

Indice

- 1) Introduzione
- 2) Requisiti
 - 2.1) Requisiti funzionali
 - 2.2) Requisiti non Funzionali
 - 2.3) Use Case Diagram
- 3) Design
 - 3.1) Class Diagram e Design pattern
 - 3.2) Sequence Diagram
 - 3.3) Activity Diagram
 - 3.4) State Diagram
- 4) Esempio di funzionamento

1) Introduzione

UpPriceFly è un'applicazione nata da un bisogno pratico. Il suo scopo è quello di monitorare i prezzi dei voli aerei e notificare eventuali variazioni.

Siccome quest'applicazione è nata affinché più utenti contemporaneamente potessero monitorare i propri voli di interesse, è stato scelto un meccanismo di funzionamento che prevede un *Admin* che gestisce gli utenti, e gli *utenti* che, una volta creati, possono autonomamente aggiungere voli da monitorare. Il funzionamento del sistema si basa sul noto servizio di messaggistica: Telegram, ma lascia aperta la possibilità di utilizzare in futuro un altro servizio di comunicazione. Ogni utente e l'Admin creano un bot Telegram e tramite questo comunicano con il sistema.

2) Requisiti

I bisogni degli utenti interessati hanno avuto un peso rilevante nella raccolta dei requisiti, essendo l'interazione con gli utenti, un elemento fondamentale per lo sviluppo di un software di alta qualità.

2.1) Requisiti funzionali:

- Gestione utenti:
 - L'admin deve poter aggiungere nuovi utenti
 - La registrazione di un utente avviene solamente attraverso l'Admin, l'utente deve chiedere all'Admin di essere aggiunto
 - Dopo che l'Admin ha aggiunto un utente, deve ricevere conferma di tale operazione
- Gestione voli:
 - L'utente deve poter aggiungere nuovi voli da monitorare
 - Dopo che l'utente ha aggiunto il volo, deve ricevere una conferma di inizio monitoraggio del volo
 - L'utente deve poter visualizzare i voli monitorati
- Monitoraggio dei voli:
 - Il sistema deve eseguire periodicamente il controllo sui prezzi dei voli, se ci sono nuovi voli da monitorare e se ci sono nuovi utenti da aggiungere
 - L'utente deve ricevere una notifica nel caso in cui il costo del volo sia aumentato o diminuito, ma non nel caso in cui il costo sia rimasto uguale
 - La notifica deve contenere il giorno e l'ora in cui è stato rilevato il cambiamento di prezzo, deve indicare a che volo si riferisce e deve ovviamente indicare il prezzo attuale e la variazione rispetto alla precedente rilevazione
 - Il sistema deve garantire il monitoraggio di voli della compagnia aerea Ryanair
- Il sistema deve consentire la comunicazione con gli utenti e l'Admin attraverso Telegram come canale principale

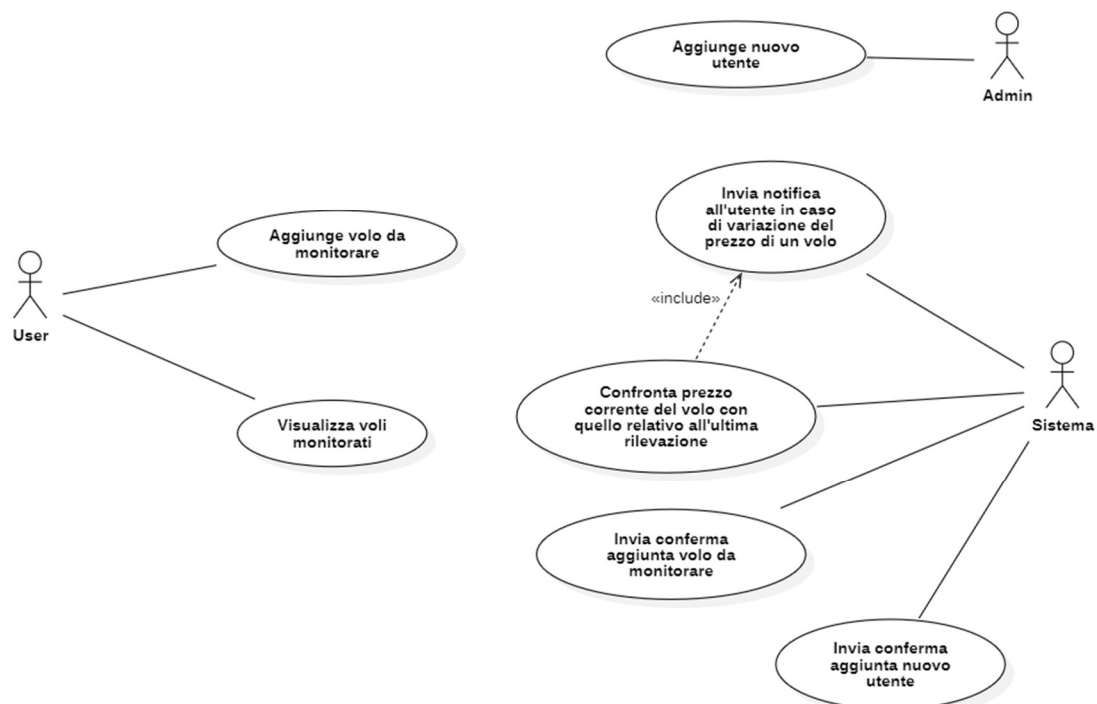
- Il sistema non prevede la possibilità per l'Admin di cancellare utenti né per gli utenti di cancellare voli da monitorare. Questa funzionalità non è stata inclusa per garantire tempi rapidi di sviluppo e rilascio.

2.2) Requisiti non funzionali:

- *Linguaggio di programmazione*: il linguaggio scelto è java
- *Affidabilità*: il sistema deve gestire il caso in cui si verifichino errori nell'invio di messaggi ed errori nella lettura dei messaggi recenti
- *Manutenibilità*: il sistema deve essere facilmente estendibile per poter aggiungere nuove compagnie aeree di cui monitorare i voli e nuovi canali di comunicazione
- *Prestazioni*: il sistema deve garantire che l'utente venga notificato immediatamente non appena venga rilevato un cambiamento nel prezzo del volo
- *Persistenza dei dati*: per semplificare e accelerare la fase di implementazione, vengono utilizzati semplici file di testo per la persistenza dei dati
- *Portabilità*: Il sistema deve essere compatibile con i principali sistemi operativi (Windows, macOS, Linux)

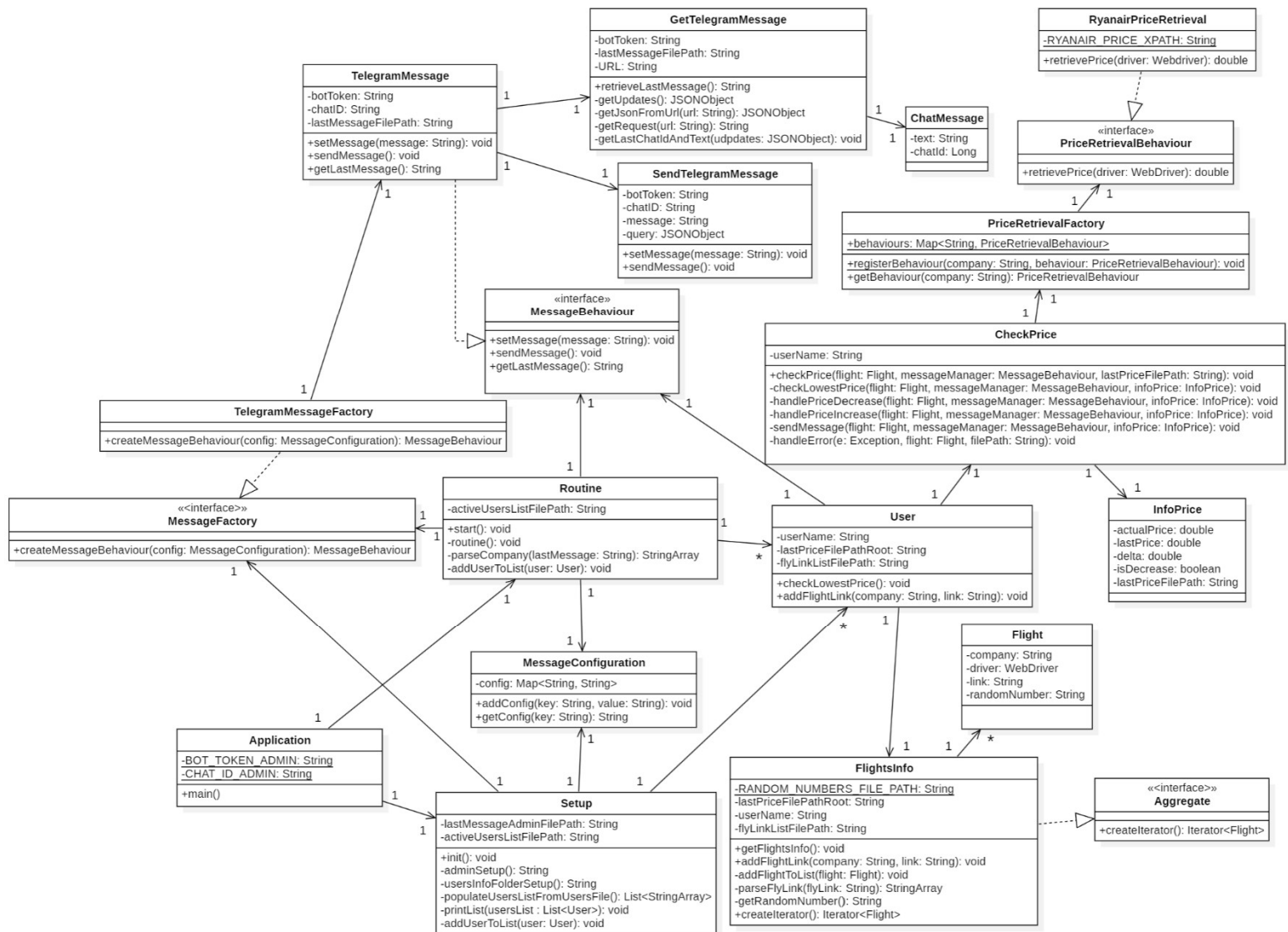
2.3) Use Case Diagram

Lo Use Case Diagram rappresenta i requisiti del sistema, fornendo una visione generale e semplificata di chi sono gli attori e quali azioni possono svolgere.



3) Design

3.1) Class Diagram e Design Pattern



Il funzionamento dell'applicazione si basa su due processi principali: l'inizializzazione e la routine, rappresentati rispettivamente dalle classi Setup e Routine.

All'avvio dell'applicazione, si entra nell'inizializzazione. In questa fase, se non sono già presenti, vengono create le cartelle e i file necessari per la memorizzazione degli utenti attivi e per la persistenza dei dati relativi all'Admin. Se sono già presenti utenti attivi, vengono create le relative istanze.

Completata l'inizializzazione, si entra nella routine, chiamata così poiché è costituita da una successione di azioni che vengono eseguite periodicamente. In questa fase, il sistema controlla:

- la presenza di un nuovo utente da aggiungere (si verifica ovvero se esiste o meno un nuovo messaggio dall'Admin, questo messaggio contiene le informazioni riguardo il nuovo utente da creare, i.e. nome del nuovo utente, bot token e chat id)
- l'esistenza di nuovi voli che gli utenti desiderano monitorare (si verifica ovvero se esiste o meno un nuovo messaggio dall'utente/i, questo messaggio contiene le informazioni riguardo il nuovo volo da monitorare, i.e. nome della compagnia e web link del volo)
- eventuali variazioni di prezzo nei voli monitorati

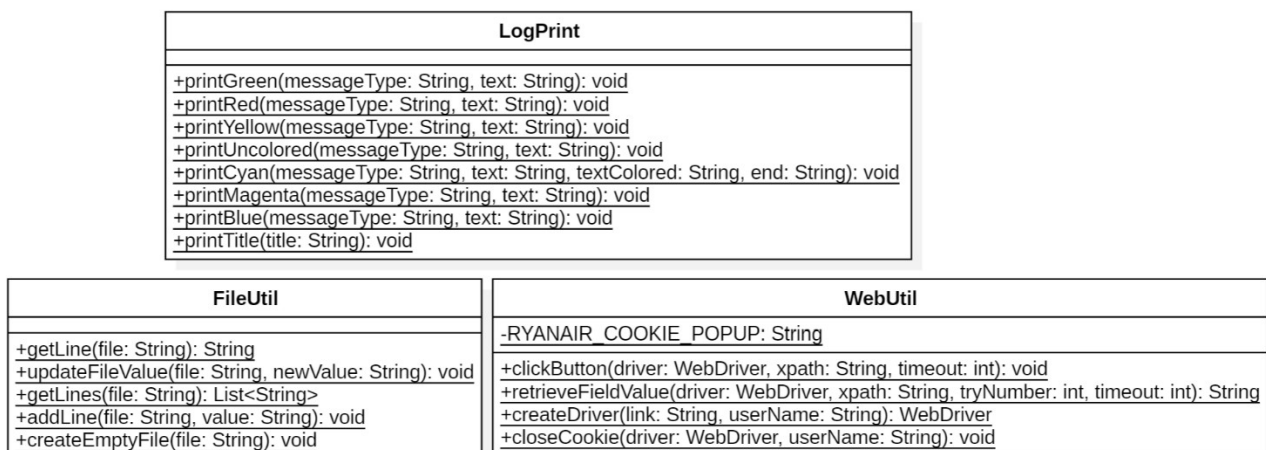
Questa fase prevede inoltre la gestione dei file necessari per memorizzare i nuovi voli e per aggiornare l'ultimo prezzo rilevato per ciascun volo. Eseguite queste operazioni, la routine si ripete a intervalli regolari.

Nell Class Diagram per semplificare la lettura, non sono stati inseriti i package esterni usati dall'applicazione, tra cui:

- Selenium, usato per fare web scraping
- java.util.concurrent, usato per eseguire periodicamente la routine
- Jansi, usato per colorare l'output nel terminale
- java.net e JSON, usati per la comunicazione attraverso Telegram

Inoltre, per semplificare la lettura sono state omesse le dipendenze da tre classi di "utilità" che contengono solamente metodi statici. Queste tre classi sono:

- LogPrint, per semplificare la stampa colorata nel terminale
- FileUtil, per operazioni su file
- WebUtil, per operazioni sul web



Per aderire ai principi SOLID in modo da sviluppare software di alta qualità, facilmente manutenibile e per venire incontro ai requisiti, sono stati utilizzati 3 design pattern:

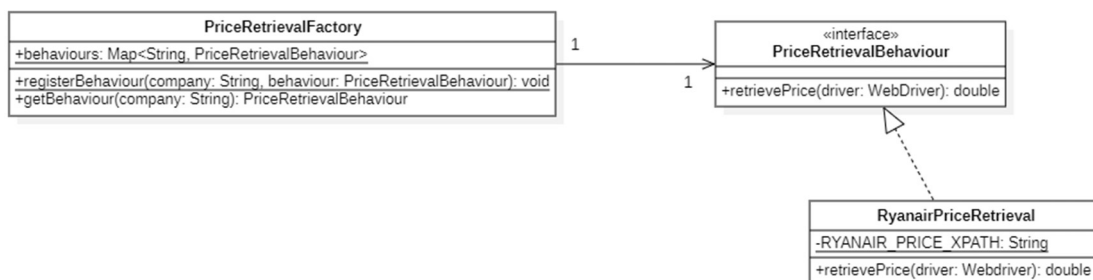
- Strategy
- Factory
- Iterator

Lo strategy pattern permette di definire una famiglia di algoritmi, incapsularli in classi separate e renderli intercambiabili. Va incontro al principio di “programmare un'interfaccia, non un'implementazione” poiché si usa un'interfaccia per rappresentare il comportamento. Rende il software aperto alle estensioni, rispettando così l'OCP. Inoltre, ogni classe in una soluzione che adotta lo Strategy Pattern ha una singola responsabilità, adempiendo al SRP. Per questi motivi questo pattern è stato usato per venire incontro al requisito per cui il sistema deve essere facilmente estendibile per poter:

- 1) *aggiungere nuove compagnie aeree* di cui monitorare i voli
- 2) *nuovi canali di comunicazione*

Il factory pattern gestisce i dettagli della creazione di oggetti. Questo permette di evitare hard dependencies nel codice che andrebbero contro l'OCP. Bisogna “programmare un'interfaccia non un'implementazione”. Rispetta anche il SRP in quanto la logica di creazione è separata dalla logica di business e il DIP poiché le classi dipendono da interfacce o astrazioni e non da implementazioni concrete.

1)

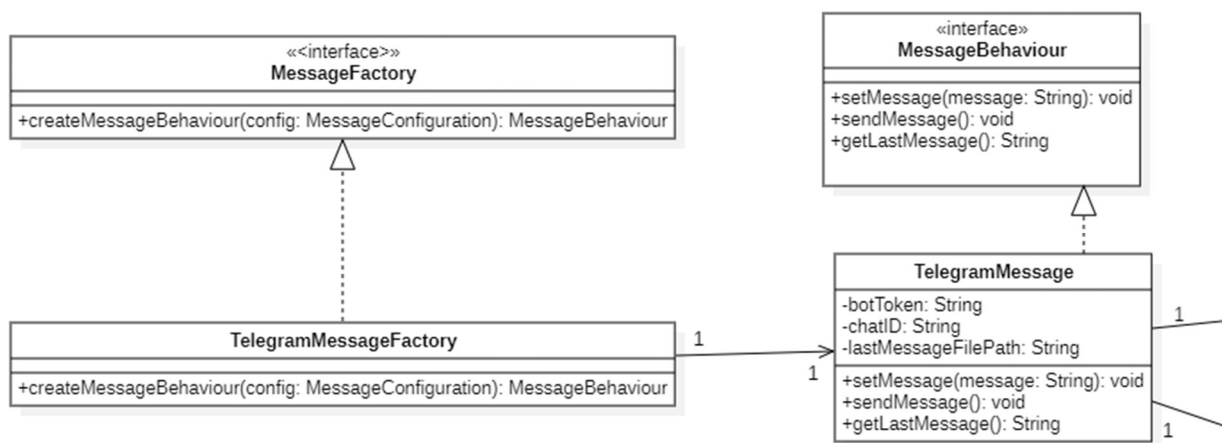


Nel primo caso è stata realizzata l'interfaccia **PriceRetrievalBehaviour** con il metodo `retrievePrice`. Ogni compagnia aerea (**ConcreteStrategy**) si servirà di un algoritmo diverso per estrarre il prezzo del volo. Inoltre, si è implementato un simple factory per gestire i dettagli della creazione delle **ConcreteStrategy**. Questo è stato realizzato utilizzando una mappa static (chiamata `behaviours`) con le associazioni tra le compagnie aeree e i rispettivi **ConcreteStrategy**. Attraverso il metodo static `registerBehaviour` si registrano i nuovi algoritmi, e `getBehaviour`, data la compagnia in input, ritorna il **ConcreteStrategy**.

Per rispettare il più possibile l'OCP, la mappa viene definita nel più basso livello, ovvero nel main; quando si aggiunge un nuovo algoritmo, non bisogna modificare la logica del factory, bensì aggiungere nel main una nuova entry alla mappa.

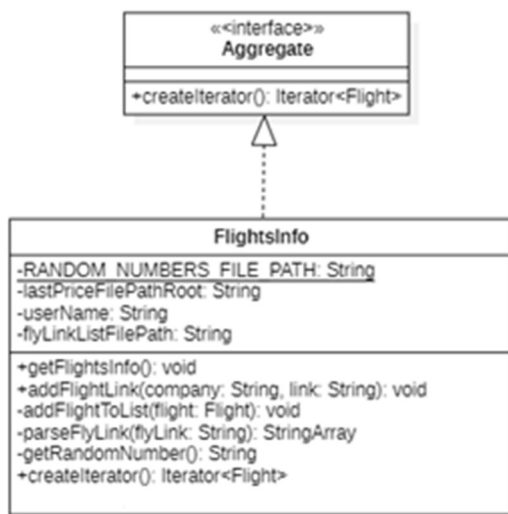
```
public static void main(String[] args) {  
  
    PriceRetrievalFactory.registerBehaviour( company: "RYANAIR", new RyanairPriceRetrieval());  
  
}
```

2)



Nel secondo caso, è stato implementato uno strategy caratterizzato dall'interfaccia **MessageBehaviour** con i tre metodi fondamentali per la comunicazione con gli utenti e con l'Admin. Se si vorrà usare un canale di comunicazione diverso da Telegram (per es. Email), basterà creare una diversa implementazione dell'interfaccia. Per evitare di implementare tutta la logica di funzionamento di **TelegramMessage** nella classe stessa, si è divisa l'implementazione in due classi diverse (qui non riportate): una con la logica per l'invio di messaggi e l'altra con la logica per la lettura dell'ultimo messaggio. Inoltre, è stato utilizzato un factory method per gestire la creazione dei **MessageBehaviour**. Il factory method definisce un'interfaccia, nel nostro caso **MessageFactory**, per la creazione di oggetti, ma lascia le sottoclassi decidere che classi istanziare. Questo è essenziale dal momento che la logica dietro l'invio e ricezione dei messaggi sarà diversa per ogni canale di comunicazione e perciò anche la creazione del relativo oggetto sarà diversa, avremo quindi diverse implementazioni di **MessageFactory**, ognuna corrispondente ad un **MessageBehaviour**. **MessageConfiguration** è una classe contenente una mappa che viene usata da contenitore per i dati necessari alla creazione del **MessageBehaviour**.

Infine, è stato utilizzato anche l'Iterator Pattern per la classe FlightsInfo. L'iterator pattern permette di navigare tra gli elementi di un oggetto senza conoscere la sua struttura interna. Nel nostro caso, questo ci permette di iterare sulla collection di flight presente in FlightsInfo. Dal momento che la collection usata è una list, questa implementa già l'interfaccia Iterator; dunque, ci limitiamo a implementare il metodo createIterator in FlightsInfo. Separando la logica per iterare su una collezione dalla logica per memorizzare dati della collezione, si rispetta il SRP.



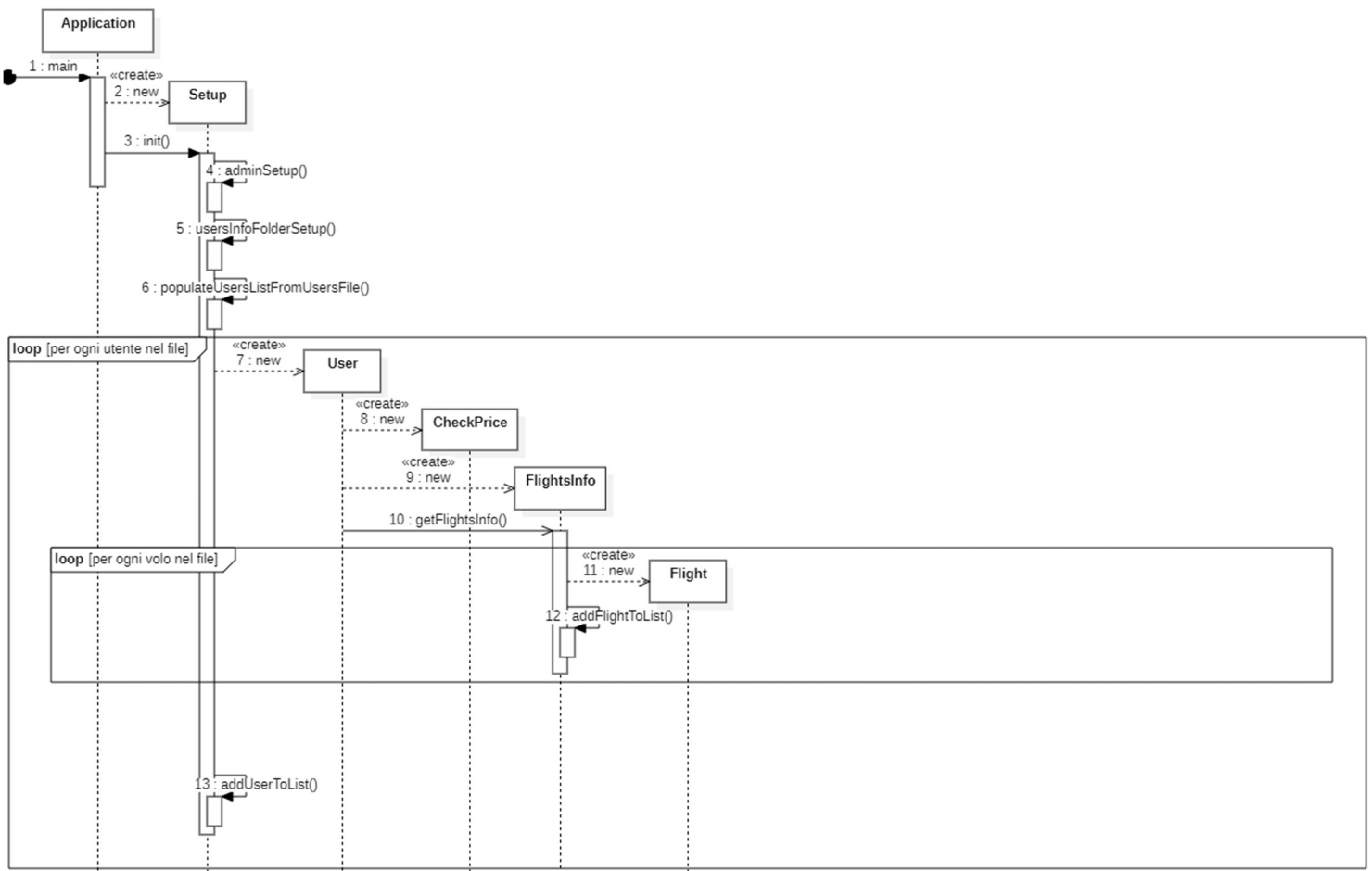
```
public Iterator<Flight> createIterator() {
    return flightsInfoList.iterator();
}
```

Come si può osservare, l'architettura dell'applicazione è stata progettata in modo tale da minimizzare l'accoppiamento, massimizzando la coesione (grazie anche all'uso dei design pattern).

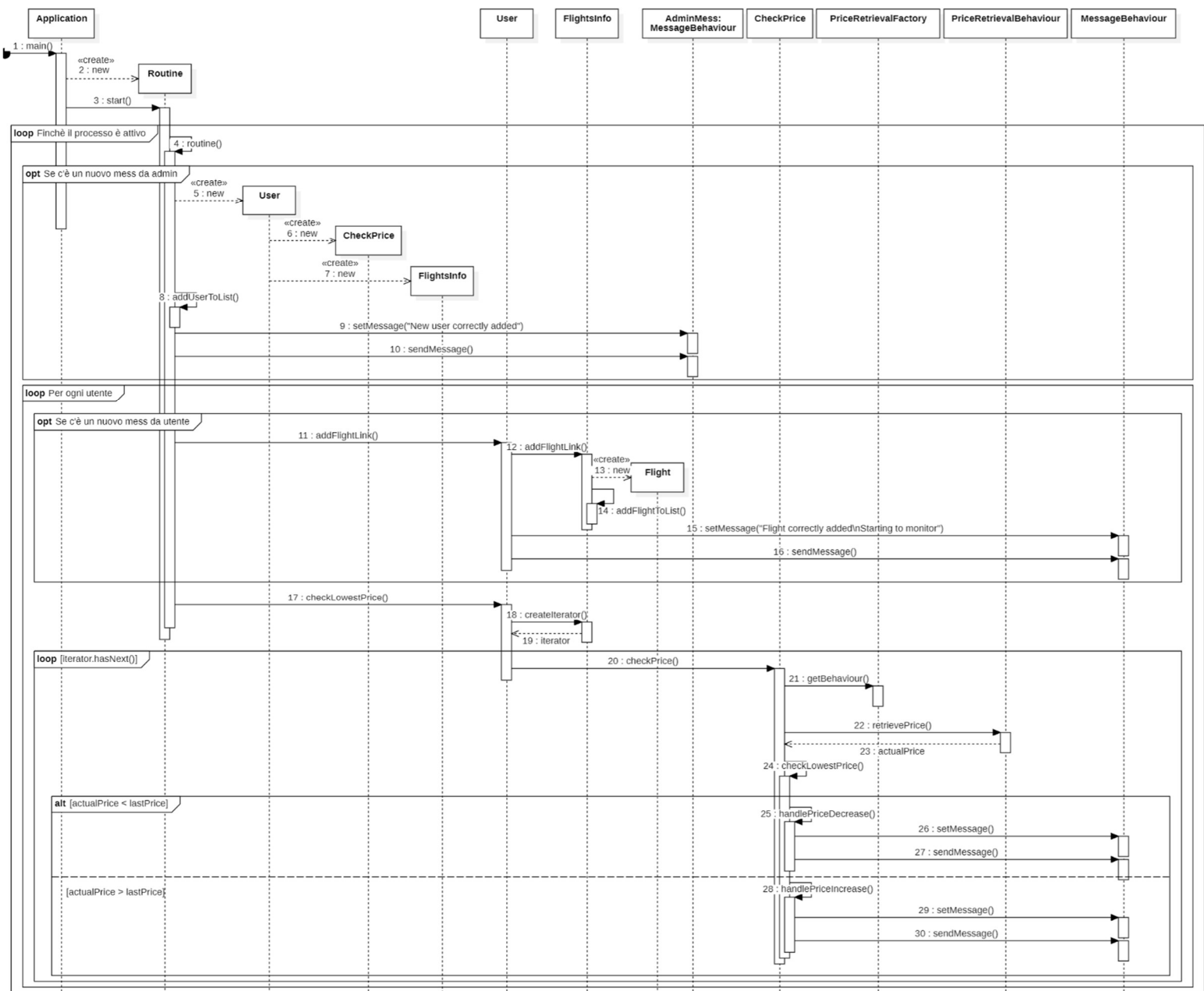
3.2) Sequence Diagram

Di seguito due sequence diagram che rappresentano come gli oggetti interagiscono tra di loro nelle due diverse fasi dell'applicazione: l'inizializzazione e la routine.

Inizializzazione



Routine

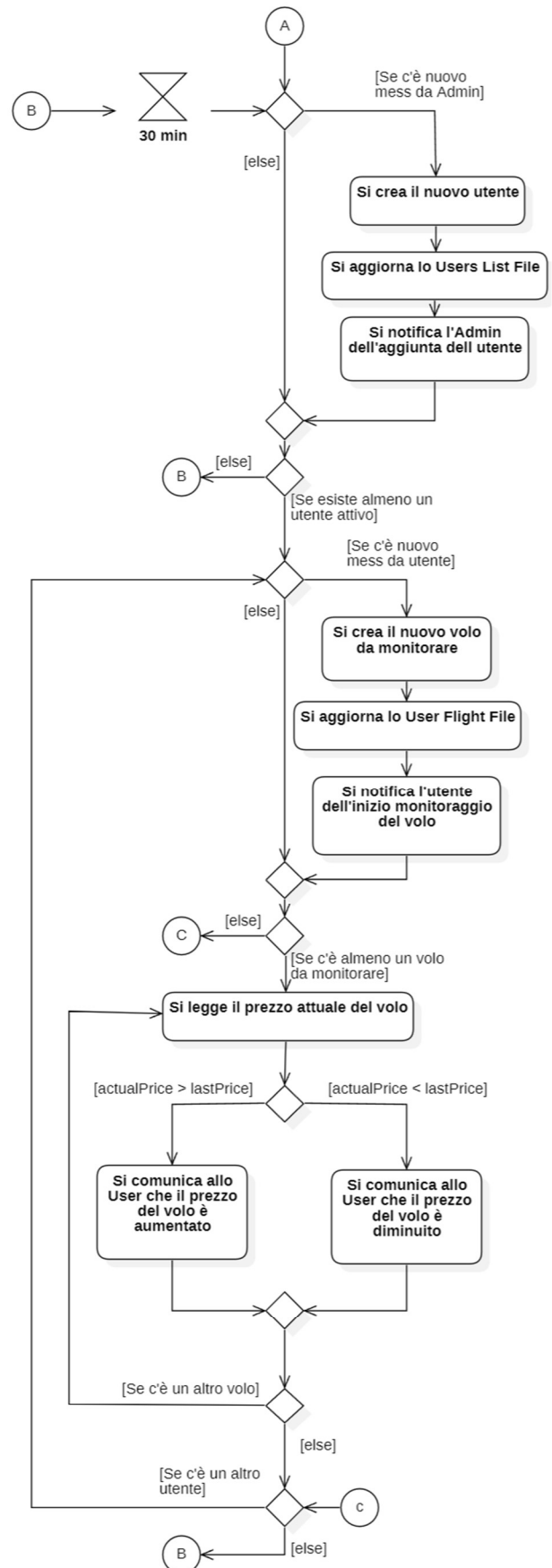
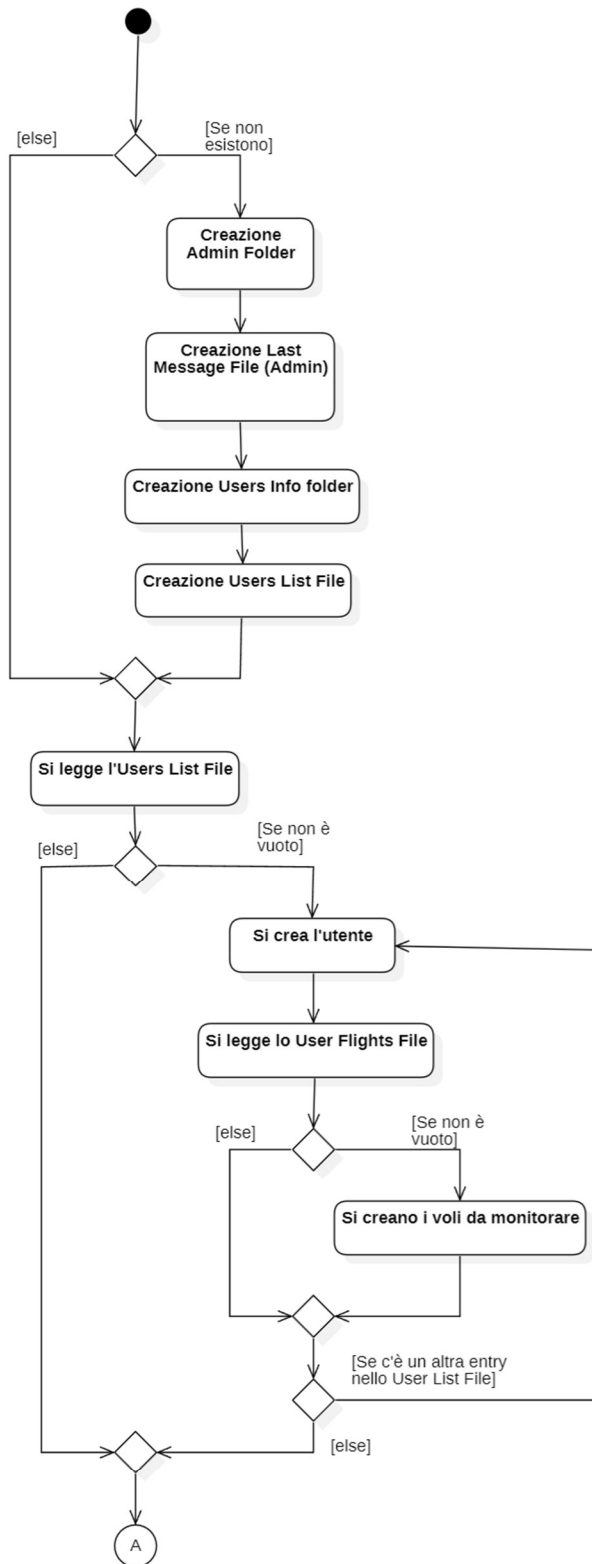


In questo secondo diagram è stato utilizzato un loop per indicare la ripetizione periodica.
In realtà, come già detto, questo compito è svolta da `java.util.concurrent`

Per semplificare la lettura, non sono stati indicati i parametri nelle chiamate ai metodi e sono state omesse tutte le chiamate ai metodi statici relativi alla gestione dei file, all'interazione con il web e alla stampa sul terminale.

3.3) Activity Diagram

Di seguito l'activity diagram dell'applicazione. L'activity diagram mostra l'ordine sequenziale delle attività, decisioni e flussi all'interno di un sistema.



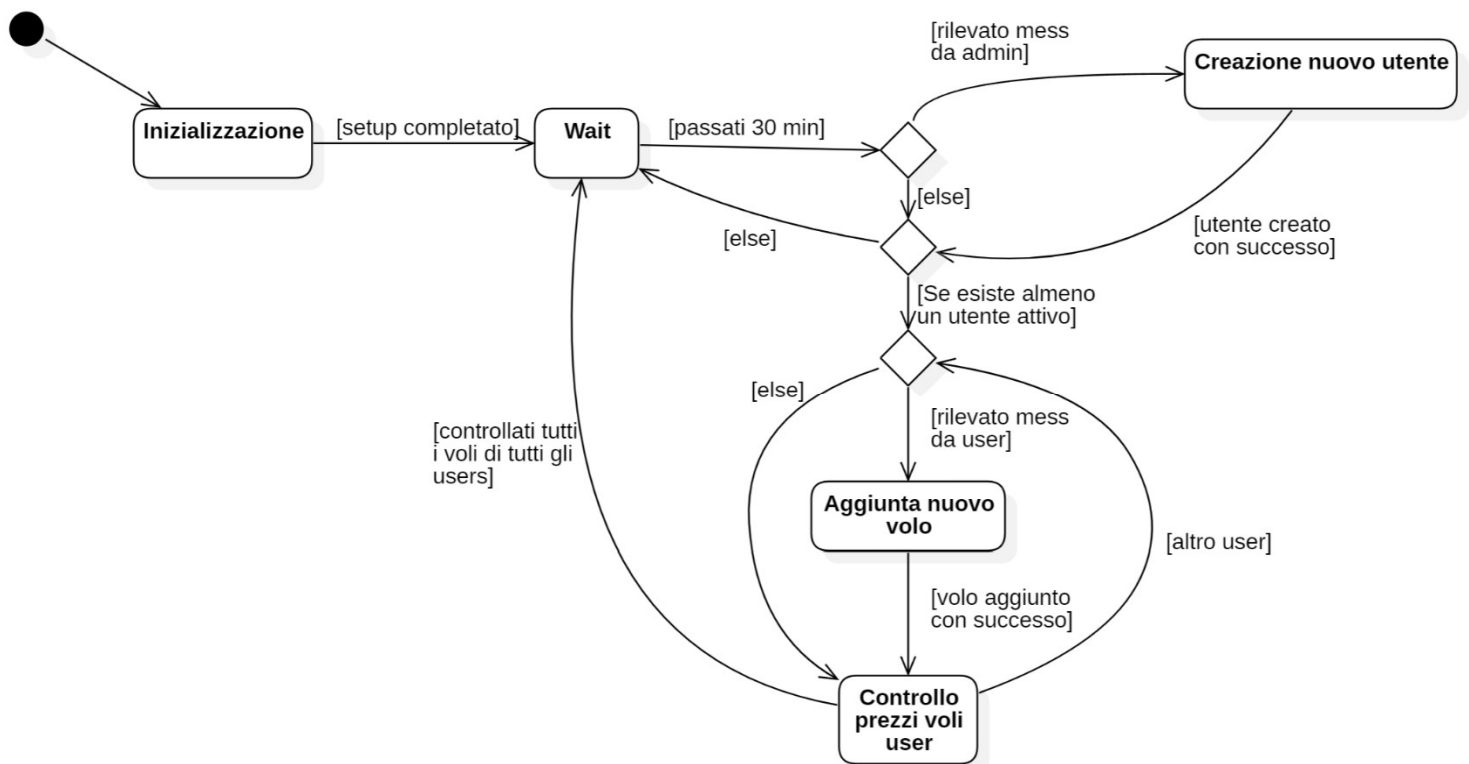
La parte di sinistra rappresenta l'inizializzazione, mentre quella di destra la routine.

Nel diagramma la routine si ripete ogni 30 min. Questo intervallo è stato scelto a piacere, non essendo specificato in nessun requisito.

Il Last Message File è fondamentale in quanto viene memorizzato l'ultimo messaggio processato. Quando si legge l'ultimo messaggio dalla chat, lo si confronta con il contenuto di questo file e se sono diversi allora si ha un nuovo messaggio da processare. Ogni utente e l'Admin hanno un proprio Last Message File.

3.4) State Diagram

Di seguito lo state diagram dell'applicazione. Lo state diagram rappresenta gli stati che un sistema può assumere e le transizioni tra questi stati in risposta a eventi o condizioni.



Per semplificare lo state diagram, non è stata rappresentata l'esecuzione della routine subito dopo l'inizializzazione.

Nel diagramma la routine si ripete ogni 30 min. Questo intervallo è stato scelto a piacere, non essendo specificato in nessun requisito.

4) Esempio di funzionamento

Simuliamo l'esecuzione dell'applicazione su un sistema windows dove non è mai stata eseguita prima.

Nel main, si setta la mappa dei price retrieval behavior e il canale di comunicazione.

```
PriceRetrievalFactory.registerBehaviour( company: "RYANAIR", new RyanairPriceRetrieval());
```

```
setup.setMessageFactory(new TelegramMessageFactory());
```

```
routine.setMessageFactory(new TelegramMessageFactory());
```

Si avvia l'applicazione.

```
ADMIN SETUP
INFO --> Creating Admin folder
INFO --> Creating last message file
INFO --> Successfully Admin setup

USER INFO FOLDER SETUP
INFO --> Creating Users Info folder
INFO --> Creating users list file
INFO --> Successfully Users info folder setup

POPULATING USER LIST FROM FILE
INFO --> Found 0 active users

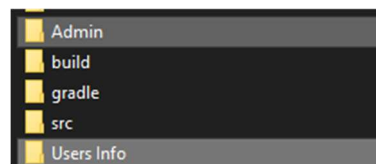
STARTING THE ROUTINE

INFO --> Last Check: 16-11-2024 17:49:04
INFO --> Found new message =
Giulio
6837655438:AAH2twY_I5U8VFykeyWf3IMsoDMJyx9bnk
1165808562
INFO --> Message sent correctly
INFO --> New user found, added to users list file

USER = Giulio

WARNING --> No recent message found
```

Essendo la prima esecuzione, vengono creati tutti i folders e files necessari



Supponiamo che l'admin aggiunga l'utente Giulio



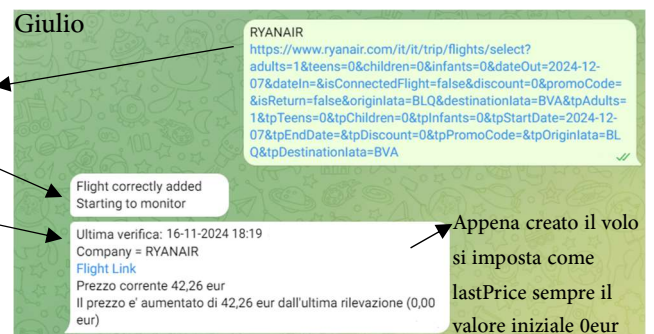
1 iterazione

```
INFO --> Last Check: 16-11-2024 18:19:04
INFO --> No recent message found

USER = Giulio

INFO --> Found new message =
RYANAIR
https://www.ryanair.com/it/it/trip/flights/select?adults=1&teens=0&children=0&infants=0&dateOut=2024-12-07&dateIn=2024-12-07&isConnectedFlight=false&discount=0&promoCode=&isReturn=false&originIata=BLQ&destinationIata=BVA&tpAdults=1&tpTeens=0&tpChildren=0&tpInfants=0&tpStartDate=2024-12-07&tpEndDate=2024-12-07&tpDiscount=0&tpPromoCode=&tpOriginIata=BLQ&tpDestinationIata=BVA
INFO --> User = Giulio
Correctly added link=https://www.ryanair.com/it/it/trip/flights/select?adults=1&teens=0&children=0&infants=0&dateOut=2024-12-07&dateIn=2024-12-07&isConnectedFlight=false&discount=0&promoCode=&isReturn=false&originIata=BLQ&destinationIata=BVA&tpAdults=1&tpTeens=0&tpChildren=0&tpInfants=0&tpStartDate=2024-12-07&tpEndDate=2024-12-07&tpDiscount=0&tpPromoCode=&tpOriginIata=BLQ&tpDestinationIata=BVA
INFO --> Message sent correctly
INFO --> user=Giulio company=RYANAIR link=https://www.ryanair.com/it/it/trip/flights/select?adults=1&teens=0&children=0&infants=0&dateOut=2024-12-07&dateIn=2024-12-07&isConnectedFlight=false&discount=0&promoCode=&isReturn=false&originIata=BLQ&destinationIata=BVA&tpAdults=1&tpTeens=0&tpChildren=0&tpInfants=0&tpStartDate=2024-12-07&tpEndDate=2024-12-07&tpDiscount=0&tpPromoCode=&tpOriginIata=BLQ&tpDestinationIata=BVA
Price increased by 42,26€ since last check
INFO --> Message sent correctly
```

Supponiamo che l'utente Giulio aggiunga un volo RYANAIR da tenere monitorato



Appena creato il volo si imposta come lastPrice sempre il valore iniziale 0eur

2 iterazione

3 iterazione

```
INFO --> Last Check: 16-11-2024 18:49:04
INFO --> No recent message found


USER = Giulio



INFO --> No recent message found
INFO --> user=Giulio company=RYANAIR link=https://www.ryanair.com/it
The price has not changed
actual_price=42.26 == 42.26=last_price
```


Nel caso in cui il prezzo fosse cambiato sarebbe stato inviato un messaggio simile a quello iniziale con lastPrice uguale a 0eur.

Ci fermiamo alla terza iterazione.


Il volo scelto da monitorare è:

 Ordina voli per ▾

 05 dic Giovedì 39,83 €	06 dic Venerdì 32,99 €	07 dic Sabato 42,26 €	08 dic Domenica 32,99 €	09 dic Lunedì 
--	------------------------------	-----------------------------	-------------------------------	---

 Operato da
Malta Air

06:00
Bologna

FR 3214
 1 h 55 m

07:55
Parigi
(Beauvais)

Tariffa Basic
42,26 €

Seleziona