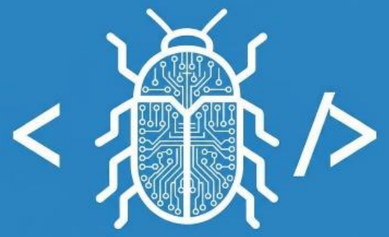
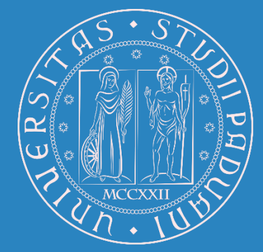


ARCHITETTURA

BugBusters gruppo 4 – A.A 2025-2026

Alberto Autiero, Marco Favero, Alberto Pignat, Marco Piro, Linor Sadè,
Leonardo Salviato, Luca Slongo

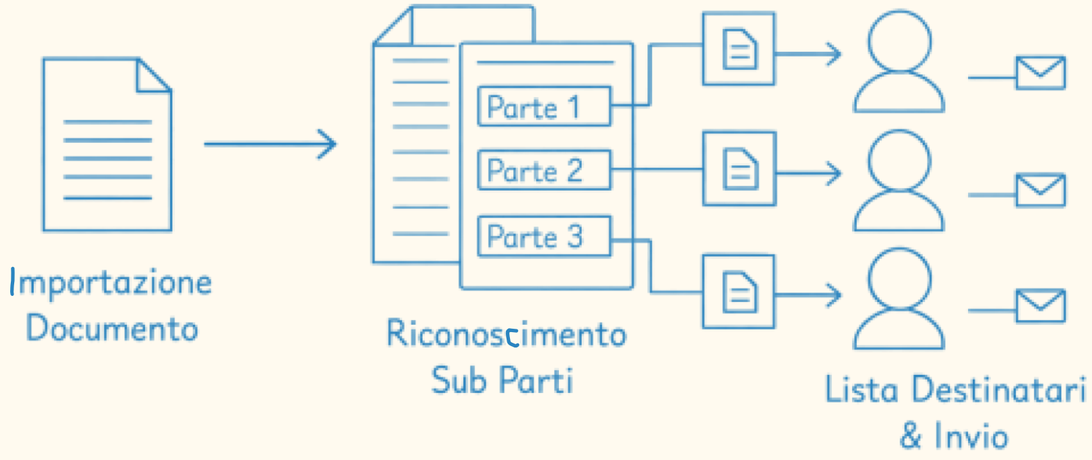
Email: bugbusters.unipd@gmail.com



Il capitolato

2 moduli:

1. Ai Assistant Generativo



2. Ai Co-pilot per CdL

SCELTE ARCHITETTURALI: DEPLOYMENT

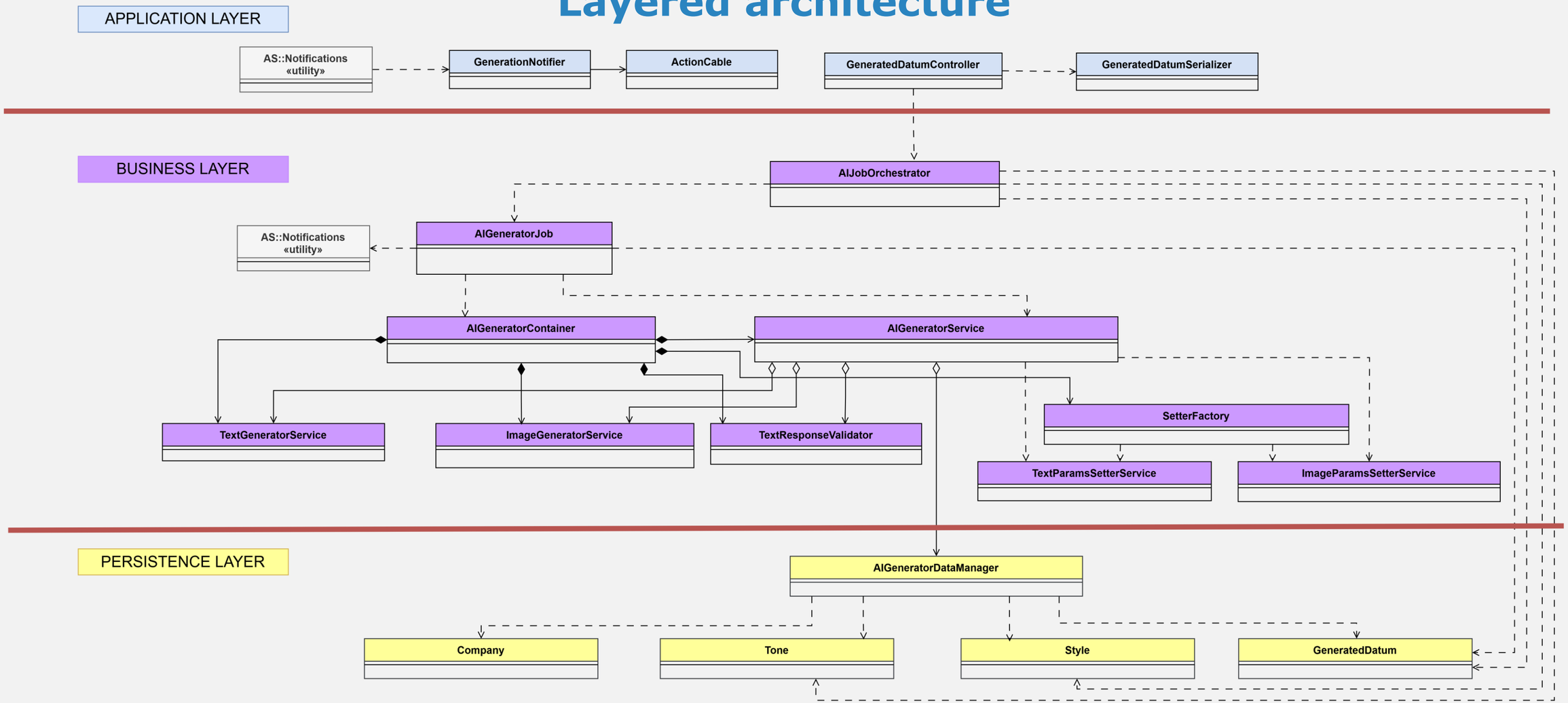
Backend: Monolite

- **Framework Rails:** Le convenzioni del framework Rails prediligono un'architettura di deploy monolitica. Ad esempio l'utilizzo di SolidQueue e SolidCache (gem di rails) richiede un database unico.
- **Natura del progetto**: Nel capitolato e durante le riunioni con l'azienda ci è stato spiegato come l'applicazione deve essere adatta per essere utilizzata per un'integrazione e come il focus sia sulla sperimentazione delle funzionalità.

Frontend: SPA

SCELTE ARCHITETTURALI: LOGICA

Layered architecture



DESIGN PATTERN

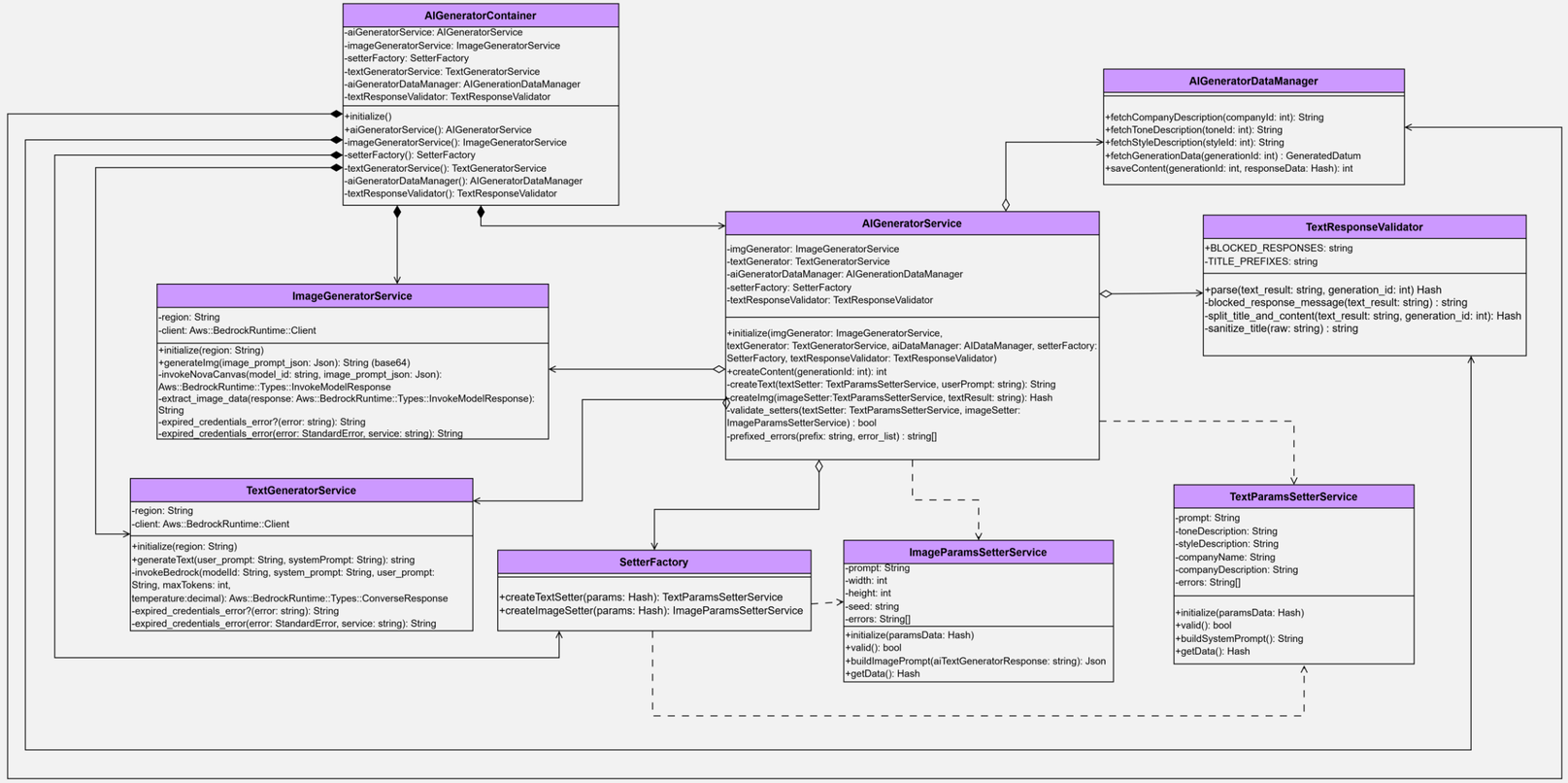
Dependency Injection

Problema:

Le classi di business logic sono troppo accoppiate, il codice sarebbe difficile da testare e ci sarebbero un sacco di dipendenze nascoste.

Soluzione:

L'AIGeneratorContainer centralizza la creazione e configurazione di tutti i servizi. Invece di istanziarli, le classi "ricevono" le dipendenze già pronte (Dependency Injection), garantendo flessibilità e facilità di testing tramite Mock.



DESIGN PATTERN

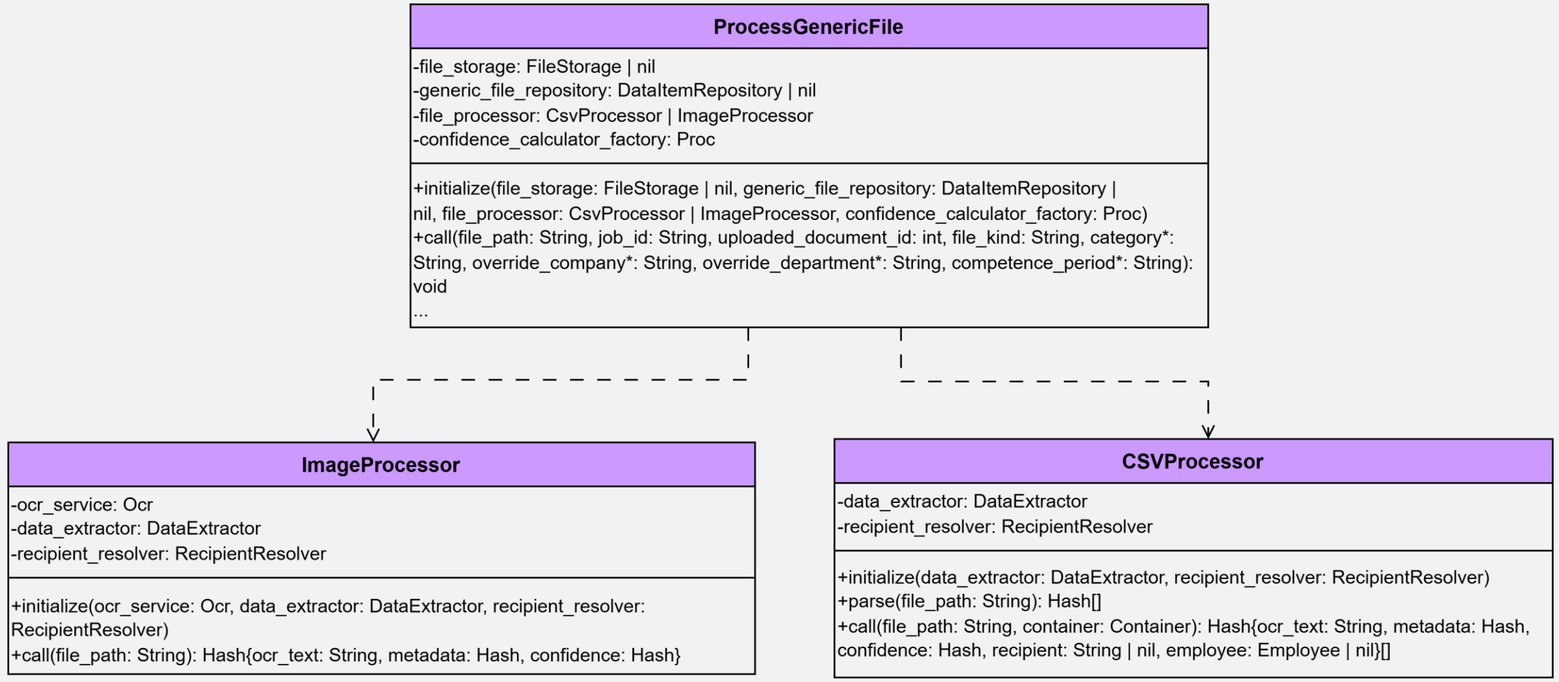
Strategy

Problema:

La classe ProcessGenericFile deve poter processare diversi tipi di file, immagini e csv.
In futuro possibile aggiunta di altri tipi senza modificare la logica di coordinamento esistente (Open/Closed Principle)

Soluzione:

ImageProcessor e CSVProcessor hanno entrambi il metodo pubblico +call che viene chiamato da ProcessGenericFile in modo polimorfico garantendo disaccoppiamento e al ta estensibilità.



DESIGN PATTERN

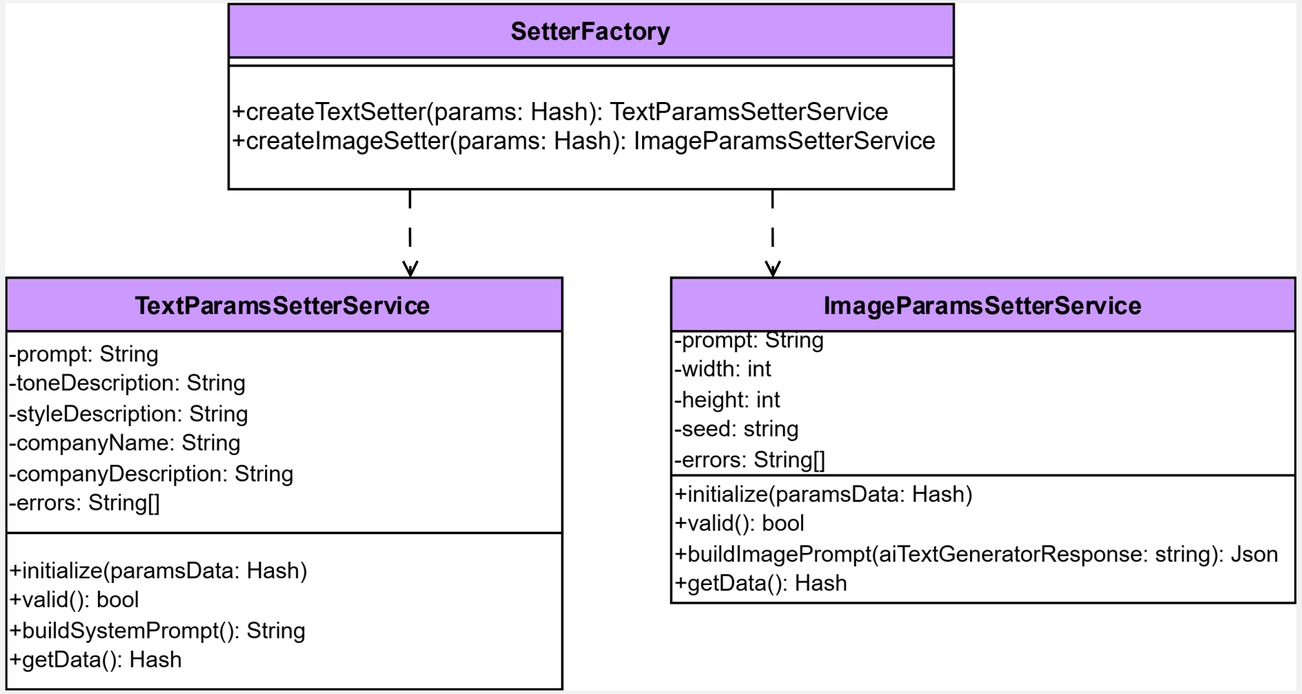
Abstract Factory Method

Problema:

La classe chiamante (AIGeneratorService) deve generare dei Prompt completi in base al tipo di servizio per le chiamate AI a cui ci si appoggia, sia per testo che per immagine.

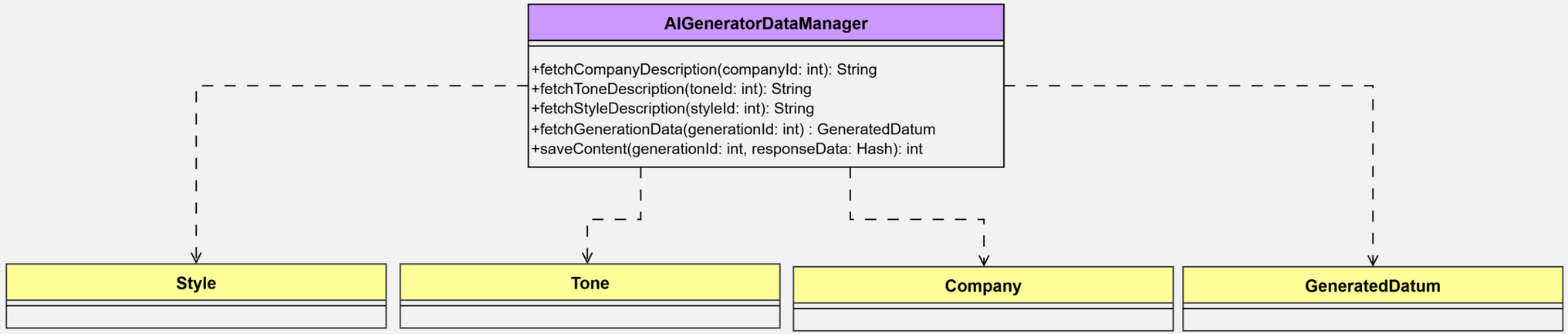
Soluzione:

La abstract factory fornisce una famiglia di oggetti correlati tra loro, in questo caso dei prompt builder per testo e immagini.
Il vero beneficio lo si vedrà quando si cambierà provider o i modelli richiederanno una struttura diversa in input.



DESIGN PATTERN

Repository



Problema:

Per query semplici il ORM di rails è sufficiente, ma per operazioni ricorrenti, complesse, concorrenza, gestione di stato, è scomodo e ripetitivo usarlo.

Soluzione:

Con il pattern repository si centralizza l'uso dei *models* messi a disposizione da Rails, in maniera da disaccopiare il codice, ridurre errori e non ripetere istruzioni ricorrenti.

DESIGN PATTERN

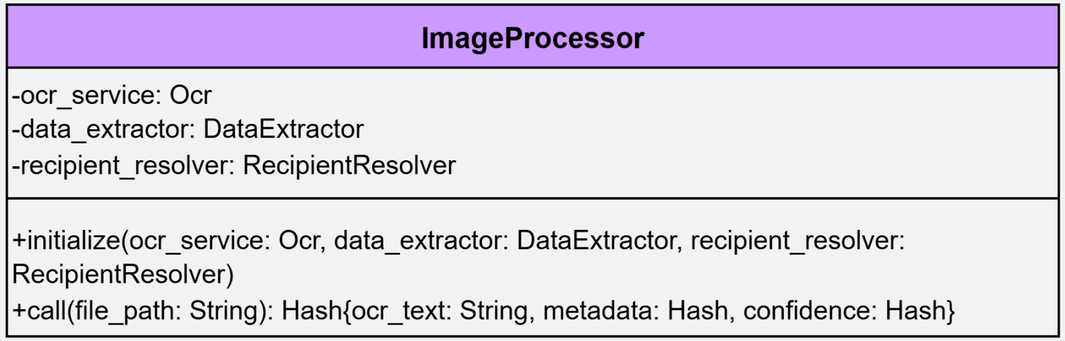
Service

Problema:

Troppe responsabilità alle singole classi, ad esempio ai controller.

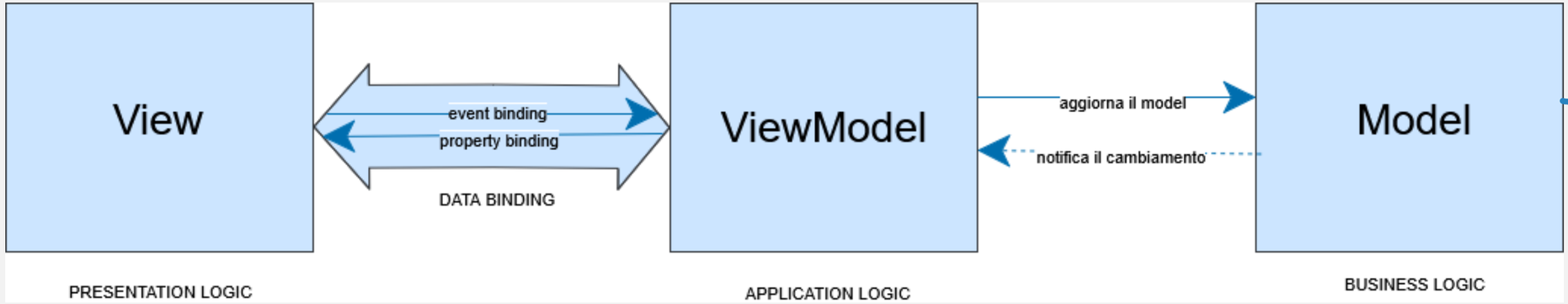
Soluzione:

Incapsulamento della logica in Service Objects. Ogni servizio ha un'unica responsabilità (Single Responsibility Principle), è riutilizzabile e facile da testare.

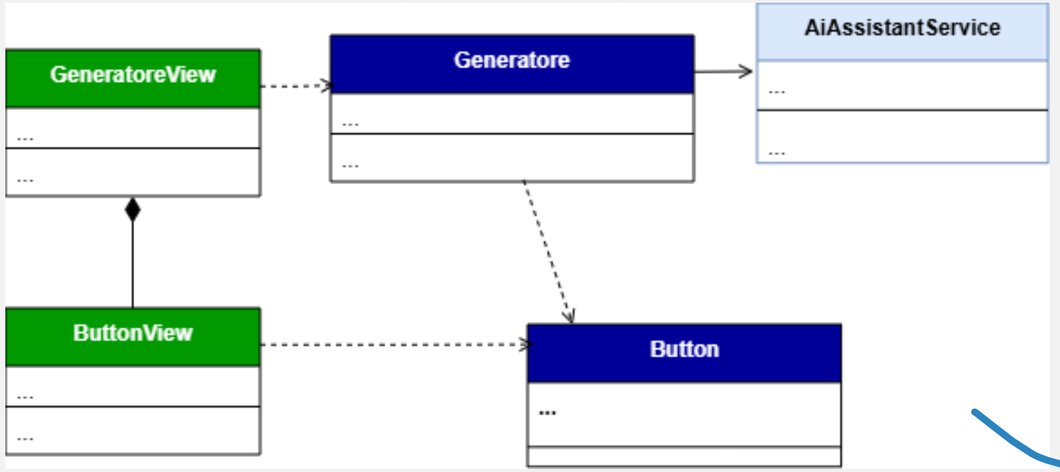


FRONTEND: pattern architetturale

Il Model View ViewModel



Esempio calato nel progetto

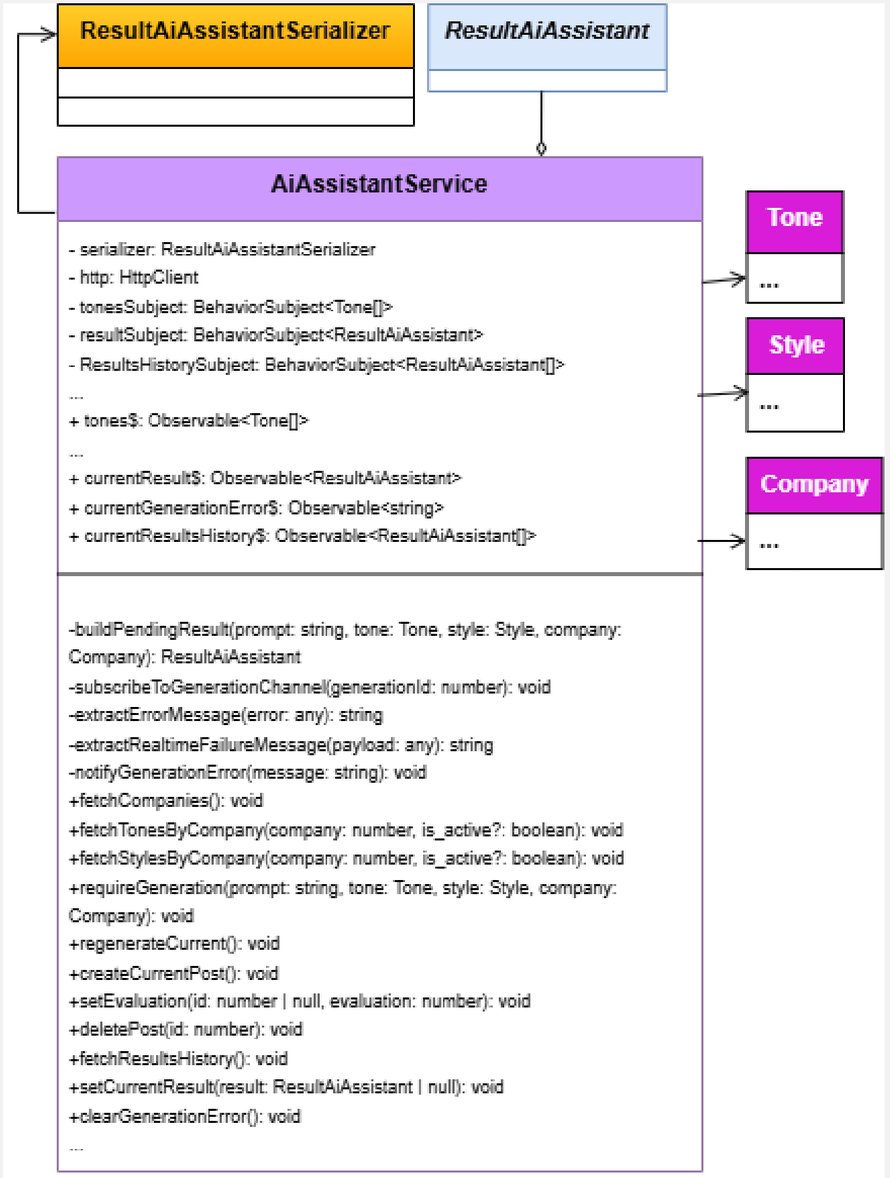


Model: sono i dati e la **logica di business** dell'applicazione. In Angular, questo è spesso rappresentato da **servizi o classi che gestiscono o rappresentano i dati.** (AiAssistantService)

ViewModel: contiene la **logica applicativa** e interagisce con i servizi per recuperare e manipolare i dati in seguito ad eventi della view. In Angular è rappresentato dalle **classi componenti, associate a ciascun template.** (Generatore, Button)

View: I **template Angular** fungono da View, visualizzando i dati e raccogliendo l'input dell'utente. (GeneratoreView, ButtonView)

FRONTEND: service e pattern strutturale



CHE RUOLO HA: coordina chiamate al backend, serializer e aggiornamenti di stato

- Contiene stato interno e mantiene consistenza
- Coordina chiamate HTTP e connessione WebSocket
- Facade verso la ViewModel

TRADE-OFF PULIZIA ARCHITETTURALE VS COMPLESSITÀ:

Una divisione più netta era tecnicamente fattibile ma:

Suddividere in *storage e api services* le responsabilità avrebbe
- Sostituito una semplice chiamata next ad una chiamata al servizio
- Frammentato la logica asincrona tra più layer

Quindi:
- avrebbe reso più fragile la sincronizzazione dello stato (+ forwarding, + mapping, + metodi) senza alleggerire davvero la complessità
- Codice per classe diminuito non di molto

In generale la leggibilità sarebbe peggiorata (complessità)