui è chiamato. Se all'interno di *commonVisit()* la comparazione secondo lo strategy avrà successo verrà ritornata una collezione contenete quell'unico oggetto, altrimenti verrà restituita una lista vuota.
- *visitTab()*: richiama il metodo privato *commonVisit()* passandogli come argomento uno stream contenente tutti gli elementi della tab su cui operiamo. *commonVisit()* ritornerà una collezione contenente tutti gli elementi che hanno avuto successo durante la comparazione con la strategia definita da Strategy, altrimenti ritorna una lista vuota.

L'aggiunta del metodo *commonVisit()* è stata effettuata per eliminare una duplicazione di codice.

RootElementVisitor

Visitor astratto che definisce l'algoritmo per la visita dell'intera struttura indistintamente dall'oggetto su cui è invocato.

A differenza del precedente visitor, questo fornisce un metodo di ricerca universale all'interno della nostra struttura, in qualunque sezione del nostro menu noi ci troviamo, in ogni momento saremo sempre in grado di poter ricercare una funzione e accedervi.

Come prima anche qui è presente un campo privato Strategy che viene inizializzato nel costruttore e definisce la strategia di visita da adottare.

I due metodi pubblici anche in questo caso si preoccupano di richiamare un metodo privato *commonVisit()* comune che farà partire la visita richiamando tre metodi astratti; vi è dunque l'implementazione di un Template Method:

1. *getRootFather()*, il quale dovrà recuperare il nodo radice della struttura;
2. *recursiveCollection()*, che raccoglierà tutti gli elementi dell'albero partendo proprio dalla radice precedentemente trovata;
3. *filter()*, ultimo metodo dell'algoritmo che filtrerà tutta la collezione secondo il predicato definito dallo strategy.

ConcreteRootFatherVisitor

Classe concreta che eredita da *RootElementVisitor* e ne implementa i metodi astratti per definire concretamente le operazioni da eseguire nel template method *commonVisit()*. L'utilizzo di questo pattern apre il codice a successive implementazioni di varianti delle operazioni da eseguire oltre che a deresponsabilizzare la classe base.

Di queste tre operazioni, l'operazione di raccolta di tutti gli elementi è quella che merita un approfondimento per comprenderne l'implementazione.

Il metodo che si occupa di questa operazione è *recursiveCollection()*, il quale deve poter distinguere tra un MenuOption e un MenuTab affinché la raccolta degli elementi proceda correttamente. Staticamente gli elementi processati da questo metodo sono tutti di tipo Menu, dunque in assenza di un overloading dinamico dei metodi da parte di Java, è stato necessario introdurre un ulteriore visitor come classe interna di *RootElementVisitor* per poter discernere il tipo effettivo dell'oggetto in questione.

VisitorForRecursiveVisitor

Il compito di questo visitor è quello di far proseguire correttamente la raccolta ricorsiva di tutti gli elementi che compongono la nostra struttura ad albero. All'interno del metodo *recursiveCollection()*, appartenente alla classe sopra descritta, partendo dal nodo radice, vengono visitati e aggiunti ad una collezione tutti i suoi figli, ma è necessario poter distinguere se tra di essi sono presenti *composite,* così da far progredire la visita anche al loro interno, oppure delle *leafs* e quindi capire di essere arrivati alla fine di un ramo della struttura.

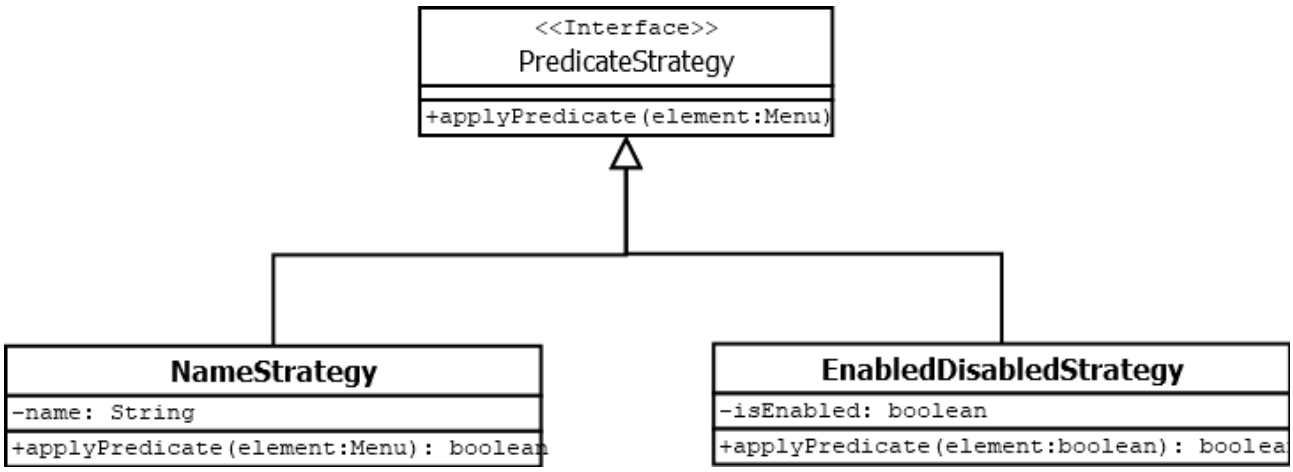
All'interno di *VisitorForRecursiveVisitor* è presente un campo di tipo *RootElementVisitor* che servirà per richiamarne il metodo *recursiveCollection()*.

Affinché ciò avvenga analizziamo i metodi della classe:

- *visitTab()*: richiama il metodo *recursiveCollection()* sul campo visitor della classe passando come argomento l'oggetto di tipo MenuTab passato al metodo.
- *visitOption()*: il metodo non esegue alcuna operazione, indice di un ramo finito dell'albero.

Questo meccanismo garantisce, richiamando il metodo accept sull'oggetto in esame all'interno di *recursiveCollection()*, di far proseguire correttamente la visita dell'intera struttura riuscendo a discernere a runtime il tipo concreto dell'elemento considerato; se infatti si tratta di un tipo *MenuOption* la visita non prosegue in quella direzione in quanto siamo arrivati ad una foglia, se invece il tipo è *MenuTab* la visita prosegue ricorsivamente all'interno della lista presente in esso.

Strategy



Questa struttura, utilizzata dai due visitor concreti, definisce la strategia di filtraggio utilizzata in quest'ultimi.

PredicateStrategy

Interfaccia che definisce l'unica operazione presente e comune a tutta la struttura *applyPredicate()*; questo metodo viene utilizzato dai visitor come predicato per il filtraggio degli elementi durante una visita e viene proprio richiamato all'interno dei vari metodi *visitTab()* e *visitOption()*.

NameStrategy

Responsabile del filtraggio tramite nome, ha come campo una stringa contenente il nome o le iniziai del nome da ricercare che viene inizializzata passando il parametro al costruttore.

All'interno del metodo *applyPredicate()* viene confrontato il nome dell'elemento passato come parametro con la stringa inizializzata dal costruttore tramite il metodo *startWith()* della classe *String*. Se il confronto ha successo viene ritornato un valore booleano "true", altrimenti "false".

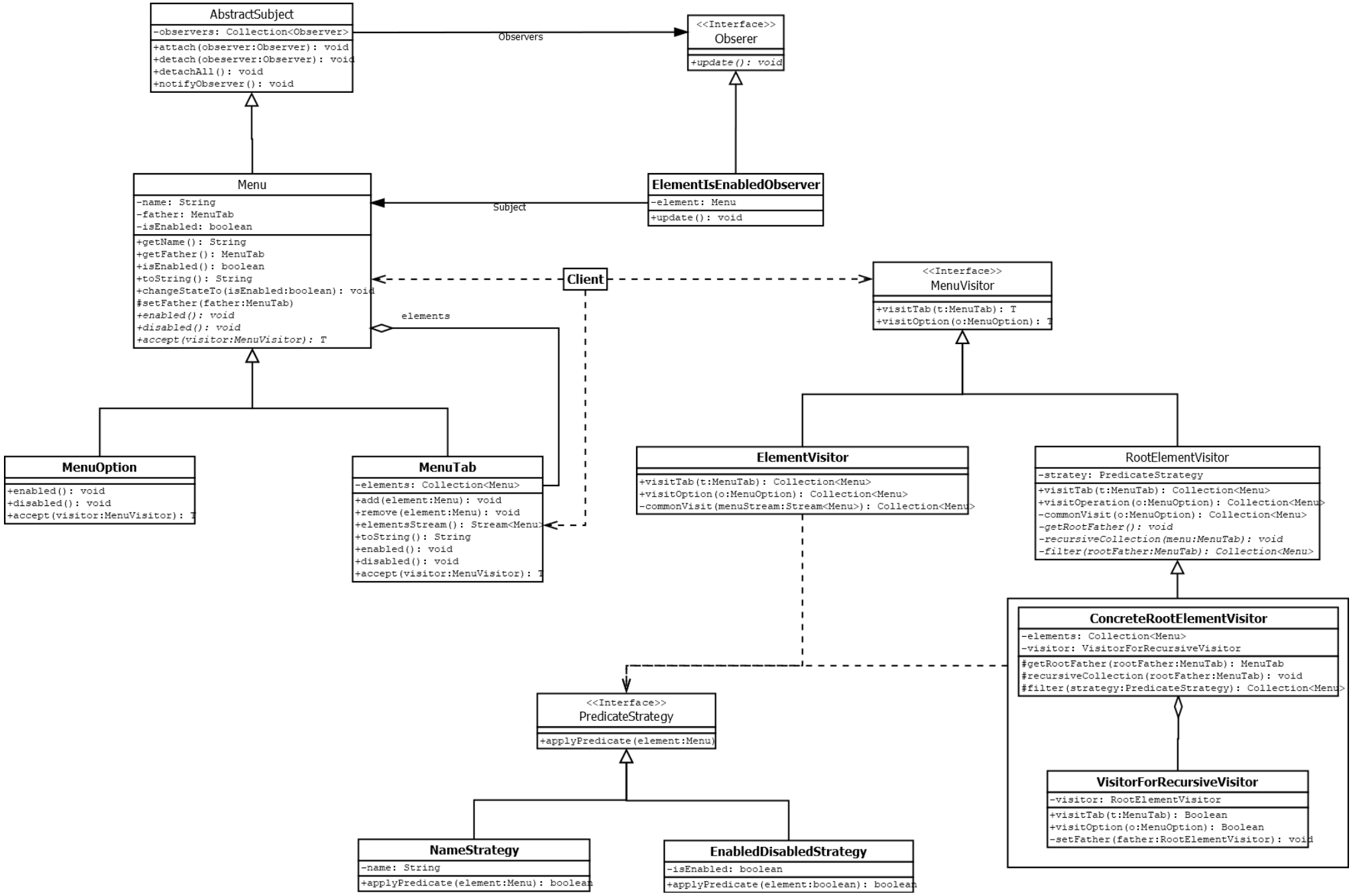
EnabledDisabledStrategy

Responsabile del filtraggio secondo il campo *isEnabled* di un oggetto Menu, ha come parametro un booleano che viene inizializzato passando il parametro al costruttore.

All'interno del metodo *applyPredicate()* viene confrontato il valore *isEnabled* dell'oggetto passato come parametro con il campo inizializzato dal costruttore, in modo da poter ricercare sia elementi abilitati sia elementi disabilitati. Se il confronto ha successo viene ritornato un valore booleano "true", altrimenti "false".

Patterns utilizzati

- Composite;
- Observer;
- Visitor;
- Template Method;
- Strategy.


AbstractSubject
-observers: Collection<Observer>
+attach(observer:Observer): void
+detach(obeserver:Observer): void
+detachAll(): void
+notifyObserver(): void
Observers
<<Interface>>
Obserer
+update(): void
Menu
-name: String
-father: MenuTab
-isEnabled: boolean
+getName(): String
+getFather(): MenuTab
+isEnabled(): boolean
+toString(): String
+changeStateTo(isEnabled:boolean): void
#setFather(father:MenuTab)
+enabled(): void
+disabled(): void
+accept(visitor:MenuVisitor): T
Subject
ElementIsEnabledObserver
-element: Menu
+update(): void
Client
elements
<<Interface>>
MenuVisitor
+visitTab(t:MenuTab): T
+visitOption(o:MenuOption): T
MenuOption
+enabled(): void
+disabled(): void
+accept(visitor:MenuVisitor): T
MenuTab
-elements: Collection<Menu>
+add(element:Menu): void
+remove(element:Menu): void
+elementsStream(): Stream<Menu>
+toString(): String
+enabled(): void
+disabled(): void
+accept(visitor:MenuVisitor): T
ElementVisitor
+visitTab(t:MenuTab): Collection<Menu>
+visitOption(o:MenuOption): Collection<Menu>
-commonVisit(menuStream:Stream<Menu>): Collection<Menu>
RootElementVisitor
-stratey: PredicateStrategy
+visitTab(t:MenuTab): Collection<Menu>
+visitOperation(o:MenuOption): Collection<Menu>
-commonVisit(o:MenuOption): Collection<Menu>
-getRootFather(): void
-recursiveCollection(menu:MenuTab): void
-filter(rootFather:MenuTab): Collection<Menu>
ConcreteRootElementVisitor
-elements: Collection<Menu>
-visitor: VisitorForRecursiveVisitor
#getRootFather(rootFather:MenuTab): MenuTab
#recursiveCollection(rootFather:MenuTab): void
#filter(strategy:PredicateStrategy): Collection<Menu>
<<Interface>>
PredicateStrategy
+applyPredicate(element:Menu)
VisitorForRecursiveVisitor
-visitor: RootElementVisitor
+visitTab(t:MenuTab): Boolean
+visitOption(o:MenuOption): Boolean
-setFather(father:RootElementVisitor): void
NameStrategy
-name: String
+applyPredicate(element:Menu): boolean
EnabledDisabledStrategy
-isEnabled: boolean
+applyPredicate(element:boolean): boolean