Piattaforma

1) Il problema da risolvere

Immagina una piccola piattaforma social: ogni utente può seguire altri utenti e ricevere aggiornamenti quando questi pubblicano un post.

La sfida è: come facciamo a inviare le notifiche ai follower senza che l'autore debba preoccuparsi di chi riceverà il messaggio o di quale canale (email, SMS, push) sarà usato?

Serve un sistema flessibile che separi le responsabilità:

- L'autore pubblica un post.
- Il sistema sa chi sono i follower.
- Ogni follower riceve la notifica sul canale che preferisce.

---

2) I pattern coinvolti

Per risolvere il problema, sfruttiamo due pattern molto diffusi:

- **Observer**: permette a un oggetto ("Subject") di informare automaticamente un gruppo di "Observers" quando accade un evento.

  Nella nostra piattaforma, il Subject è l'utente che pubblica, mentre gli Observers sono i follower che vogliono ricevere aggiornamenti.

- **Strategy**: serve a cambiare il comportamento di un oggetto in base a una strategia intercambiabile.

  Qui lo usiamo per inviare la notifica attraverso canali diversi (email, SMS, push), senza cambiare il codice dell'utente.

Questa combinazione di pattern ci aiuta a mantenere il codice pulito, flessibile e facile da estendere.

---

3) I ruoli principali

- **User**: rappresenta una persona sulla piattaforma. Può seguire altri utenti, pubblicare post e ricevere notifiche.
- **Post**: è l'oggetto che incapsula un contenuto pubblicato da un utente.

- **Observable**: è il “motore” dell’Observer pattern. Tiene traccia dei follower e gestisce la lista di chi deve essere notificato.
- **Notifier**: è l’interfaccia comune dei vari canali di notifica (email, SMS, push). Ogni canale implementa la stessa logica di base: inviare un messaggio.
- **Inbox**: è la “casella” personale di un utente, dove finiscono tutte le notifiche ricevute.
- **Notification**: rappresenta una singola notifica arrivata (con informazioni su mittente, destinatario, canale, data, stato di lettura).
- **LoggableMixin**: aggiunge la capacità di tracciare in un log tutto ciò che succede (es. quando un utente segue un altro, pubblica un post o riceve una notifica).

---

4) Come funziona il flusso

1. **Follow**: Alice decide di seguire Bob. Il suo nome viene aggiunto all’elenco di osservatori di Bob.
2. **Post**: Bob pubblica un post. Questo evento viene passato al sistema di notifiche.
3. **Notify**: il sistema controlla chi sono i follower di Bob (per esempio, Alice).
4. **Receive**: ciascun follower riceve il messaggio nel proprio canale preferito (email, SMS o push).
5. **Inbox**: la notifica viene salvata nell’inbox del follower, pronta per essere letta.
6. **Statistiche**: l’inbox permette di vedere quante notifiche non lette ci sono, filtrare per canale, esportare i dati in JSON o CSV, ecc.
7. **Logging**: ogni azione (follow, post, ricezione) viene registrata in automatico nel log.

---

5) Collegamento con le milestone

- **M1**: creiamo il modello base. Gli utenti possono seguire, smettere di seguire e pubblicare post.

- **M2**: introduciamo l'Observable, che tiene traccia dei follower e sa a chi inviare notifiche.
- **M3**: aggiungiamo le strategie di notifica: email, SMS e push. Ogni utente sceglie il proprio canale preferito.
- **M4**: costruiamo l'inbox per salvare le notifiche ricevute.
- **M5**: arricchiamo l'inbox con funzioni di ricerca e statistiche (filtri per utente, canale, non lette).
- **M6**: rendiamo il sistema persistente, con la possibilità di esportare e importare notifiche da file JSON e CSV.
- **M7**: aggiungiamo il mixin di logging per avere traccia di tutto ciò che accade.
- **M8**: perfezioniamo il sistema con confronti e ordinamenti sulle notifiche.

M1 — Modello base e follow

- In `User.follow(self, other)`: registra `self.username` in `other.attach(self.username)`.
- In `User.unfollow(self, other)`: `other.detach(self.username)`.
- In `User.post(self, content)`: crea e **ritorna** `Post(self.username, content)` (non notifica ancora).

  Riferimento al test `test_m1`.

M2 — Observable + notify

- In `Observable.attach/detach/observers`: usa un `set[str]` interno.
- In `Observable.notify(self, post) → list[str]`: ritorna la lista dei follower (stringhe).

  Nota: la propagazione "vera" ai canali accade in M3/M4 tramite `receive()`.
  Vedi `test_m2`.

M3 — Strategy dei canali

- Implementa `EmailNotifier/SMSNotifier/PushNotifier.send()` creando e **ritornando** `Notification(channel, to, message)`.
- In `User.receive(self, message)`: `notif = self.preferred.send(self.username, message)` poi `self.inbox.add(notif)` e ritorna `notif`.

  Vedi `test_m3`.

M4 — Inbox + dunder

- `__len__`: `return len(self._items)`
- `__contains__`: membership su `self._items`.
- `add`: appende una `Notification`.

  Vedi `test_m4`.

M5 — Filtri/Statistiche

- `by_user(username)`: filtra su `n.to == username`.
- `by_channel(channel)`: filtra su `n.channel == channel`.
- `unseen_count()`: conta `not n.seen`.

  Vedi `test_m5`.

M6 — Persistenza JSON/CSV

- **Export JSON:** serializza con `asdict()` (dataclasses) e `datetime.isoformat()`.
- **Import JSON:** ricrea `Notification`, evita duplicati su `(channel, to, message)`; ritorna quanti aggiunti.
- **Export/Import CSV:** analoghi, con intestazioni coerenti.

  Vedi `test_m6`.

M7 — Mixin di logging

- `LoggableMixin.log(event)`: `self._log.append(f"{datetime.utcnow().isoformat()} {event}")`.
- Chiamalo nei punti salienti: `follow/unfollow/post/receive/notify`.

  Vedi `test_m7`.

M8 — Confronti/ordinamenti

- `Notification.__eq__` già definito (stesso canale/destinatario/messaggio).
- Ordinabilità per data (facoltativa): `sorted(inbox._items, key=lambda n: n.created_at)`.

  Vedi `test_m8`.
