Piattaforma di Notifiche (Observer + Strategy)

Guida didattica alla **soluzione M1..M8** con spiegazioni e **porzioni di codice** mirate. Per ogni blocco: prima la spiegazione, poi lo snippet.

---

1) Modello di dominio: `Post` e `Notification`

Perché: modelliamo gli eventi fondamentali. `Post` rappresenta ciò che scatena le notifiche; `Notification` è l'esito dell'invio sul canale scelto.

- `Post` è un semplice value object con `author`, `content` e timestamp automatico.
- `Notification` include `channel`, `to`, `message`, `created_at`, `seen` e ridefinisce `__eq__` per considerare uguali notifiche con stessi canale/destinatario/messaggio.

```
@dataclass
class Post:
    author: str
    content: str
    created_at: datetime = field(default_factory=datetime.utcnow)

@dataclass
class Notification:
    channel: str
    to: str
    message: str
    created_at: datetime = field(default_factory=datetime.utcnow)
    seen: bool = False

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Notification):
            return NotImplemented
        return (self.channel, self.to, self.message) == (other.channel,
other.to, other.message)
```

Note didattiche - `default_factory=datetime.utcnow` evita di fissare la data alla definizione della classe. - L'uguaglianza basata su attributi semplifica la deduplica in import/export.

---

2) Inbox: composizione, dunder e utility

Perché: separiamo la **raccolta** delle notifiche dalla logica di invio/ricezione. L'`Inbox` offre un'API pulita, dunder per un uso naturale e funzioni di filtro/statistiche.

```
class Inbox:
    def __init__(self) -> None:
```

```
        self._items: List[Notification] = []

    def __len__(self) -> int:                # M4
        return len(self._items)

    def __contains__(self, item: Notification) -> bool:  # M4
        return any(n == item for n in self._items)

    def add(self, notif: Notification) -> None:          # API base
        self._items.append(notif)

    def by_user(self, username: str) -> List[Notification]:  # M5
        return [n for n in self._items if n.to == username]

    def by_channel(self, channel: str) -> List[Notification]: # M5
        ch = channel.lower()
        return [n for n in self._items if n.channel.lower() == ch]

    def unseen_count(self) -> int:                           # M5
        return sum(1 for n in self._items if not n.seen)
```

Note didattiche - `__len__` e `__contains__` rendono l'oggetto "collezione-like". - I filtri operano per attributo, mantenendo l'`Inbox` indipendente dal resto del sistema.

---

3) Persistenza: JSON/CSV con deduplica (M6)

Perché: testiamo serializzazione e robustezza I/O. Si esporta in formato leggibile; in import si **deduplica** basandosi su `(channel, to, message)`.

```
    def export_json(self, path: str) -> None:
        payload = []
        for n in self._items:
            d = asdict(n)
            d["created_at"] = n.created_at.isoformat()
            payload.append(d)
        with open(path, "w", encoding="utf-8") as f:
            json.dump(payload, f, ensure_ascii=False, indent=2)

    def import_json(self, path: str) -> int:
        if not os.path.exists(path):
            return 0
        with open(path, "r", encoding="utf-8") as f:
            data = json.load(f)
        sigs = {(n.channel, n.to, n.message) for n in self._items}
        added = 0
        for d in data:
            sig = (d.get("channel", ""), d.get("to", ""), d.get("message",
"",))
```

```
            if sig in sigs:
                continue
            ts = d.get("created_at")
            try:
                created = datetime.fromisoformat(ts) if ts else
datetime.utcnow()
            except Exception:
                created = datetime.utcnow()
            self._items.append(Notification(sig[0], sig[1], sig[2], created,
bool(d.get("seen", False))))
            sigs.add(sig)
            added += 1
        return added
```

Note didattiche - `asdict` dalle dataclass semplifica l'export. - `isoformat()` mantiene il timestamp in un formato standard. - La deduplica è un **contratto** utile anche per evitare duplicati tra diversi canali di import.

---

4) Observer: `Observable` e `notify` (M2)

Perché: disaccoppiamo l'autore dai follower. L'`Observable` gestisce l'elenco degli osservatori (username) e, su un nuovo `Post`, fornisce la lista dei destinatari da notificare.

```
class Observable:
    def __init__(self) -> None:
        self._observers: set[str] = set()

    def attach(self, username: str) -> None:   # M2
        self._observers.add(username)

    def detach(self, username: str) -> None:   # M2
        self._observers.discard(username)

    def observers(self) -> List[str]:          # M2
        return sorted(self._observers)

    def notify(self, post: Post) -> List[str]: # M2
        return self.observers()
```

Note didattiche - In M2 `notify` restituisce gli username. L'invio reale avviene in M3 attraverso la Strategy del canale. - L'uso di `set` impedisce duplicati tra osservatori.

---

5) Strategy: `Notifier` (Protocol) e canali concreti (M3)

Perché: isoliamo il **come** inviare la notifica (email/SMS/push) dal **quando** inviarla. Cambiare canale non tocca il resto del sistema.

```python
@runtime_checkable
class Notifier(Protocol):
    channel: str
    def send(self, to: str, message: str) -> Notification: ...

class EmailNotifier:
    channel = "email"
    def send(self, to: str, message: str) -> Notification:
        return Notification(self.channel, to, message)

class SMSNotifier:
    channel = "sms"
    def send(self, to: str, message: str) -> Notification:
        return Notification(self.channel, to, message)

class PushNotifier:
    channel = "push"
    def send(self, to: str, message: str) -> Notification:
        return Notification(self.channel, to, message)
```

Note didattiche - `Protocol` (typing) consente **duck typing statico**: qualsiasi oggetto con `channel` e `send()` è accettato. - Le tre classi concrete incapsulano differenze di canale senza cambiare l'API.

---

6) Logging cross-cutting: `LoggableMixin` (M7)

Perché: separiamo la **tracciabilità** dalla logica core. Il mixin fornisce `log()` e `get_log()` riutilizzabili da più classi.

```python
class LoggableMixin:
    def __init__(self) -> None:
        self._log: List[str] = []

    def log(self, event: str) -> None:
        self._log.append(f"{datetime.utcnow().isoformat()} {event}")

    def get_log(self) -> List[str]:
        return list(self._log)
```

Note didattiche - Pattern **Mixin**: ereditarietà orizzontale per "aggiungere" comportamenti comuni senza gerarchie profonde.

---

7) Utente: orchestrazione e punto di estensione (M1, M3)

Perché: `User` è il punto d'incontro dei pattern. Eredita `Observable` (ha dei follower) e `LoggableMixin` (traccia eventi), possiede `inbox` e una `preferred` strategy.

```
class User(LoggableMixin, Observable):
    def __init__(self, username: str, preferred: Optional[Notifier] = None) -
> None:
        LoggableMixin.__init__(self)
        Observable.__init__(self)
        self.username = username
        self.preferred: Notifier = preferred or EmailNotifier()
        self.inbox = Inbox()

    def follow(self, other: "User") -> None:  # M1
        other.attach(self.username)
        self.log(f"follow {other.username}")

    def unfollow(self, other: "User") -> None:  # M1
        other.detach(self.username)
        self.log(f"unfollow {other.username}")

    def post(self, content: str) -> Post:       # M1
        self.log("post")
        return Post(author=self.username, content=content)

    def receive(self, message: str) -> Notification:  # M3
        notif = self.preferred.send(self.username, message)
        self.inbox.add(notif)
        self.log(f"receive via {notif.channel}")
        return notif
```

Note didattiche - **Composizione**: `User` possiede un'`Inbox`. - **Polimorfismo**: `preferred` può essere cambiato a runtime con qualsiasi implementazione di `Notifier`. - Separazione netta tra **creazione del post** (M1), **determinazione dei follower** (M2) e **invio su canale** (M3).

---

8) Test e harness (verifica M1..M8)

Perché: l'harness di test guida lo sviluppo per milestone, facilita il TDD e fornisce feedback immediato.

```
def _safe_test(name: str, fn):
    try:
        fn(); print(f"✅ {name}"); return True
    except AssertionError as e:
        print(f"❌ {name} — Test fallito: {e}"); return False
    except Exception as e:
        print(f"❌ {name} — Errore inaspettato: {type(e).__name__}: {e}");
return False
```

Note didattiche - Ogni milestone ha un test isolato; `_safe_test` cattura eccezioni e produce un output leggibile per lo studente.

---

9) Mappa concettuale rapida

- **Observer**: `User` (Subject) → `Observable` (lista follower) → `notify()` restituisce i destinatari.
- **Strategy**: `Notifier` (Protocol) → `Email/SMS/PushNotifier` (implementazioni).
- **Composizione**: `User` → `Inbox` per gestire le `Notification`.
- **Mixin**: `LoggableMixin` aggiunge tracciabilità senza inquinare le classi dominio.
- **Dunder**: `__len__`, `__contains__` per un oggetto collezione naturale.
- **Persistenza**: export/import JSON/CSV con deduplica.