DStack

*dstackgroup@gmail.com*

# Allegato Tecnico

## Informazioni sul documento

| | |
|---|---|
| **Nome documento** | Allegato Tecnico |
| **Versione** | v1.0.0 |
| **Data approvazione** | 2019-04-04 |
| **Responsabili** | Federico Rispo |
| **Redattori** | Niccolò Vettorello<br>Enrico Trinco<br>Eleonora Signor<br>Harwinder Singh<br>Alberto Schiabel<br>Federico Rispo<br>Elton Stafa |
| **Verificatori** | Alberto Schiabel<br>Eleonora Signor |
| **Stato** | Approvato |
| **Lista distribuzione** | Gruppo *DStack*<br>*Prof. Riccardo Cardin* |
| **Uso** | Esterno |

## Sommario

Il presente documento contiene le scelte architetturali che del gruppo *DStack* per la realizzazione del progetto Butterfly. Comprende i design pattern e i diagrammi di attività, sequenza, classi e package.

# Indice



# Elenco delle figure

# 1 Introduzione

## 1.1 Scopo del documento

Lo scopo del presente documento é quello di descrivere in maniera coesa, coerente ed esaustiva le caratteristiche tecniche salienti del prodotto *Butterfly*$_{\mathbf{G}}$, sviluppato dal gruppo *DStack*. Nella sua parte conclusiva l'*Allegato Tecnico*$_{\mathbf{G}}$ pone in essere un confronto tra lo stato del prodotto cosí come presentato in sede di *Technology Baseline* e lo stato attuale (*Product Baseline*), e fornisce una panoramica su casi d'uso e requisiti soddisfatti.

## 1.2 Scopo del prodotto

Butterfly nasce dall'esigenza di uniformare e accentrare la gestione delle segnalazioni generate a partire da sistemi di terze parti, quali Redmine, GitLab e SonarQube. Questi strumenti sono parte integrante dei processi gestionali, di versionamento e di Continuous Integration dell'azienda committente. La maggior parte di essi fornisce già dei meccanismi di notifica ed inoltro delle possibili segnalazioni, sono configurabili e accessibili da dashboard molto diverse tra loro, di difficile interazione e anche con limitazione di accessibilità. Inoltre, in caso di segnalazioni di bug in ambienti di produzione è fondamentale assicurarsi che gli sviluppatori in grado di risolvere il problema siano segnalati tempestivamente, senza aspettare che loro accedano a qualche dashboard specifica. Il gruppo *DStack* si propone quindi di sviluppare una rete di soluzioni che offrano un'interfaccia condivisa, estendibile per gestire le segnalazioni relative alla pipeline di sviluppo software di *Imola Informatica S.P.A.*. Questa interfaccia deve inoltre permettere una configurazione automatica e personalizzabile di tali segnalazioni.

## 1.3 Glossario

All'interno del documento sono presenti termini che possono presentare significati ambigui o incongruenti a seconda del contesto. Per evitare questo tipo di problema viene allegato glossario nel file *Glossario v1.0.0*, che contiene tali termini e la loro spiegazione. Nella seguente documentazione viene indicata in corsivo e seguita da una "G" a pedice solo la prima occorrenza dei termini presenti nel glossario, per favorire maggiore chiarezza ed evitare ridondanza.

## 1.4 Riferimenti informativi

- *Analisi dei Requisiti v3.0.0*;
- Ingegneria del Software A.A. 2018/2019, lezione "Software Architecture Patterns"
  https://www.math.unipd.it/ rcardin/sweb/2019/L03.pdf

## 1.5 Riferimenti normativi

- *Norme di Progetto v3.0.0*;
- Capitolato d'appalto C1: Butterfly
  https://www.math.unipd.it/ tullio/IS-1/2018/Progetto/C1.pdf

# 2 Architettura del prodotto

## 2.1 Descrizione generale

In fase di progettazione, il gruppo *DStack* ha deciso di modellare l'architettura generale del progetto *Butterfly* come una rete di *microservizi*$_{\mathbf{G}}$, la cui interazione con l'utente avviene tramite *API REST*$_{\mathbf{G}}$, e la cui comunicazione interna si basa sul Messaging Pattern "*Publish/Subscribe*$_{\mathbf{G}}$". Quest'ultimo è stato applicato nella sua accezione *topic*$_{\mathbf{G}}$-based, utilizzando *Apache Kafka*$_{\mathbf{G}}$ come *Broker*$_{\mathbf{G}}$ dei messaggi.
Il pattern *Publish/Subscribe* (abbreviato spesso in *pub/sub*) definisce tre ruoli distinti[1]:

- **Publisher**: rappresenta un componente che si occupa di inviare o generare messaggi di interesse per il resto del sistema. Non ha alcuna conoscenza di chi effettivamente riceverà il contenuto, ma si limita a generarlo e a classificarlo per *topic*;
- **Broker**: lo scopo di questo componente è quello di fungere da coda di messaggi: riceve i contenuti generati dai *Producers* e li mette a disposizione dei Subscribers;
- **Subscriber**: ruolo ricoperto dalle parti del sistema interessate a ricevere i contenuti inviati dai *Producers*.

Tale paradigma é stato scelto per i seguenti motivi:

- **Dominio del problema**: il progetto *Butterfly* consiste nella realizzazione di un sistema di comunicazione e scambio di messaggi tra microservizi, ed il pattern *Publish/Subscribe* approccia e risolve elegantemente proprio questo tipo di problemi;
- **Forte disaccoppiamento**: i *Producers* sono ignari dell'esistenza dei *Consumers*, ed entrambi i componenti, durante la loro esecuzione, non hanno bisogno di conoscere la topologia generale del sistema. A livello pratico la comunicazione avviene in maniera asincrona, permettendo così l'operatività continua delle parti;
- **Scalabilitá**: il forte disaccoppiamento tra le componenti stateless, la possibilità di eseguire contemporaneamente più istanze degli stessi servizi e la comunicazione asincrona tramite un sistema a coda rendono il sistema fortemente scalabile.

## 2.2 Diagramma dei package

Segue il diagramma dei package che rappresenta l'architettura nel suo complesso.

[1] Nel contesto del progetto ci riferiremo ai componenti che assumono il ruolo di *Publisher* come *Producers*$_{\mathbf{G}}$, e a quelli che ricoprono il ruolo di Subscriber come *Consumers*$_{\mathbf{G}}$

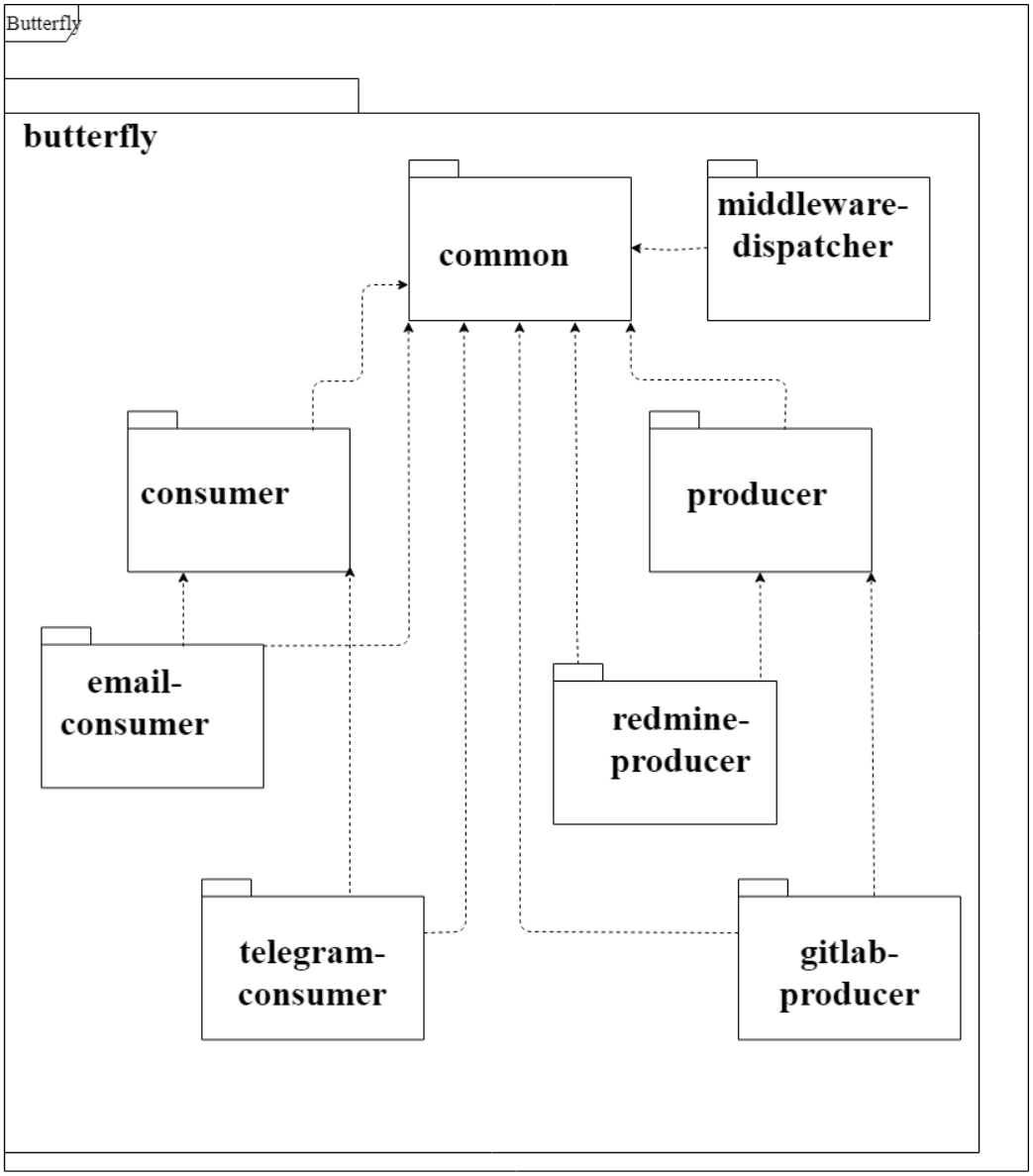


**Figura 1:** Diagramma package architettura generale

## 2.3 Diagramma di sequenza generale

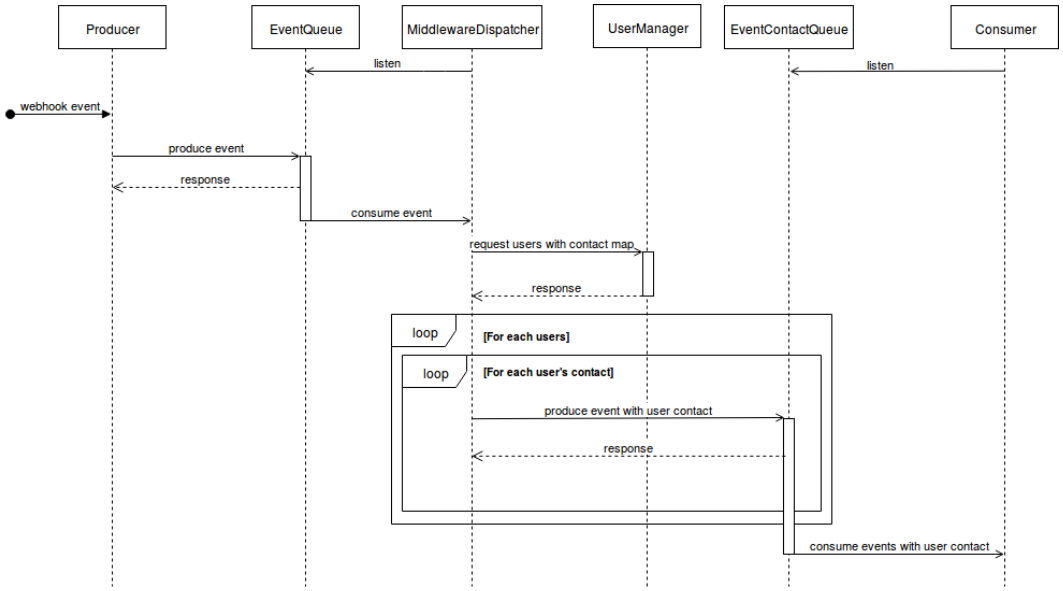


**Figura 2:** Diagramma di sequenza generale

## 2.4 Design Patterns notevoli utilizzati

Oltre al modello *Publish/Subscribe* descritto sopra, nella realizzazione del progetto sono stati impiegati anche i seguenti patterns:

- **Template Method Pattern**: descritto nel dettaglio in sezione §2.7;
- **Command Pattern**: descritto nel dettaglio in sezione §2.6.2.5.2;
- **Strategy Pattern**: descritto nel dettaglio in sezione §2.6.2.4;
- **Observer Pattern**: descritto nel dettaglio in sezione §2.6.2.4.

## 2.5 Architettura User Manager

### 2.5.1 Descrizione

Il *Gestore Personale*$_{\mathbf{G}}$ (User Manager in inglese) é il servizio incaricato di esporre *API REST* per la creazione e la gestione di utenti, progetti, e delle iscrizioni degli utenti ai possibili eventi dei suddetti progetti. Tutte le comunicazioni con il Gestore Personale avvengono tramite richieste *REST* il cui contenuto è in formato *JSON*$_{\mathbf{G}}$. Il Gestore Personale è l'unico servizio che ha accesso diretto al database SQL che contiene i dati inseriti dagli utenti. Lo User Manager è inoltre usato dal Middleware Dispatcher per determinare, a partire da un evento serializzato in *JSON*, quali siano gli utenti che dovranno essere informati di tale evento e attraverso quali piattaforme.

### 2.5.2 Diagrammi dei package

Seguono i diagrammi dei package relativi allo User Manager.

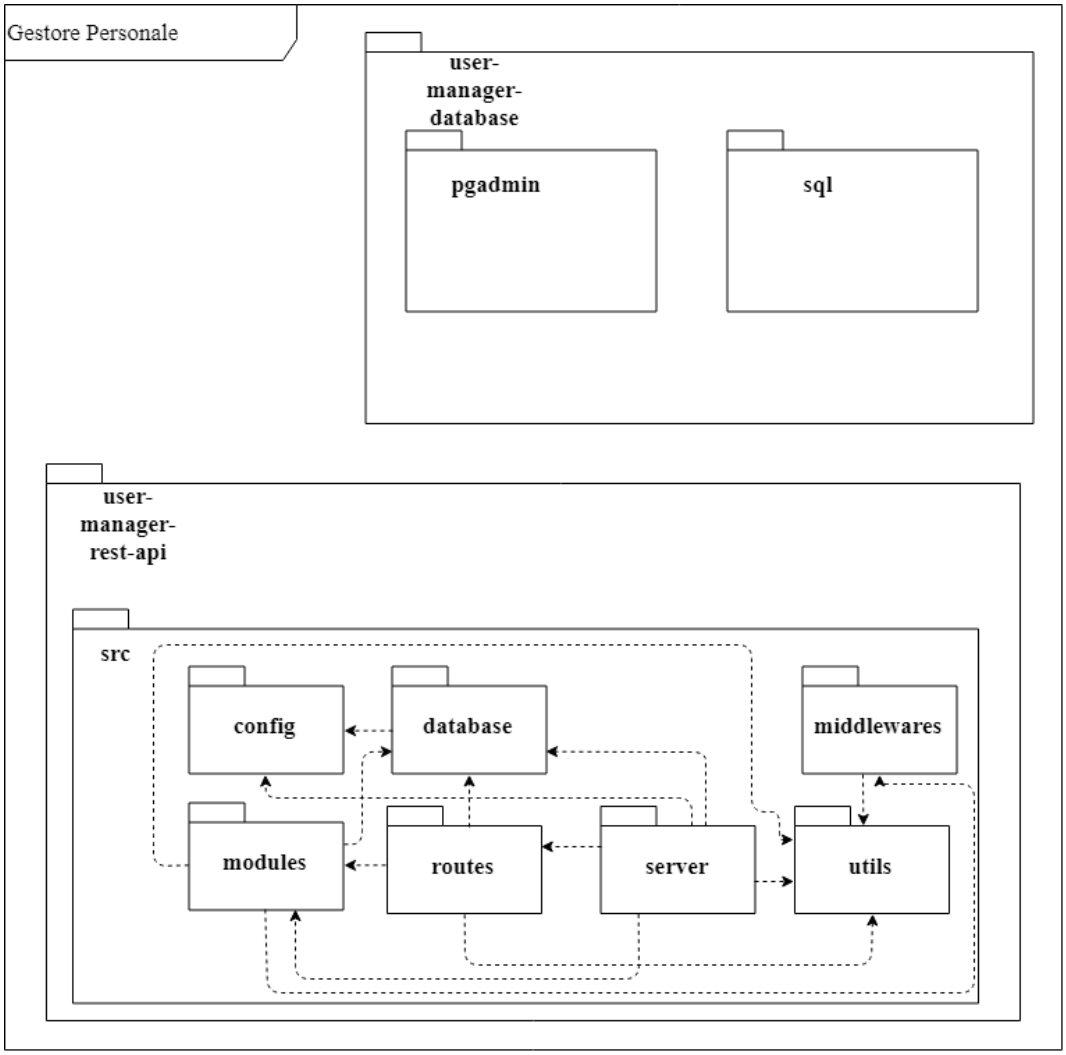


**Figura 3:** Diagramma package Gestore Personale

## 2.6 Architettura servizi di Produzione e Consumo

Con "servizi di Produzione e Consumo" indichiamo i componenti del sistema che svolgono i ruoli di *Publisher* e *Subscriber* rispettivamente: i primi sono incaricati di inoltrare gli eventi ricevuti dai servizi di produzione (Redmine, GitLab, SonarQube), mentre i secondi ricevono le informazioni dell'evento corredate dalle informazioni di contatto degli utenti che devono riceve notifiche degli eventi prodotti.
Per ampliare i dati dell'evento in ingresso abbiamo impiegato il servizio chiamato Middleware Dispatcher che richiede al Gestore Personale, tramite interfaccia REST, la lista di utenti con i relativi sistemi di contatto a cui inoltrare l'evento stesso. Nei paragrafi seguenti la loro architettura viene illustrata piú dettagliatamente: é presente anche una descrizione dei Design Patterns piú importanti utilizzati e del motivo per cui sono stati scelti.

### 2.6.1 Producer

#### 2.6.1.1 Descrizione

I *Producer* sono i componenti del sistema che generano i contenuti di interesse. Essi sono responsabili della creazione dei messaggi e del loro invio al Message Broker

*Apache Kafka*. All'interno di *Butterfly* esistono tre tipi di *Producer*, uno per ogni tipo di servizio d'interesse:

1. Gitlab Producer;
2. Redmine Producer;
3. Sonarqube Producer.

#### 2.6.1.2 Diagrammi dei package

Seguono i diagrammi dei package relativi ai Producer.

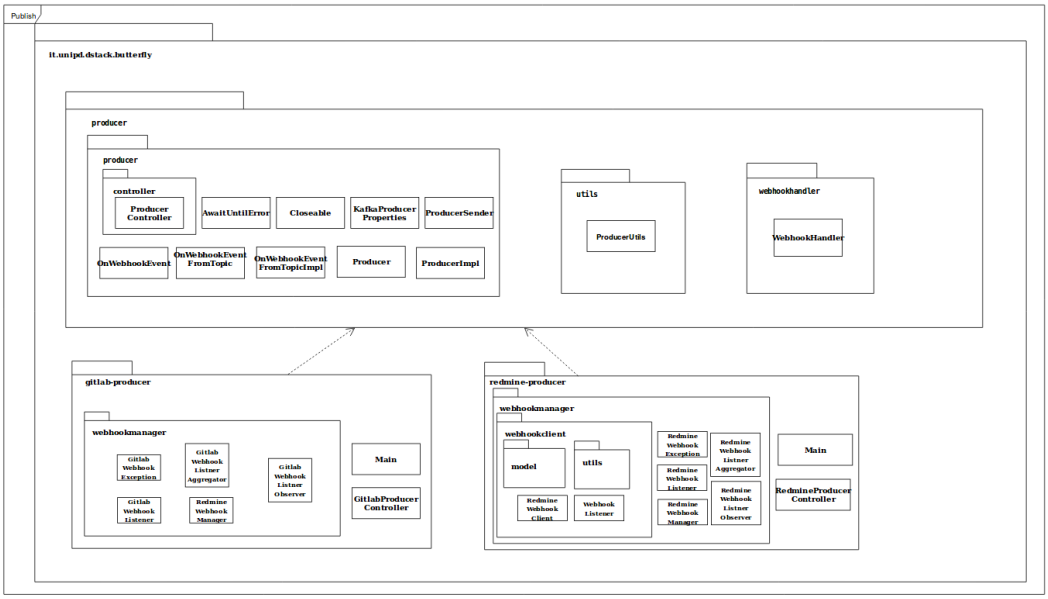


**Figura 4:** Diagramma package Producer

#### 2.6.1.3 Diagrammi di sequenza

Seguono i diagrammi di sequenza relativi ai Producer.

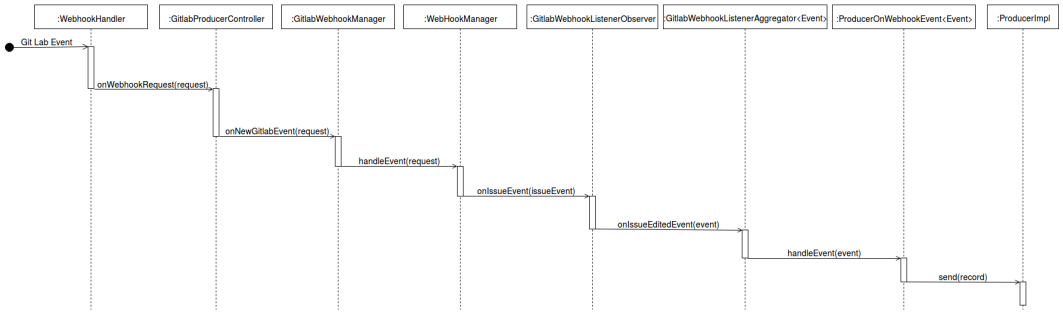


**Figura 5:** Diagramma sequenza Producer

### 2.6.2 Consumer

#### 2.6.2.1 Descrizione

I *Consumer* sono i componenti del sistema che leggono i contenuti dal Message Broker. Il loro compito é quello di consumare i messaggi presenti in coda, il cui

topic sia correlato al servizio di contatto associato a tali Consumer. All'interno di *Butterfly* esistono tre tipi di *Consumer*, uno per ogni servizio di contatto:

1. Slack Consumer;
2. Email Consumer;
3. Telegram Consumer.

#### 2.6.2.2 Diagrammi dei package

Seguono i diagrammi dei package relativi ai Consumer.

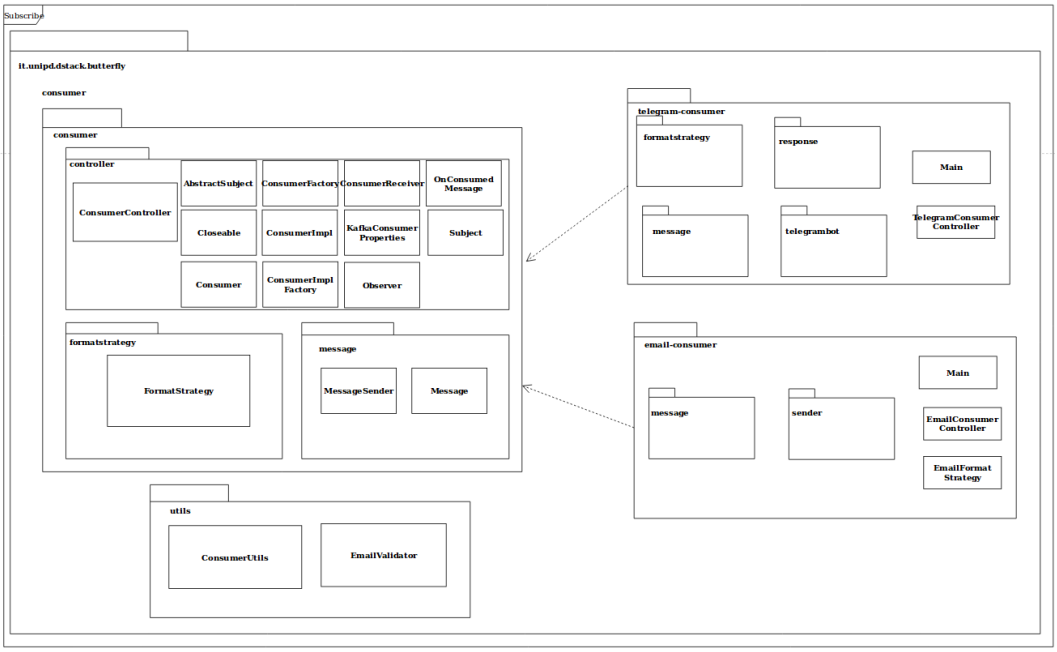


**Figura 6:** Diagramma package Consumer

#### 2.6.2.3 Diagrammi di sequenza

Seguono i diagrammi di sequenza relativi ai Consumer.

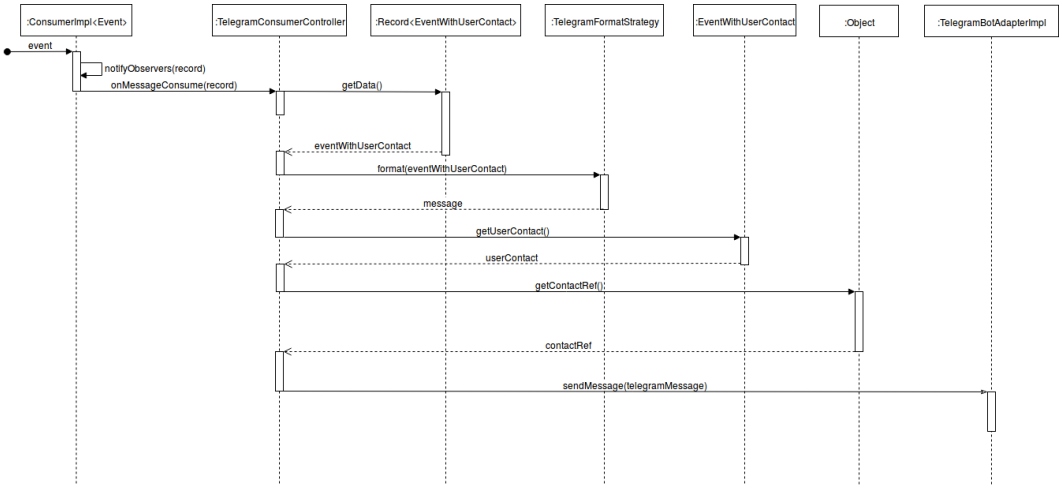


**Figura 7:** Diagramma di sequenza Consumer

#### 2.6.2.4 Design Patterns notevoli utilizzati

- **Strategy Pattern**: i *Consumer* devono estrapolare il messaggio ed inviarlo al servizio di contatto più opportuno. Tuttavia servizi di contatto diversi si aspettano formattazioni del testo diverse. Durante la progettazione del sistema abbiamo quindi incapsulato l'algoritmo di formattazione per rendere il design flessibile ed estendibile. Abbiamo dunque applicato lo *Strategy Pattern*.
  In particolare, all'interno del package *Consumer* é presente l'interfaccia *FormatStrategy*, che definisce la segnatura per un generico algoritmo di formattazione. Ogni Consumer specifico, come *Telegram consumer*, utilizza poi una classe concreta per formattare e presentare il testo in base al servizio di contatto di destinazione. Al momento della formattazione quindi non vi é mai un riferimento esplicito alla logica utilizzata: questa risulta infatti totalmente disaccoppiata dal suo impiego vero e proprio.
  La scelta dello *Strategy Pattern* porta i seguenti benefici:

  – Maggiore flessibilità e riusabilità;
  – L'algoritmo è reso indipendente dall'utilizzatore, è passato a run-time tramite Dependency Injection e può essere cambiato a run-time;
  – Diventa possibile aggiungere nuovi tipi di formattazione del testo semplicemente creando una nuova implementazione dell'interfaccia *FormatStrategy*.

  La figura n.8 rappresenta il diagramma delle classi che descrive l'implementazione del pattern appena descritto.

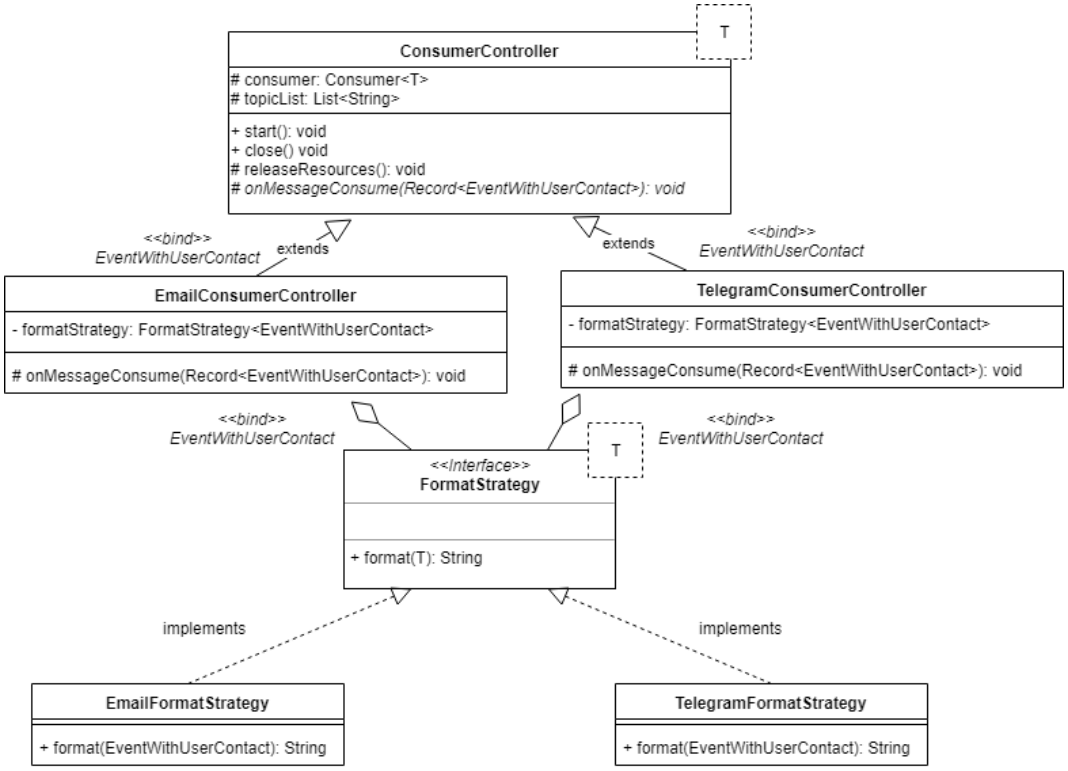


**Figura 8:** Strategy Pattern

- **Observer Pattern**: ogni *Consumer* richiede periodicamente un nuovo $batch_{\mathbf{G}}$ di messaggi da consumare. Per disaccoppiare la logica di consumo da quella di elaborazione dei singoli messaggi ricevuti, abbiamo impiegato il Pattern Observer per notificare la ricezione di un singolo messaggio (come illustrato in figura n.9).
  L'utilizzo di tale pattern:
  - Favorisce il *loose coupling* tra classi;
  - Offre un modo elegante per separare la logica di consumo del batch di messaggi da quella di reazione ad un nuovo messaggio;
  - Permette di definire in maniera semplice e chiara più logiche di reazione allo stesso evento.

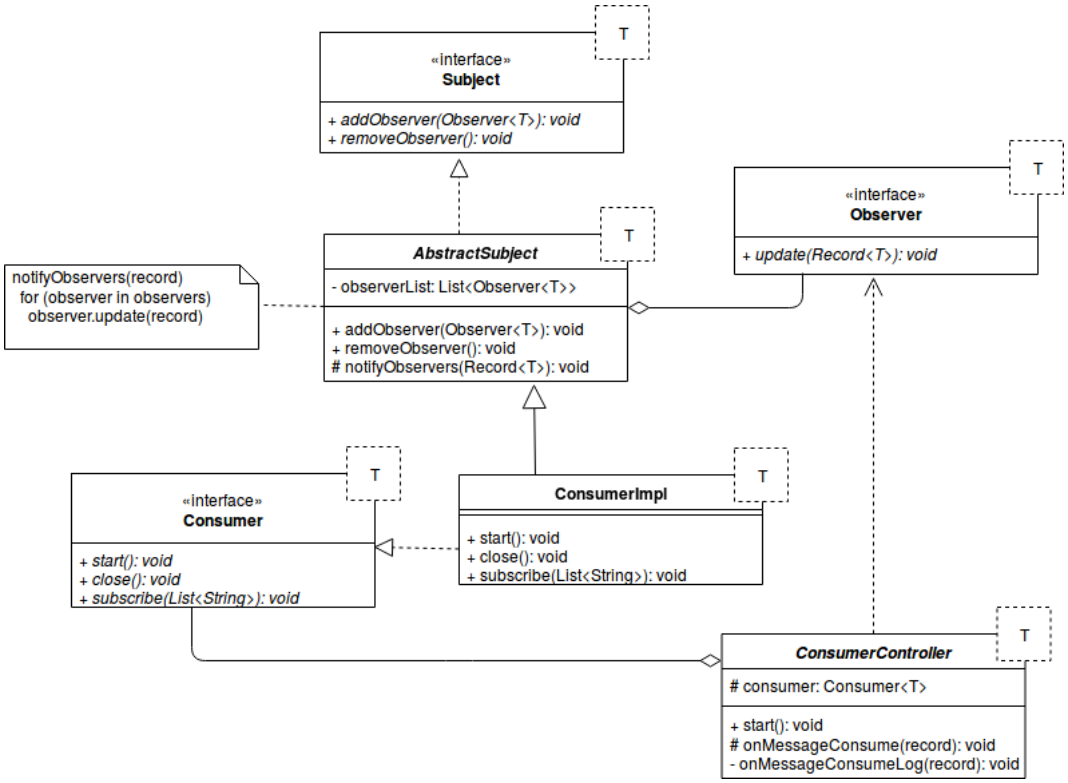


**Figura 9:** Observer Pattern

### 2.6.2.5 Dettaglio Telegram Consumer

#### 2.6.2.5.1 Descrizione

Implementazione specifica di *Consumer* per il servizio di contatto Telegram. Per poter determinare l'identitá dell'utente, é necessario che egli invii la propria email: per farlo deve utilizzare il $Bot_{\mathbf{G}}$ Telegram.
Il suddetto *Bot* supporta due comandi:

- *start*: per l'avvio del Bot;
- *email*: per sottomettere la mail allo User Manager.

Queste richieste vengono modellate mediante l'utilizzo del *Command Pattern*.

#### 2.6.2.5.2 Design Patterns utilizzati

- **Command Pattern**: il *Bot* supporta i comandi *start*, per l'avvio, ed *email*, per l'invio della propria mail. In fase di progettazione il gruppo *DStack* ha deciso di utilizzare il pattern *Command* per modellare tali richieste poichè:
  - Permette di disaccoppiare la classe che invoca una richiesta dalla richiesta stessa;
  - Rende possibile cambiare la richiesta a run-time;
  - Per aggiungere nuovi comandi in futuro sarà sufficiente estendere l'interfaccia apposita, senza modificare il codice del *Consumer*.

La figura 10 evidenzia il pattern sopra citato.

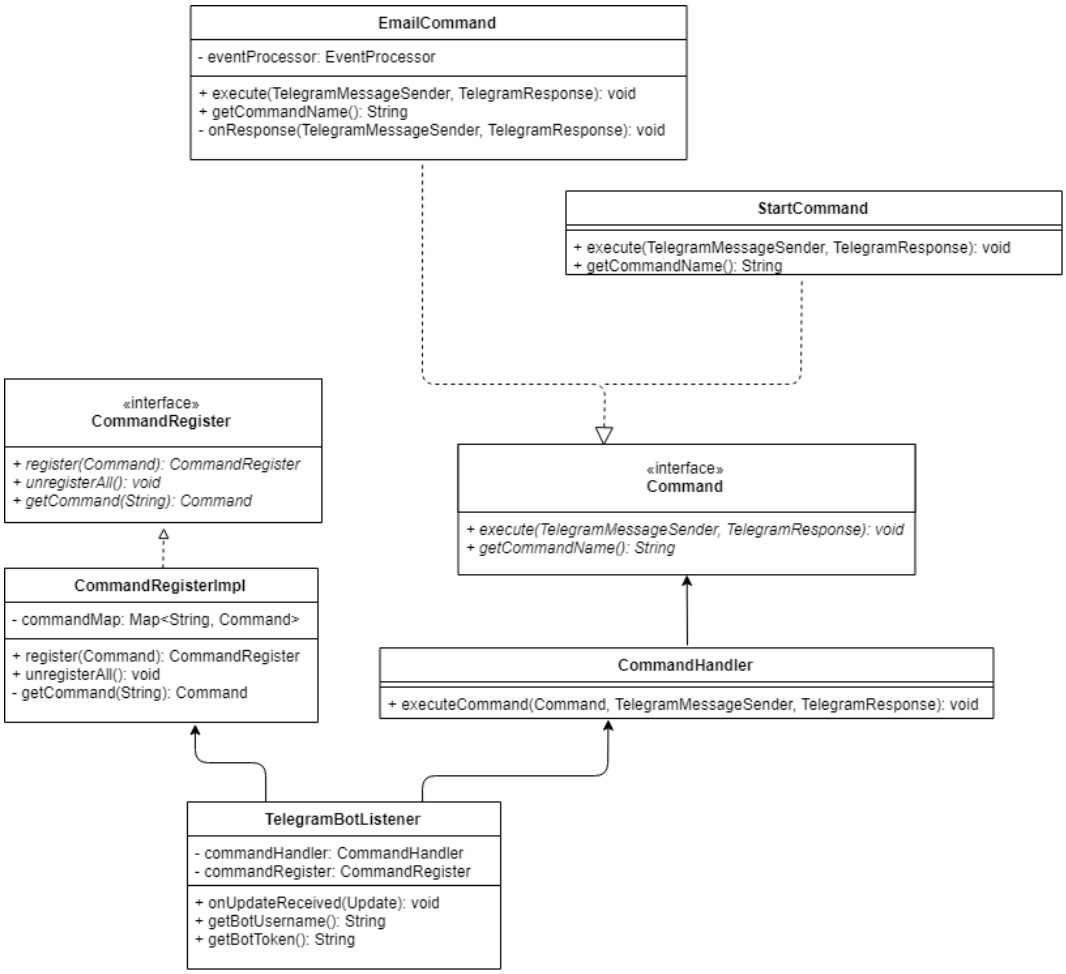


**Figura 10:** Command Pattern

## 2.7 Template Method Pattern

Il Configuration Manager é una componente che definisce una interfaccia per leggere informazioni di configurazione, definire valori di default e convertirli esplicitamente a stringhe, interi e valori booleani. Il gruppo ha deciso di usare il *Template Method Pattern* durante la fase di progettazione per modellare la struttura del Configuration Manager.

- **Template Method Pattern**: la classe *AbstractConfigManager* definisce il metodo *getConfigValue*, che fa uso di *readConfigValue*: quest'ultimo metodo é astratto e il suo comportamento può essere cambiato dalle varie sottoclassi. Questa struttura é conforme al *Template Method Pattern*, scelto dal gruppo per i seguenti vantaggi:
  - Favorisce l'*Inversion of Control*, e permette al codice di adattarsi in maniera migliore ai principi *SOLID*;
  - Riduce la duplicazione del codice;
  - Incentiva il riuso;
  - Aumenta la flssibilitá generale, poiché permette alle sottoclassi di decidere come implementare passi dell'algoritmo generale.

L'implementazione di questo paradigma é spiegata in dettaglio nel relativo diagramma delle classi (figura n.11).

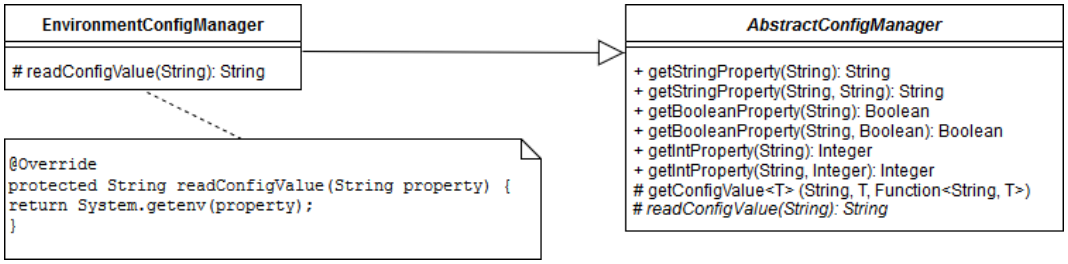


**Figura 11:** Template Method Pattern