
UNIVERSITÀ
DEGLI STUDI
FIRENZE

# *Metodologie di Programmazione*

*A.A. 2023/2024*

*PROGETTO MP*


| |
|---|
| **Nome**: Tommaso |
| **Cognome**: Pastorelli |
| **Matricola**: 7119242 |
| **Email**: tommaso.pastorelli1@edu.unifi.it |
| **Data di consegna**: 11/02/2024 |

# SOMMARIO

# DESCRIZIONE SISTEMA IMPLEMENTATO

Il sistema scelto rappresenta un'applicazione dove sono presenti delle **community**, formate da **sotto-community**, o come sono state chiamate nel progetto "**SubCommunity**" e da delle **chat**. Specifichiamo meglio: l'idea è quella di avere delle chat, nel nostro caso con soli messaggi testuali per ragioni di semplicità ovvie, dove gli utenti, tutti ovviamente diversi tra loro, possano inviare messaggi e ricevere eventuali notifiche riguardanti vari **eventi** che accadono all'interno della chat. L'idea quindi è quella di avere una struttura ad albero, dove abbiamo una community principale che conterrà dei gruppi, e/o delle sotto-community che a loro volta potranno contenere altre sotto-community e/o gruppi.
Per fare un esempio, potremmo avere una community sul calcio, con all'interno un insieme di sotto-community riguardanti i vari campionati di calcio mondiali(Serie A, Liga Spagnola ecc…), e per ognuno di questi avere al loro interno un insieme di gruppi testuali per ogni squadra di quello specifico campionato. Questo implica ovviamente che un utente potrebbe unirsi ad una chat di una community e ad un altra no in base alle sue preferenze(Per esempio potrei volermi unire alla chat di una certa squadra ma non a quella di un altra).
Sulla struttura potranno essere eseguite varie operazioni, quelle scelte in questo caso specifico sono state 2, ovvero, la "**raccolta**" di tutti gli utenti presenti in una sotto-community o di una chat, ovviamente senza duplicati, e la stampa dei messaggi di tutte le chat, come prima, di una certa sotto-community o direttamente di una chat.
Come già accennato, inoltre, gli utenti di una chat potranno essere notificati all'avvenire di un certo avvenimento, come, per esempio nel nostro caso, quando un messaggio viene inviato, o un utente aggiunto o rimosso. E' stato implementato anche una struttura per distinguere e gestire tutti i tipi di eventi in base al loro tipo.
Si specifica che il progetto è stato sviluppato in **Java 11**.

# Pattern applicati

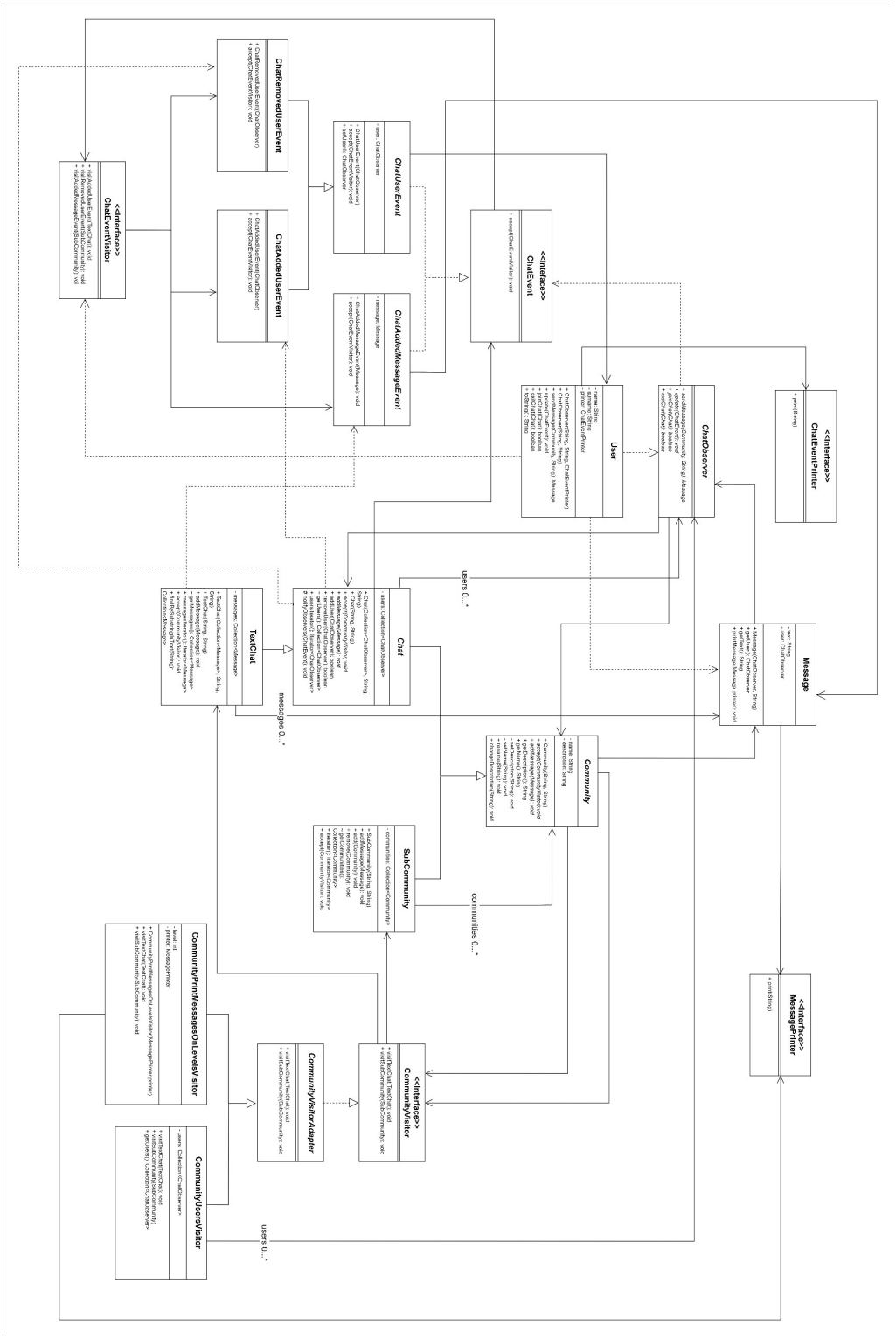
I pattern applicati nel progetto sono i seguenti:

- ***COMPOSITE***
- ***VISITOR***
- ***OBSERVER***
- ***ADAPTER***

UNIVERSITÀ DEGLI STUDI FIRENZE

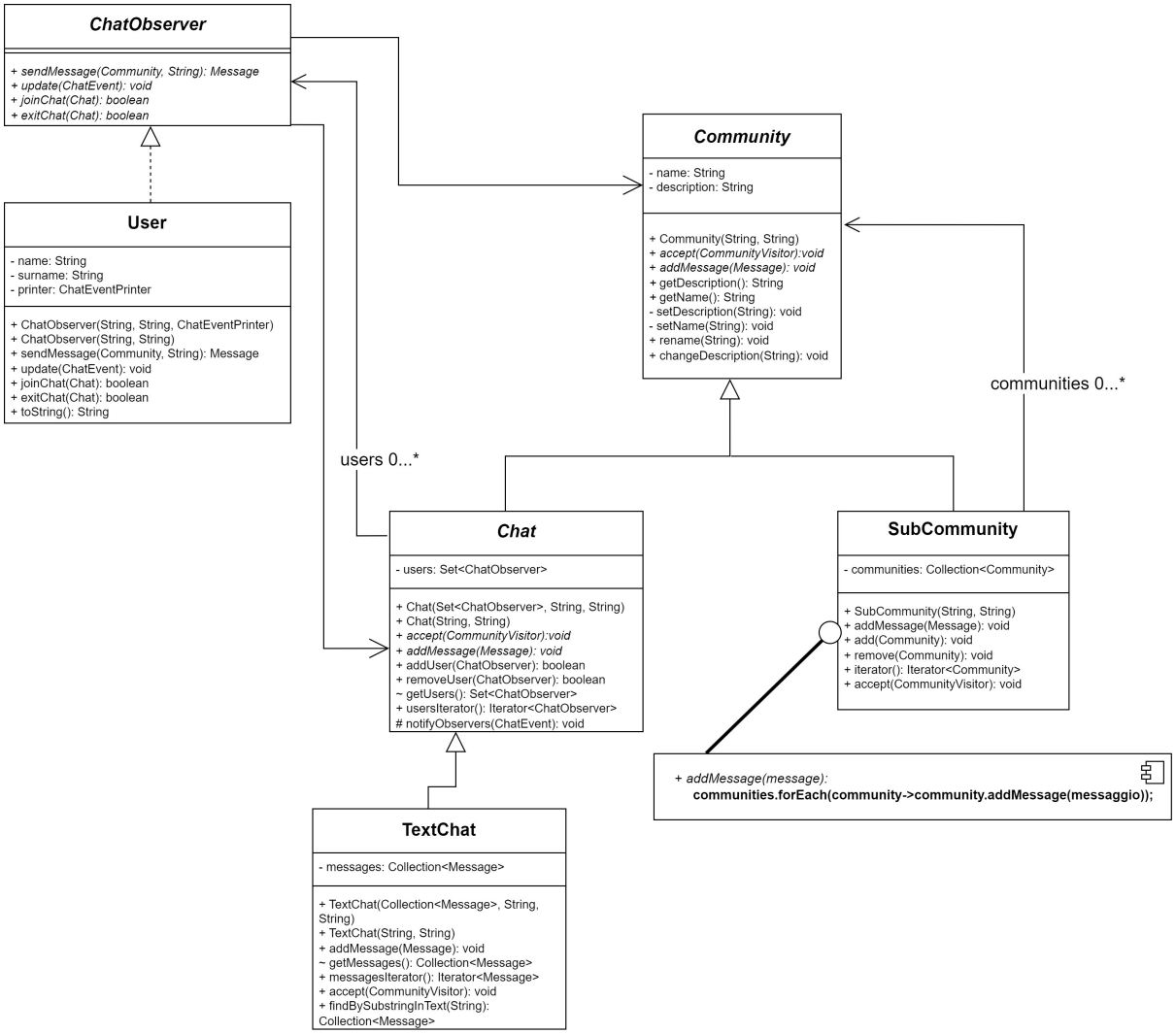
# DIAGRAMMI UML E SPIEGAZIONE PROGETTO

- ***UML COMPLETO:***



# • *PATTERN COMPOSITE:*

## UML




Si voleva utilizzare un pattern in grado di supportare il tipo di struttura ad albero, con community, sotto-community e chat, specificata prima. Il pattern più adatto a questo compito è risultato essere il pattern **Composite**. Delle due versioni applicabili del pattern, quella "***Design for Uniformity"*** e quella *"**Design for Type Safety***", è stata scelta la seconda, questo poiché si è preferito che il client non possa andare ad effettuare operazioni di un composito su una foglia incontrando eventuali errori run-time, soprattutto in questo caso dove è giusto sapere se stiamo lavorando su una foglia(***TextChat***) o una sotto-community(***SubCommunity***). Le operazioni comuni a tutta la struttura, ovvero definite nella classe astratta **Community**, sono le operazioni di aggiunta di un messaggio e di accept(che servirà poi per il visitor). L'operazione di aggiunta di un messaggio viene eseguita nelle foglie aggiungendo il messaggio direttamente alla struttura contenente i messaggi, mentre nel composito viene eseguita inoltrando il messaggio a tutte le sotto community o chat di cui

l'utente fa parte. Per inoltrarlo solo alle Chat di cui l'utente fa parte è stato inserito un controllo prima dell'aggiunta del messaggio all'interno del metodo *addMessage.*

Le operazioni di aggiunta, rimozione e restituzione dei vari Component(*Community*), designate ovviamente solo per la classe concreta SubCommunity, sono state inserite solo in quest'ultima. In particolare l'operazione di restituzione delle community è stata aggiunta sia con un getter package-private da utilizzare nei test, sia tramite un metodo pubblico che restituisce l'Iterator della Collection di communities(E' stato utilizzato quello di Java, per cui non verrà considerata come applicazione del pattern).

Prendiamo invece ora in esame la classe astratta **Chat**. Questa è stata inserita come astrazione per rendere meno ridondante il codice, infatti per come è stato pensato il progetto, tutti i tipi di chat che potremmo voler aggiungere, oltre a quello già presente, TextChat, avranno sicuramente al loro interno degli utenti, e per questo, si è pensato di inserire un Set di utenti nella classe astratta, con annesse tutte le varie operazioni su questi, ovvero aggiunta, rimozione, restituzione di un Iteratore, e, sempre tramite un metodo package-private, un metodo utile per i test "*getUsers()*". In particolare i metodi "*addUser*" e "*removeUser*" che prendono come parametro un "ChatObserver", che è un'astrazione dell'**User**(che sarà poi l'observer nel pattern relativo) , restituiranno un booleano per indicare semplicemente ove l'utente che si aggiunge(o viene aggiunto) alla chat, sia riuscito o meno a completare l'operazione(lo stesso vale per la rimozione/uscita da un gruppo). Vedremo poi meglio come la scelta di posizionare gli utenti nella superclasse astratta sia anche parte integrante nell'applicazione di un altro pattern, ovvero l'observer.

Tornando a noi, esaminiamo infine la classe **TextChat** che, per l'appunto, estende Chat, e che salverà al suo interno una collezione di messaggi della classe **Message**. La collezione di messaggi è stata inserita in TextChat e non in Chat per questioni di semplicità e per non rendere, almeno nel contesto del progetto, troppo carica di metodi e attributi la superclasse astratta. Nel caso si volesse effettuare un Pull Up della collezione di Messaggi e dei metodi di gestione di quest'ultimi nella superclasse non vi sarebbero molte complicanze, infatti, basterebbe poi inserire solo le nuove operazioni(e implementare i metodi astratti) nelle sottoclassi concrete di Chat, ovvero le foglie della struttura composite, lasciando le operazioni di gestione di base dei messaggi alla superclasse(Per esempio vorrei poter aggiungere un tipo di Chat solo per le comunicazioni da parte di certi utenti specifici come avviene su molte app di messaggistica esistenti).

Parliamo anche delle classi **User** e **Message** e dell'interfaccia **ChatObserver** che sono nel nostro scope i client che interagiscono con le chat. L'interfaccia ChatObserver fungerà sia da Observer per il pattern apposito… sia da astrazione per gli utenti, infatti, riferendoci a questa astrazione invece che direttamente alle sottoclassi concrete, sarà possibile facilitare l'espansione futura del programma aggiungendo nuovi tipi di utenti, come magari un amministratore o un utente in sola lettura, per fare dei semplici esempi. Nel nostro caso, come vedremo ora, si è scelto di utilizzare una sola implementazione concreta poichè ritenuta sufficiente per mostrare le funzionalità del progetto e l'utilizzo dei pattern.
L'interfaccia ChatObserver conterrà **4 metodi astratti** da ridefinire, ovvero "*sendMessage*", "*update*", "*joinChat*" e "*exitChat*".
Il primo metodo sarà utilizzato per inviare un messaggio ad una certa chat, lasciando la creazione del messaggio all'interno del metodo data una stringa di input. Il metodo *update*

sarà utilizzato per gestire le notifiche in arrivo. I metodi *joinChat* e *exitChat* saranno usati dagli utenti per gestire l'aggiunta e la rimozione da una chat.

La classe **User** oltre a dei banali "name" e "surname", avrà come attributo un **ChatEventPrinter** che servirà nella stampa di eventuali eventi.

Infine **Message** è definita come classe concreta contenente il testo del messaggio e il mittente di quest'ultimo. Non appartiene ad una gerarchia di classi, ma potrebbe tranquillamente essere inserita in una, magari con messaggi di diverso tipo, come vocali, immagini ecc… oltre a quelli testuali. Ovviamente dovremmo poi inserire dei metodi appositi nelle classi utente per permettere a questi di inviare diversi tipi di messaggi nelle Chat(per esempio un metodo "*sendVoiceMessage"* o un metodo "*sendImage*").

## TESTING

Specifichiamo innanzitutto che i test sono stati effettuati tramite assertj, precisamente "*assertj-core-3.14.0.jar"* nella cartella *"test-libs*".

Si è deciso di testare le operazioni di TextChat(comprese ovviamente quelle definite in "Chat") e di SubCommunity. Per testare le operazioni di TextChat siamo andati, come strategia generale, a crearci degli utenti prima di ogni metodo per averli sempre pronti, questo all'interno del metodo "*before()*" con l'annotazione **@Before**.

Sono stati testati tutti i metodi rilevanti della classe, ovvero l'aggiunta/rimozione di utente, l'aggiunta di un messaggio direttamente alla lista di messaggi e la ricerca di un messaggio tramite sottostringa nella lista di messaggi. Inoltre sono stati testati anche i metodi di creazione degli iterator degli utenti e dei messaggi, la ridenominazione di una chat e il cambiamento di descrizione di questa.
Le ultime due sono state testate anche nel caso delle **SubCommunity**, tuttavia qui siamo andati anche a verificare la correttezza delle operazioni di aggiunta e rimozione di una **Community** alla Collezione di elementi, e anche qui siamo andati a verificare la corretta gestione dell'iteratore di *communities*. Come prima nel metodo *before()* siamo andati a crearci delle istanze di User e di TextChat così da non doverle istanziare manualmente ogni volta che si deve testare qualcosa di nuovo.

E' stata testata anche la classe User, e i metodi che siamo andati a testare sono ***sendMessage***, dove si è constatato che, dopo l'invocazione del metodo, i messaggi fossero effettivamente stati aggiunti alla lista di messaggi della chat. Sono stati inoltre testati i metodi joinChat e exitChat oltre a tutti metodi di test per la verifica della corretta ricezione delle notifiche riguardanti i vari eventi delle chat in cui lo user è presente.

## *• PATTERN VISITOR(SU COMPOSITE) E ADAPTER*

Si precisa che questa è la prima applicazione del pattern visitor nel progetto, ovvero quella sulla struttura Composite, infatti questo è stato applicato anche una seconda volta insieme al pattern Observer che vedremo. L'obiettivo in questo caso era definire una struttura capace

di eseguire operazioni in modo distinto su **Community** in base al suo tipo effettivo a runtime, ovvero TextChat o SubCommunity.

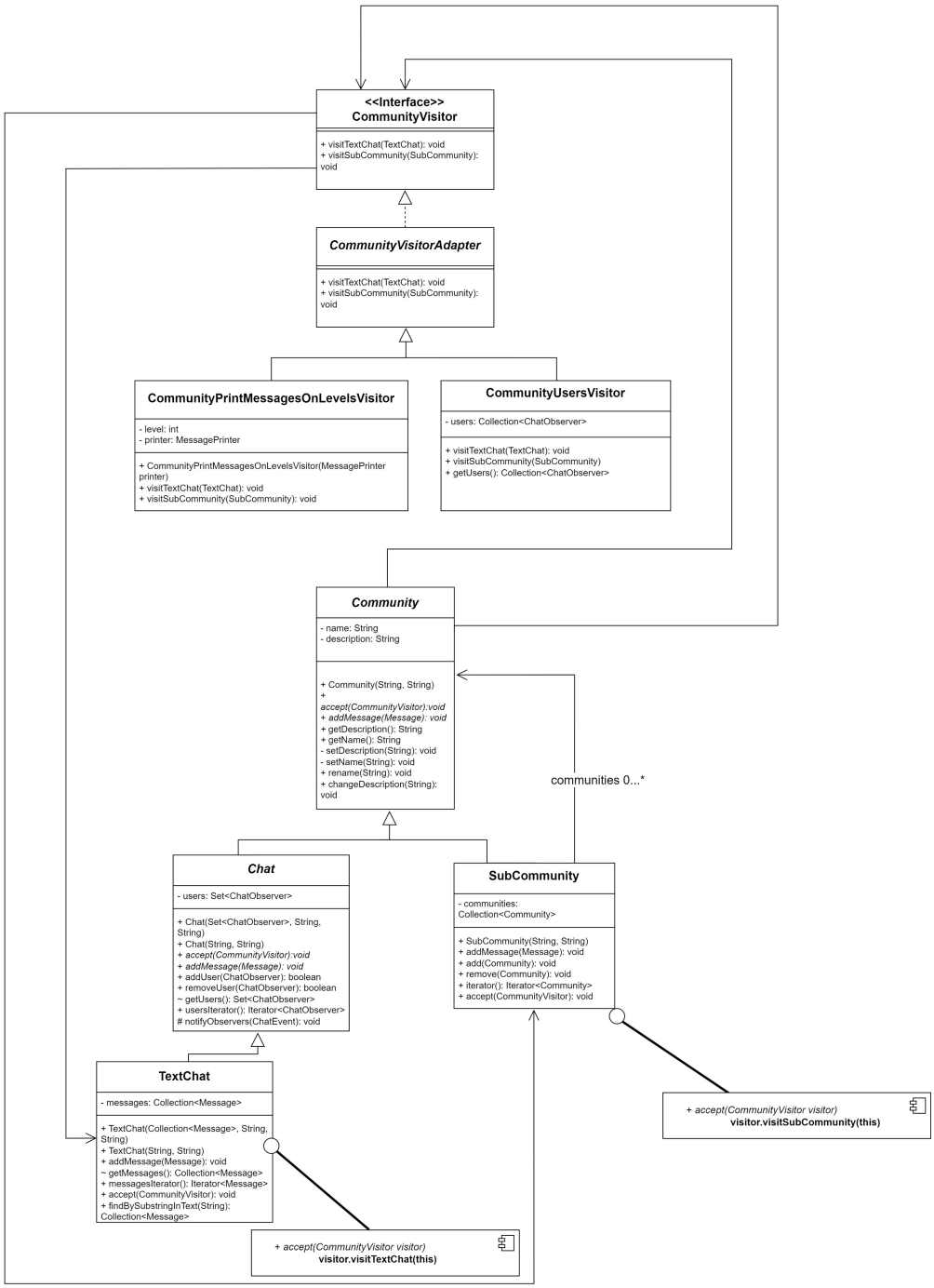
Innanzitutto è stata implementata un' interfaccia **CommunityVisitor** che definisce i metodi void astratti ***visitTextChat*** e ***visitSubCommunity*** e che funge per l'appunto da interfaccia per la struttura formata dalle varie classi Visitor.
Le operazioni dei visitor che sono state inserite sono tutte ricorsive. In base al tipo effettivo, si andrà o ad effettuare l'operazione concretamente senza continuare la "visita" in depth della struttura di classi se si parla di una foglia, o a effettuare un' operazione continuando la visita su tutti i component, nel nostro caso "*communities*", nel caso si parli di una SubCommunity.
Per rendere più pulito il codice e soprattutto meno ridontante, si è deciso di inserire una classe **Adapter**(classe astratta **CommunityVisitorAdapter**) che implementi il metodo di visita per le TextChat con un metodo vuoto, e il metodo di visita delle SubCommunity come la visita ricorsiva di tutte le *communities* della Collection. In questo modo, quando si sono andati ad implementare i due metodi dei visitor si è dovuto solo eseguire l'operazione specifica rimandando il resto alla **superclasse Adapter**:

```
@Override
public void visitTextChat(TextChat chat) {
        …emtpy…
}

@Override
public void visitSubCommunity(SubCommunity subCommunity) {
        Iterator<Community> iterator = subCommunity.iterator();
        while(iterator.hasNext()) {
                iterator.next().accept(this);
        }
}
```

Più nello specifico i visitor concreti implementati sono **CommunityUsersVisitor** e **CommunityPrintMessagesOnLevelsVisitor**. Il primo molto banalmente utilizza un set per salvare tutti gli user, ovviamente tutti distinti, e poi alla fine restituirli fornendo un metodo apposito, mentre il secondo visitor stampa i messaggi delle varie chat a livelli, in modo da poterli leggere in modo più ordinato e comprensibile. In questo caso si sono usati visitor di tipo **void**, infatti in entrambi i casi ci salviamo i valori in un'attributo privato della classe.

# UML



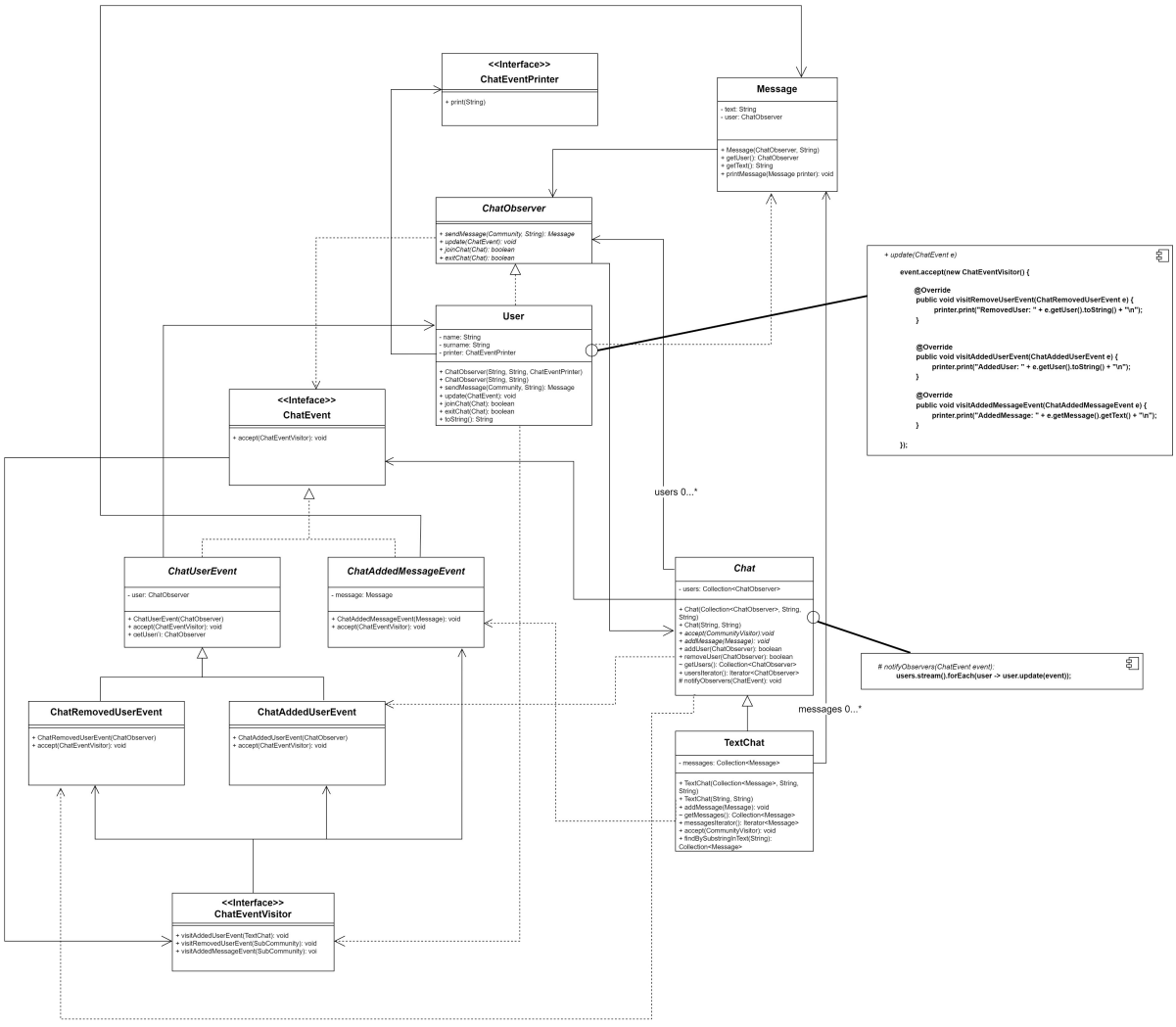

# TESTING

Per i test dei 2 visitor siamo andati a crearci delle istanze di utenti, chat e sotto community nel metodo *before()* esattamente come fatto prima. Per il primo visitor, ovvero **CommunityPrintMessagesOnLevelsVisitor**, si è testato il visitor nel caso in cui nella chat non vi fossero messaggi, e nel caso in cui ci fossero messaggi. Per le SubCommunity si è testato il caso in cui si prova a effettuare una visita su una subCommunity senza alcuna Chat/sotto-community contenuta, e nel caso in cui invece le chat ci sono. Tutti i test hanno

dato i risultati voluti. Lo stesso si è fatto per **CommunityUsersVisitor**, ovvero con e senza utenti per la chat, e con e senza sotto-community per SubCommunity.
In generale per i test, è bene notare come sia stata creata un implementazione Mock dell'interfaccia ***MessagePrinter***, interfaccia che può essere utilizzata per astrarre la stampa di un messaggio, che ha come attributo unico uno StringBuilder che "costruisce" via via la stringa da restituire.

## • *PATTERN OBSERVER*

## UML




Per fare in modo di poter notificare un utente in una chat quando avviene qualcosa in una chat, come in questo caso l'aggiunta o rimozione di un utente o l'aggiunta di un messaggio si è pensato di inserire una struttura di classi di eventi. Nell'interfaccia **ChatEvent** si è andato a inserire il metodo astratto *accept(ChatEventVisitor)* che prende in input un visitor della struttura di eventi. E' stata implementata anche una superclasse astratta

**ChatUserEvent** per rappresentare la gerarchia degli eventi sugli utenti(Notiamo che quando ci si riferisce ad “utenti” ci si riferisce non alla classe concreta “User” ma a un qualsiasi tipo di utente che estende l’interfaccia ChatObserver). Notiamo che la struttura è aperta ad espansioni future, sarà infatti possibile aggiungere nuove sottoclassi per nuovi eventi.

Il **Subject** nella struttura rappresentata è la chat, la quale conterrà, come detto prima, un Set di utenti che, oltre ad essere quelli che interagiscono con il gruppo, saranno anche gli Observer notificati dei vari eventi. La notifica dell’evento viene “generata” dalla chat stessa. Per esempio quando un utente viene aggiunto o si unisce ad una chat si dovrà, nel metodo *addUser,* notificare tutti gli utenti della collezione dell’aggiunta nuovo utente. Lo stesso viene fatto per la rimozione dell’utente in *removeUser.*
La notifica avviene in questi metodi tramite l’invocazione del metodo protected di **Chat**(protected così da rendere il suo uso disponibile alle sottoclassi) “*notifyObservers(ChatEvent)*” che invoca il metodo ***update*** di ogni user nella Collection passando come parametro il tipo di evento che si vuole notificare. Nel caso dell’aggiunta/rimozione di utente passeremo a *notifyObservers* un’istanza della classe concreta **ChatAddedUserEvent / ChatRemovedUserEvent**. Per l’aggiunta di messaggi in una TextChat invece si invocherà sempre il metodo *notifyObservers* ma passandogli un **ChatAddedMessageEvent**.

```
@Override
public void addMessage(Message message) {
        if(getUsers().contains(message.getUser())) {
                messages.add(message);
                notifyObservers(
                        new ChatAddedMessageEvent(message)
                );
        }
}
```

L’**Observer** invece è il **ChatObserver** che per l’appunto definisce il metodo astratto update da ridefinire poi nelle sottoclassi, ovvero in **User**. In User il metodo *update* è stato implementato invocando sull’evento passato come parametro il metodo *accept(ChatEventVisitor)* con come parametro un visitor di eventi anonimo che semplicemente andrà a effettuare una *print(...)* dell’evento nella forma ***“AddedUser: Tommaso Pastorelli”*** tramite un oggetto ChatEventPrinter (attributo della classe, che dovrà essere passato come parametro al costruttore), interfaccia che è stata inserita solo per fornire un astrazione per la stampa degli eventi come in questo caso.

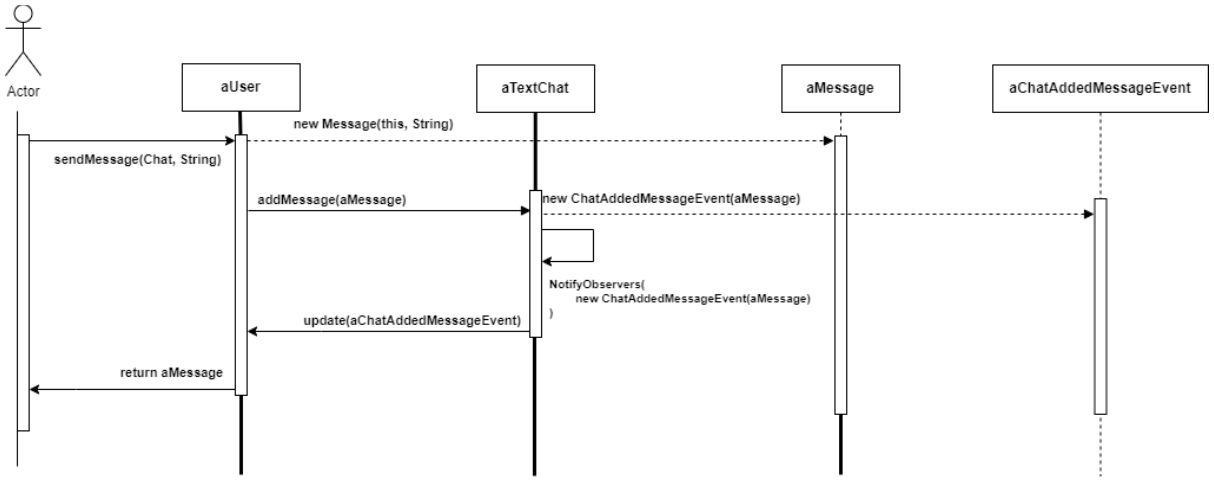
Concludiamo col dire che è stata implementata anche un interfaccia **ChatEventVisitor** che rappresenta il visitor della gerarchia di eventi, e che servirà all’observer per distinguere il tipo di evento ricevuto come notifica. L’unica implementazione che è stata implementata è una classe anonima in User.

## TESTING

I test della corretta funzionalità delle notifiche sono stati effettuati all'interno della classe per i Test degli utenti. I test effettuati sono stati 3, uno per ogni tipo di evento. In un caso si è verificato che aggiungendo due utenti ad una chat al primo fossero arrivate sia la notifica sia della propria aggiunta sia dell'aggiunta del secondo utente, lo stesso si è fatto per il metodo di rimozione. Per l'aggiunta di un nuovo messaggio alla chat si è fatto semplicemente inviare tramite il metodo *sendMessage* un messaggio a uno dei due utenti e si è verificato ove la notifica di questo fosse arrivata solo all'utente che l'ha ricevuta e non a quello che l'ha mandata.

Di seguito andiamo a rappresentare tramite un *sequence diagram* la sequenza di operazioni eseguita all'invocazione del metodo **sendMessage:**

### • *Sequence Diagram dell'operazione SendMessage:*




Notiamo che la notifica del nuovo messaggio(update) arriva anche all'utente nel diagramma, il quale lo ha inviato. Tuttavia nel programma è stato inserito un controllo per fare in modo di stampare la notifica solo nel caso in cui l'utente sia diverso da quello che ha mandato il messaggio.