ONION

SOFTWARE

softwareonion@gmail.com

# Allegato tecnico

### Informazioni sul documento

| | |
|---|---|
| **Versione** | 1.0.0 |
| **Data approvazione** | 2019-06-04 |
| **Responsabili** | Linpeng Zhang |
| **Redattori** | Federico Brian |
| | Matteo Lotto |
| | Nicola Pastore |
| | Nicola Zorzo |
| **Verificatori** | Federico Omodei |
| | Alessio Lazzaron |
| **Stato** | Approvato |
| **Lista distribuzione** | *prof. Riccardo Cardin* |
| | *Onion Software* |

## Scopo del documento

*Il presente documento contiene le scelte architetturali di Onion Software, adeguatamente motivate, 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

Il documento ha lo scopo di descrivere in maniera dettagliata, coesa e coerente le peculiarità del prodotto software *Butterfly*, sviluppato dal team *Onion Software*. Si descriveranno i *design pattern*$_G$ utilizzati e si illustreranno i *diagrammi di attività*$_G$, *sequenza*$_G$ e *package*$_G$. Infine, verrà effettuato un confronto tra lo stato d'avanzamento dello sviluppo del prodotto operato in sede di *Technology Baseline*$_G$ e l'attuale *Product Baseline*$_G$, ponendo particolare attenzione su casi d'uso e requisiti soddisfatti.

## 1.2 Scopo del prodotto

Una realtà *enterprise* che opera nel campo dell'*information technology* implementa processi di *Continuous Integration* e *Continuous Delivery*[1] per farlo con efficacia, utilizza degli strumenti che aiutino l'automatizzazione di certe operazioni. La maggior parte di questi strumenti fornisce "out-of-the-box" dei meccanismi di segnalazione che permettono la notifica di eventuali problematiche riscontrate nelle varie fasi. Ognuno di questi strumenti ha, però, un proprio meccanismo specifico di esposizione dei messaggi/segnalazioni, spesso con limitate capacità di configurazione.
I limiti sopra indicati sfociano spesso nella necessità per l'utente di interfacciarsi con molteplici dashboard specifiche, ognuna delle quali con propria struttura. Spesso, poi, queste interfacce sono anche di difficile accessibilità, sia a causa della complessità applicativa, sia a causa di limitazioni di visibilità in rete: è una considerazione troppo ottimistica pensare che tutti siano in grado di utilizzare e gestire con efficacia tutti i meccanismi di tutti gli strumenti utilizzati per costruire un'architettura software. Cosa succede se la persona che si occupa di una determinata attività è in ferie/sta male ed è quindi impossibilitata a rispondere a quell'attività? Il risultato è che l'attività rimane in standby o si perde tra le notifiche.
È qui che nasce l'esigenza di un **monitor per i processi CI/CD**, che quindi raggruppi tutte le notifiche e ne permetta una gestione personalizzata. Questa è la mansione principale che *Butterfly* porta a compimento.

## 1.3 Glossario

Al fine di evitare possibili ambiguità relative al linguaggio utilizzato nei documenti formali, viene fornito il Glossario v3.0.0. In questo documento vengono definiti e descritti tutti i termini con un significato specifico. Per facilitare la comprensione, i termini saranno contrassegnati da una 'G' a pedice.

## 1.4 Riferimenti informativi

- ***Analisi dei requisiti v3.0.0***;
- **Slide L03 del corso di Ingegneria del Software - 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: un monitor per i processi CI/CD**
  https://www.math.unipd.it/~tullio/IS-1/2018/Progetto/C1.pdf.

[1] CI/CD

# 2 Architettura del prodotto

## 2.1 Descrizione generale

L'architettura software utilizzata è *event-driven*$_{G}$ *con topologia broker*$_{G}$.
Ogni componente che fa parte del sistema conosce solamente l'esistenza del *broker* e comunica con esso attraverso l'invio o la ricezione di messaggi che rispecchiano l'avvenimento di un determinato evento. Proprio per questo motivo ogni componente è trattato come un'unità isolata e asincrona.

Questo tipo di architettura è stata scelta in base alle seguenti considerazioni:

- **dominio del problema**: il progetto *Butterfly* prevede l'implementazione di componenti specifici (*producer e consumer*) per servizi e canali di comunicazione completamente scollegati tra loro, che non hanno bisogno di conoscere la presenza l'uno dell'altro;
- **scalabilità**: l'indipendenza dei servizi/canali di comunicazione e il comportamento asincrono dei componenti permette una buona scalabilità del sistema.

Le principali tipologie di componenti del sistema sono:

- ***producers***: rappresentano i componenti che rimangono in ascolto del *webhook*$_{G}$ proveniente dal servizio loro assegnato (*Redmine, GitLab o SonarQube*), producono il messaggio relativo all'evento descritto da quest'ultimo e lo inviano al *broker*;
- ***consumers***: rappresentano i componenti che ricevono il messaggio relativo a un determinato evento dal *broker* e lo inoltrano ai destinatari finali attraverso il canale di comunicazione designato (*email, Telegram o Slack*);
- ***gestore personale***: rappresenta i componenti che si occupano della manipolazione dei messaggi inviati dai *producer* al *broker* e da quest'ultimo ai *consumer*. Nello specifico, hanno il compito di filtrare i messaggi in entrata per stabilire il *consumer* appropriato a cui viene inviato il messaggio e la lista dei destinatari finali.

## 2.2 Diagramma dei package generale

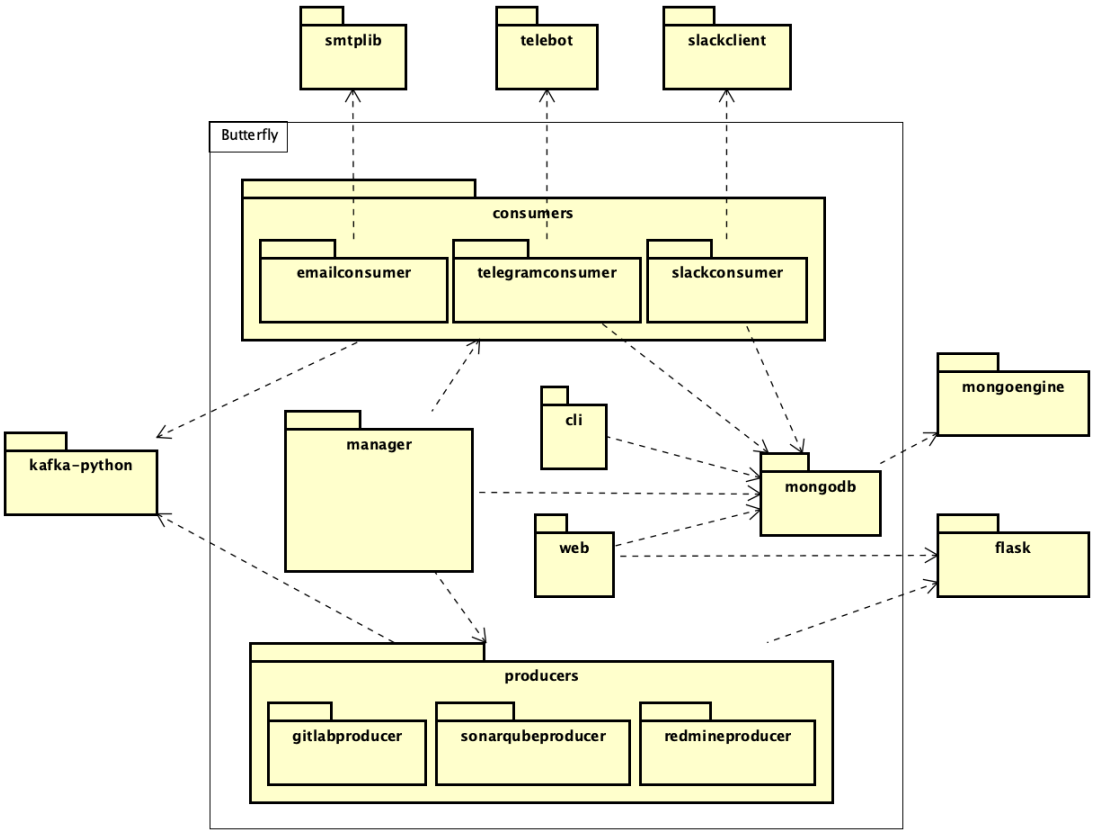


Figura 1: Diagramma dei package dell'architettura generale

Il diagramma di sequenza del funzionamento generale di *Butterfly* è volutamente omesso poiché ritenuto poco efficace: le varie componenti del sistema verranno illustrate separatamente e nel dettaglio nelle prossime sezioni.

### 2.2.1 Architettura delle classi interfaccia

*Onion Software* ha sviluppato una piccola applicazione web, che ha lo scopo di essere user-friendly per l'utilizzatore finale. Il team ha anche sviluppato una $CLI_G$, per poter comunicare direttamente con il gestore personale tramite riga di comando.

#### 2.2.1.1 Diagramma delle classi

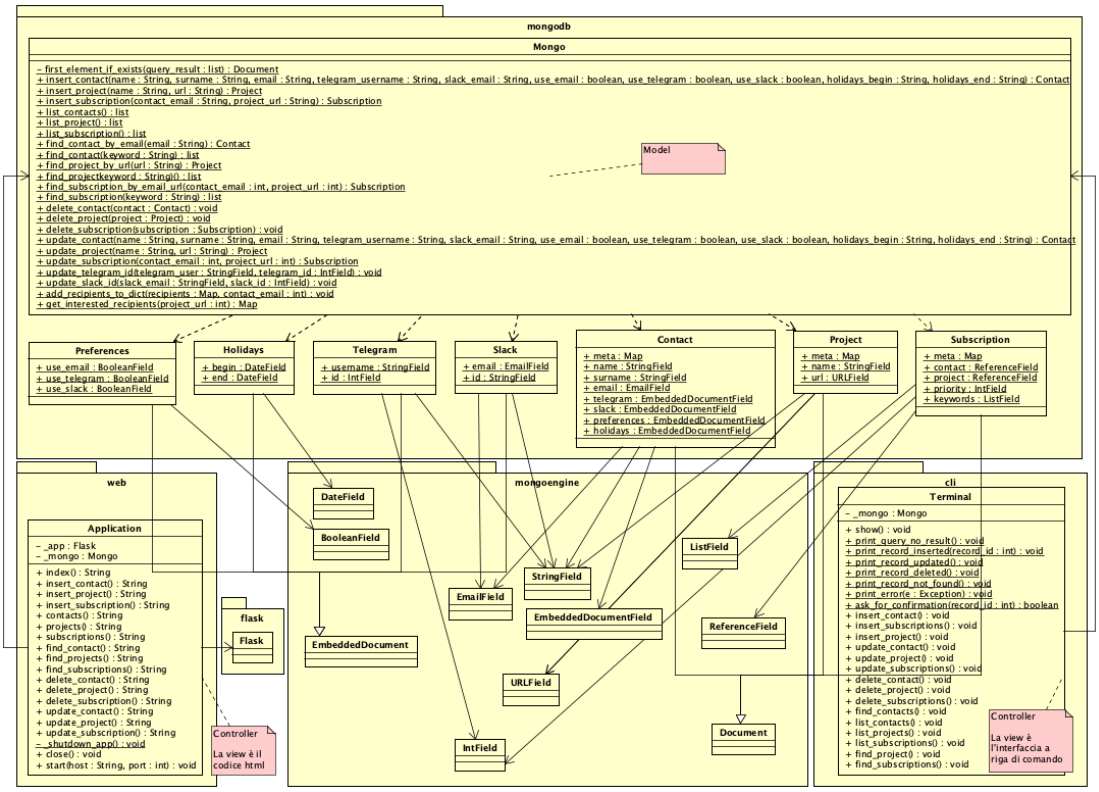


Figura 2: Diagramma delle classi del componenti interfaccia

#### 2.2.1.2 Design pattern utilizzati

##### 2.2.1.2.1 Model View Controller

La scelta di questo pattern è cruciale in quanto permette a *Butterfly* di essere totalmente indipendente dal modo in cui si vuole visualizzare qualsiasi informazione e manipolare dati, ad opera dell'utente finale.
Come suggerisce il nome, le componenti di questo pattern sono tre:

- **model**, che incorpora i dati, nel nostro caso il database *MongoDB*.
- **view**, che permette all'utente di interagire con i dati contenuti nel model;
- **controller**, che esegue le operazioni decise dall'utente finale comunicando con la view manipolando i dati contenuti nel model.

Nel nostro caso, è stato scelto di implementare un controller per ogni specifica view sviluppata. Questo permette maggior separazione di codice, quindi maggiore scalabilità e maggiore possibilità di integrare nuove view.

## 2.3 Architettura di `manager`

Il *manager*, o gestore personale, è il componente protagonista dell'architettura di *Butterfly*: esso infatti si occupa di catturare i messaggi inviati dai *producer*, interagire con il database *mongodb* per cercare le informazioni necessarie all'invio dei messaggi; successivamente invia tali messaggi ai *consumer*. La comunicazione avviene interamente utilizzando il formato $JSON_G$ ed utilizzando l'architettura $API\ REST_G$.

### 2.3.1 Diagramma delle classi

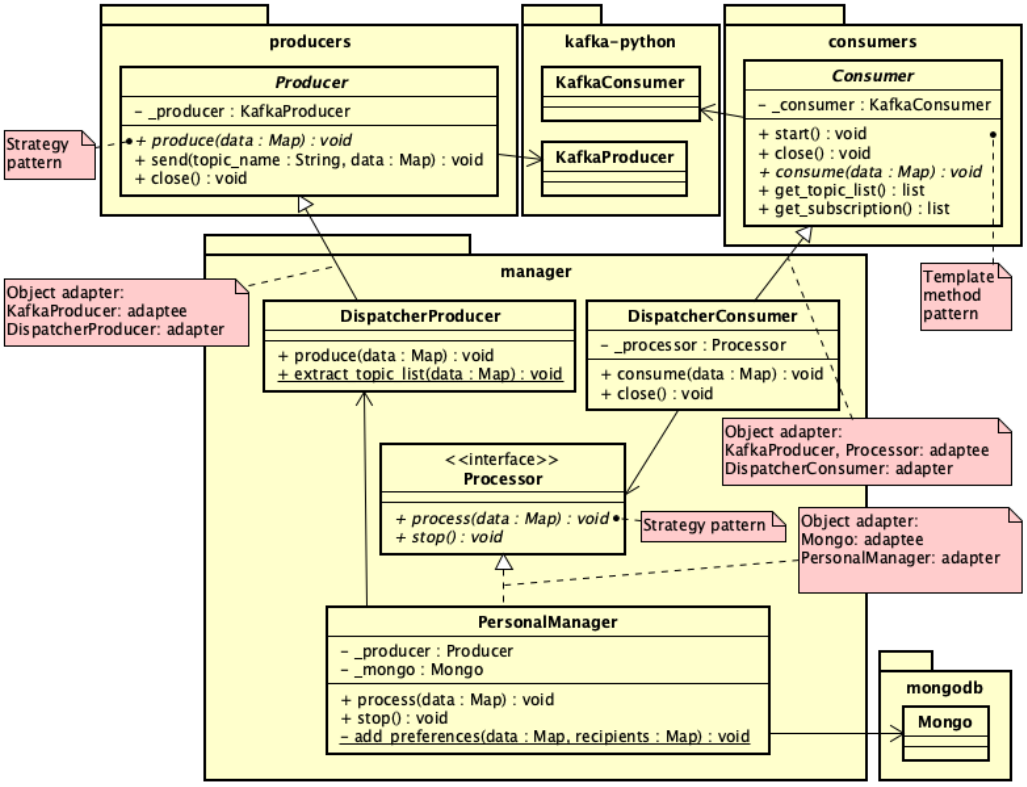


Figura 3: Diagramma delle classi del componente *manager*

### 2.3.2 Design pattern utilizzati

#### 2.3.2.1 Strategy

- `Producer`:
  Lo Strategy Pattern utilizzato dalla classe astratta `Producer` è illustrato nel §2.4.1.2.2;
- `Processor`:
  La classe astratta `Processor` può definire diversi tipi di gestore personale: questa scelta è stata attuata per dare ulteriore modularità al codice. In particolare, il metodo astratto *`process(data: map)`* non fornisce alcun tipo di informazione circa il modo di processare il messaggio proveniente da `DispatcherConsumer`, come ad esempio la scelta del consumer da utilizzare. È stato scelto di implementare quindi lo Strategy Pattern per far sì che `DispatcherConsumer` potesse decidere a run-time come processare l'informazione adeguatamente in un'unica classe.

  Questa scelta offre i seguenti benefici:

  - **organizza tipologie di algoritmi simili in un'unica classe**, agevolando il riuso del codice in contesti analoghi;
  - **una valida alternativa al *subclassing*** $_G$, che altrimenti legherebbe significativamente il contesto d'utilizzo con i diversi algoritmi, rendendo il codice poco riusabile, difficile da comprendere e da mantenere, senza contare che non sarebbe possibile scegliere dinamicamente[2] l'algoritmo giusto da usare;

[2] a run-time

– **esclude gli statement condizionali**: sarà infatti il *dynamic-binding*$_G$ dell'oggetto d'invocazione a stabilire il giusto algoritmo da utilizzare, evitando codice che appesantirebbe ulteriormente la classe.

#### 2.3.2.2 Template Method

Il Template Method Pattern, utilizzato dalla classe astratta `Consumer`, è illustrato nel §2.4.2.2.1.

#### 2.3.2.3 Object Adapter

- **fra** `DispatcherProducer` **e** `KafkaProducer`:
  `dispatcherProducer` riceve da `PersonalManager`[3], un messaggio contenente varie informazioni, come ad esempio il titolo ed il contenuto della segnalazione. È necessario rendere tale messaggio comprensibile per il broker Kafka: si rende quindi indispensabile l'utilizzo del pattern Object Adapter, in modo da codificare il messaggio ricevuto dai producer per far sì che risulti utilizzabile dal `KafkaProducer`;
- **fra** `KafkaConsumer` **a** `DispatcherConsumer`:
  il messaggio, ricevuto precedentemente da `KafkaProducer`, si trova attualmente all'interno di Kafka. Il broker provvederà ad arricchire il messaggio con altre informazioni di varia natura. È necessario rendere tale messaggio comprensibile per i futuri consumer: si rende quindi indispensabile l'utilizzo del pattern Object Adapter, in modo da far risultare utilizzabile il messaggio da parte dei consumer. Di questo se ne occuperà la classe `DispatcherConsumer`;
- **fra** `PersonalManager` **e** `Mongo`:
  `PersonalManager` dovrà utilizzare le funzionalità di Mongo per poter inviare con efficacia ed efficenza il messaggio al destinatario giusto, secondo i canali designati. Per fare questo ha bisogno di comunicare con il database per arricchire la segnalazione: si rende quindi necessario il template Object Adapter.

### 2.3.3 Diagramma di sequenza

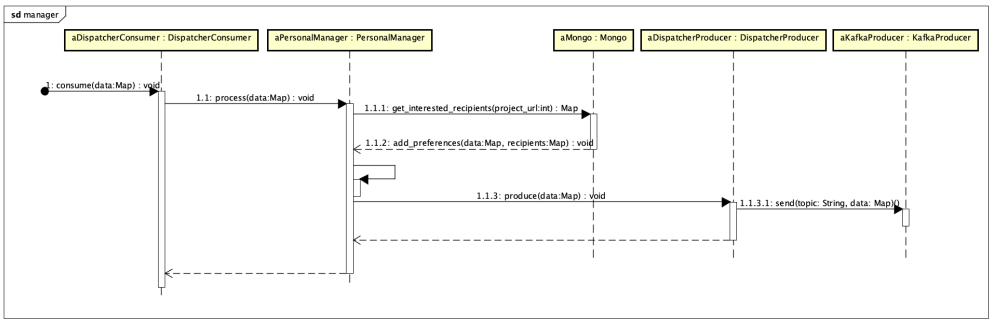


Figura 4: Diagramma di sequenza del componente *manager*

## 2.4 Architettura dei servizi `producer` e `consumer`

### 2.4.1 Producer

I *producer* sono i componenti dell'architettura che hanno lo scopo di generare informazioni: sono infatti i responsabili della creazione dei messaggi ed il loro invio al *dispatcher-producer* contenuto nel *manager*.
*Onion Software* ha sviluppato tre componenti producer:

[3] il gestore personale

- Redmine producer;
- GitLab producer;
- SonarQube producer.

#### 2.4.1.1 Diagramma delle classi

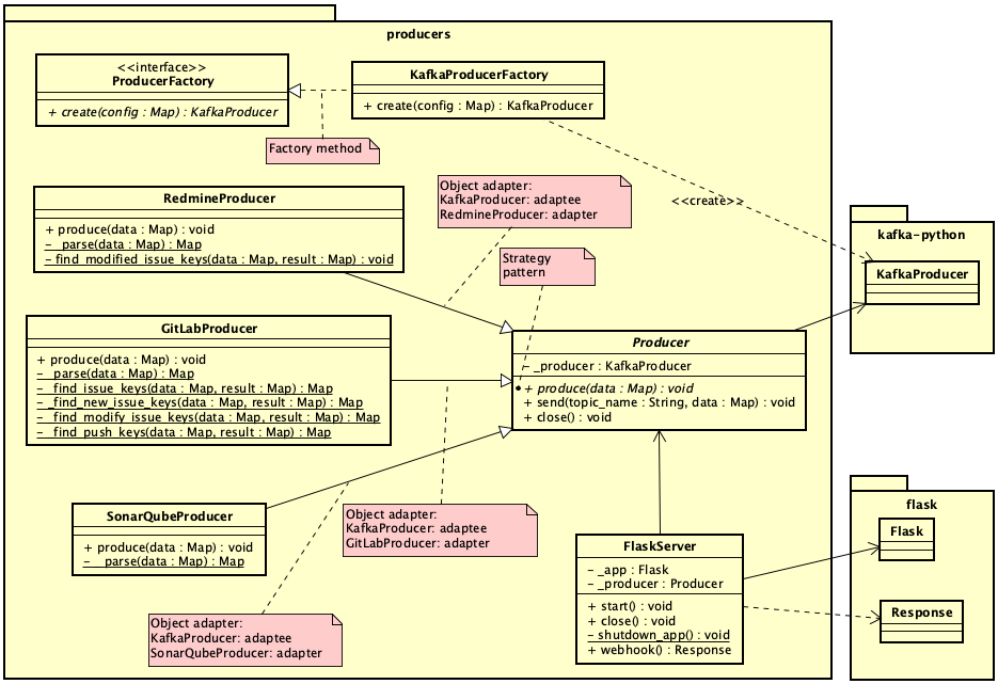


Figura 5: Diagramma delle classi dei componenti *producer*

#### 2.4.1.2 Design pattern utilizzati

##### 2.4.1.2.1 Factory Method

L'interfaccia `ProducerFactory` mette a disposizione un metodo astratto *`create(config: map)`* a disposizione della classe concreta derivata `KafkaProducerFactory`. Questo elimina il bisogno di classi con applicazioni specifiche al codice. Esso, infatti, ha a che fare solamente con l'interfaccia, pertanto può funzionare con ogni classe da essa derivata.

##### 2.4.1.2.2 Strategy

La classe `FlaskServer`, che si interfaccia con l'applicazione web adibita ad interagire con l'utente finale, ha bisogno di un algoritmo di *parsing*$_G$ per il linguaggio JSON diverso per ogni componente producer implementato. È stato quindi scelto di incapsulare i differenti algoritmi di parsing, utilizzati per estrapolare le informazioni d'interesse per *Butterfly*, in un'unica classe: questo per poter cominciare a generare un'informazione più astratta e vicina al linguaggio naturale, senza incorporare per ogni classe producer il suo personale algoritmo.
Questa scelta offre i seguenti benefici:

- **organizza tipologie di algoritmi simili in un'unica classe**, agevolando il riuso del codice in contesti analoghi;
- **una valida alternativa al *subclassing*$_G$**, che altrimenti legherebbe significativamente il contesto d'utilizzo con i diversi algoritmi, rendendo il codice poco riusabile, difficile da

comprendere e da mantenere, senza contare che non sarebbe possibile scegliere dinamicamente[4] l'algoritmo giusto da usare;

- **esclude gli statement condizionali**: sarà infatti il *dynamic-binding*$_G$ dell'oggetto d'invocazione a stabilire il giusto algoritmo da utilizzare, evitando codice che appesantirebbe ulteriormente la classe.

#### 2.4.1.2.3 Object adapter

La classe `KafkaProducer`, propria del broker, accetta messaggi codificati in una certo modo. È compito della classe `RedmineProducer` assicurarsi che venga generato un messaggio comprensibile per il broker. Si rende necessario, quindi, l'utilizzo di un Object Adapter fra le due classi sopracitate.

**N.B.:** tale scelta architetturale è stata effettuata anche per `GitLabProducer` e `SonarQubeProducer`, per motivi del tutto analoghi.

### 2.4.1.3 Diagrammi di sequenza

#### 2.4.1.3.1 Redmine

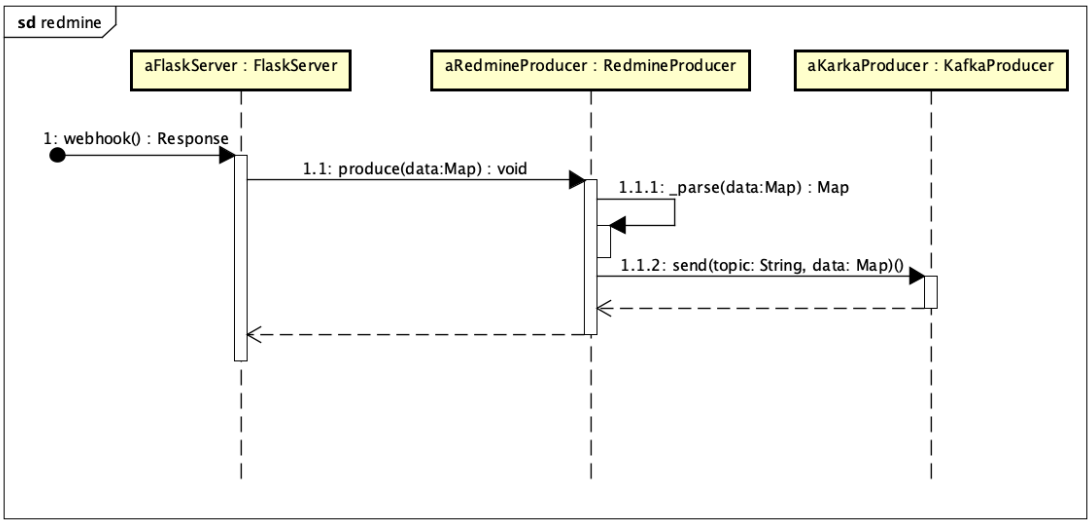


Figura 6: Diagramma di sequenza del componente *producer* Redmine

[4] a run-time

#### 2.4.1.3.2 GitLab

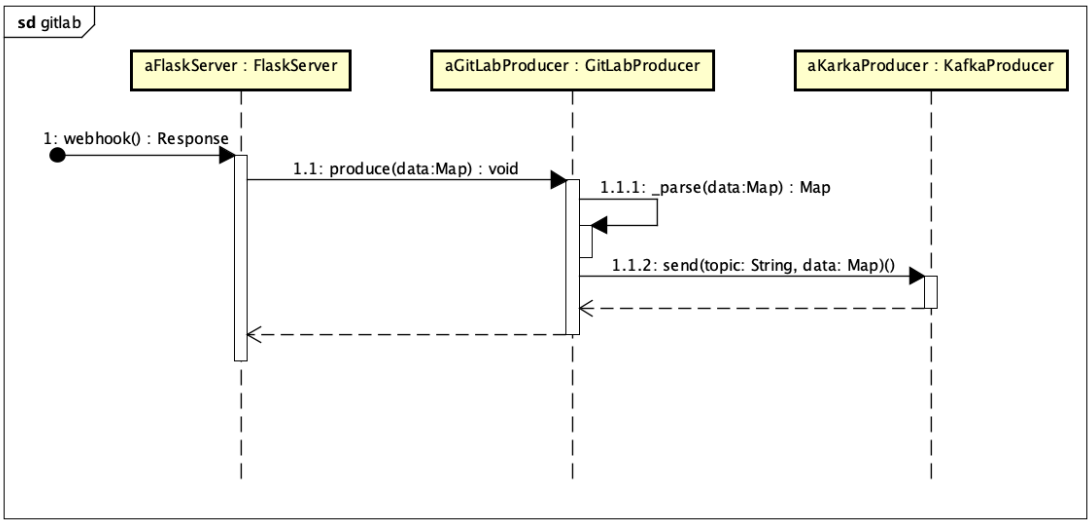


Figura 7: Diagramma di sequenza del componente *producer* GitLab

#### 2.4.1.3.3 SonarQube

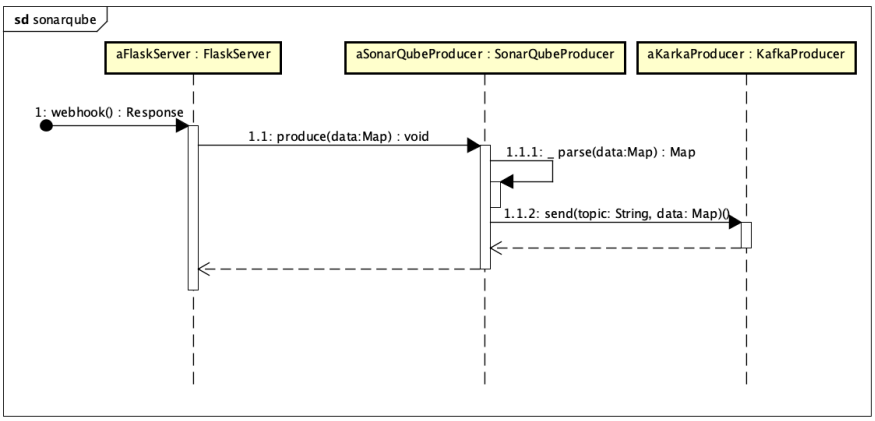


Figura 8: Diagramma di sequenza del componente *producer* SonarQube

### 2.4.2 Consumer

I *consumer* sono i componenti dell'architettura che hanno lo scopo di ricevere l'informazione, sotto forma di messaggio, e provvedere ad inviarla per mezzo di un certo canale.
Tali canali, individuati da *Onion Software*, sono:

- e-mail consumer;
- Telegram consumer;
- Slack consumer.

#### 2.4.2.1 Diagramma delle classi

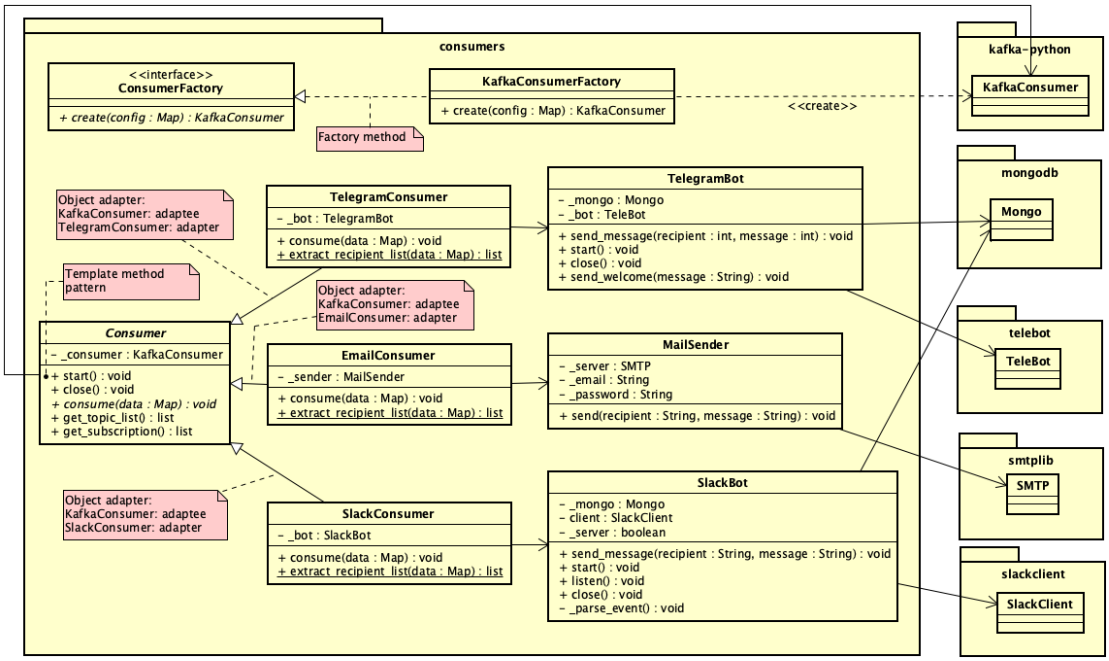


Figura 9: Diagramma delle classi dei componenti *consumer*

#### 2.4.2.2 Design pattern utilizzati

##### 2.4.2.2.1 Template Method

Il metodo `start()`, che implementa il pattern *Template Method*, ha comportamento identico per tutte e tre le classi consumer. È stato quindi scelto di incapsulare tale metodo in una classe base astratta, `Consumer` (supertipo astratto di `TelegramConsumer`, `EmailConsumer` e `SlackConsumer`) per lasciare che sia il dynamic-binding a decidere su quale consumer far avvenire la chiamata al metodo `start()`. Tale scelta comporta i seguenti benefici:

- un maggior **riuso del codice**;
- **raggruppano i comportamenti in comune** tra diverse classi, rendendosi indispensabili in contesti come, ad esempio, in classi di librerie.

##### 2.4.2.2.2 Object adapter

La classe `TelegramConsumer` accetta messaggi codificati in una certo modo. È suo compito assicurarsi che venga generato un messaggio comprensibile proveniente dalla classe `KafkaConsumer`, propria del broker, per poter inviare efficacemente una segnalazione. Si rende necessario, quindi, l'utilizzo di un Object Adapter fra le due classi sopracitate.

**N.B.:** tale scelta architetturale è stata effettuata anche per `EmailConsumer` e `SlackConsumer`, per motivi del tutto analoghi.

##### 2.4.2.2.3 Factory Method

L'interfaccia `ConsumerFactory` mette a disposizione un metodo astratto *`create(config: map)`* a disposizione della classe concreta derivata `KafkaConsumerFactory`. Questo elimina il bisogno di classi con applicazioni specifiche al codice. Esso, infatti, ha a che fare solamente con l'interfaccia, pertanto può funzionare con ogni classe da essa derivata.

#### 2.4.2.3 Diagramma di sequenza

##### 2.4.2.3.1 E-mail

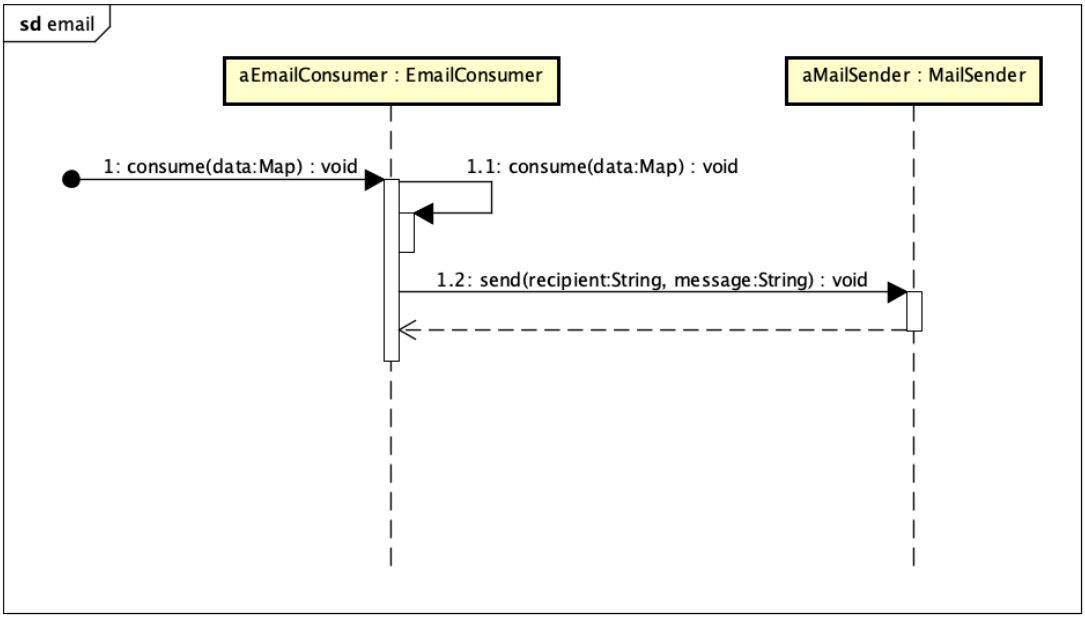


Figura 10: Diagramma di sequenza del componente *consumer* e-mail

##### 2.4.2.3.2 Telegram

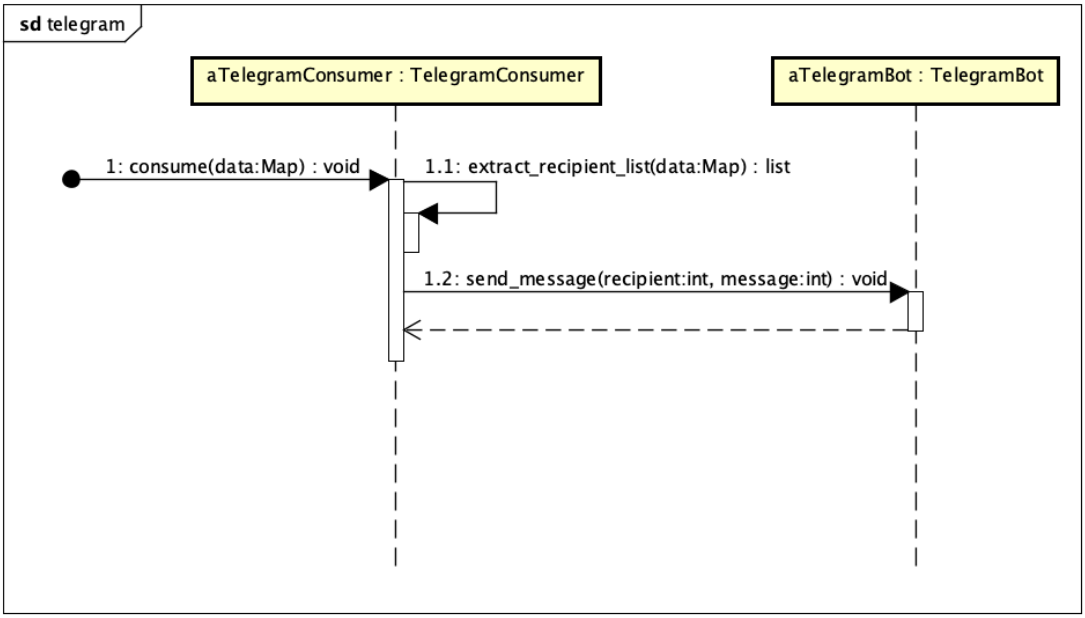


Figura 11: Diagramma di sequenza del componente *consumer* Telegram

#### 2.4.2.3.3 Slack

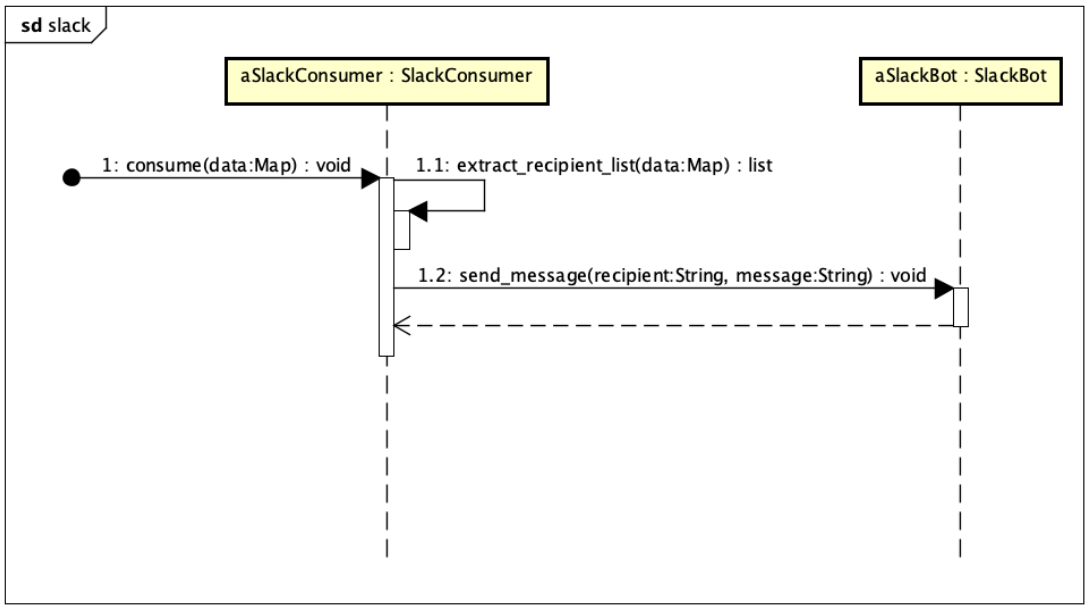


Figura 12: Diagramma di sequenza del componente *consumer* Slack