Collegamenti alle classi Concrete di Observer Pattern

Osservando il diagramma e la descrizione del sistema Ticketoo, metterei il **Ticket sulla sottoclasse ConcreteSubject** (come già mostrato nel diagramma).

Ecco il ragionamento:

Analisi del Pattern

Il sistema implementa il **Pattern Observer** combinato con il **Pattern Proxy**. La struttura mostra:

- `Subject` : interfaccia astratta per la gestione degli observer
- `ConcreteSubject` : implementazione concreta che gestisce lo stato e le notifiche
- `LocalSubject` e `ProxySubject` : specializzazioni per gestire rispettivamente operazioni locali e remote

Motivazioni per la Collocazione su ConcreteSubject

1. **Responsabilità Specifica**: Il `Ticket` rappresenta il dato di business centrale del sistema. ConcreteSubject è il livello appropriato dove gestire lo stato concreto dell'applicazione.
2. **Pattern Proxy**: La presenza di `ProxySubject` suggerisce che il sistema simula comunicazione locale per operazioni remote (come descritto: "dialogando come se entrambe stessero eseguendo sulla stessa macchina"). Il ticket è l'entità che viene trasferita/elaborata in questo processo.
3. **Separazione delle Responsabilità**:
   - `Subject` si occupa solo della gestione degli observer
   - `ConcreteSubject` gestisce il ticket e notifica i cambiamenti di stato
   - Le sottoclassi (`LocalSubject` / `ProxySubject`) specializzano il comportamento per contesti diversi

Alternative Non Ottimali

Mettere il ticket su `Subject` renderebbe l'interfaccia troppo specifica per questo dominio, violando il principio di riusabilità del pattern Observer.

La collocazione attuale mantiene una chiara separazione tra i meccanismi di notifica (Observer) e la logica di business (gestione ticket), garantendo un'architettura più modulare e manutenibile.

No, sono pattern per scopi completamente diversi!

Non è che Command sia "più generale" - sono pattern **ortogonali** che risolvono problemi diversi:

Command Pattern

COSA FA: Trasforma un'azione in un oggetto **PERCHÉ**: Per poter manipolare, memorizzare, annullare l'azione

```
// Command = "Un'azione impacchettata"
interface Command {
    void execute();
    void undo();
}

class CompraTicketCommand implements Command {
    private Ticket ticket;
    private User user;

    public void execute() { /* compra */ }
    public void undo() { /* rimborsa */ }
}

// Uso: posso mettere il command in una lista, in una coda, etc.
List<Command> history = new ArrayList<>();
Command cmd = new CompraTicketCommand(ticket, user);
cmd.execute();
history.add(cmd); // per undo successivo
```

Template Method Pattern

COSA FA: Definisce la struttura di un algoritmo **PERCHÉ**: Per avere processi fissi con passi personalizzabili

```
// Template = "Ricetta con variazioni"
abstract class ProcessaPagamento {
    public final void processaPagamento() {  // STRUTTURA FISSA
        validaDati();
        autenticaUtente();
        elaboraPagamento();    // ← QUESTO cambia
        inviaRicevuta();
    }

    protected abstract void elaboraPagamento(); // ← QUI ogni sottoclasse fa
diverso
}
```

Esempio Concreto: Sistema Ticketoo

SCENARIO: Vendere un ticket

Con Command:

```
// Impacchetto l'azione di vendita
class VendiTicketCommand implements Command {
    public void execute() {
        // tutta la logica di vendita
    }
    public void undo() {
        // annulla vendita
    }
}

// Posso metterlo in coda, fare undo, etc.
VendiTicketCommand cmd = new VendiTicketCommand();
```

Con Template:

```
// Struttura fissa per processare qualsiasi ticket
abstract class ProcessaTicket {
    public final void processa() {
        scaricaTicket();
        estraiCodice();      // ← ogni provider fa diverso
        validaCodice();
        notifica();
    }

    protected abstract void estraiCodice(); // Ticketmaster vs Eventbrite
}
```

La Differenza Chiave

- **Command**: "Voglio eseguire/annullare/memorizzare un'AZIONE"
- **Template**: "Voglio un PROCESSO standard con variazioni specifiche"

Non sono alternativi - puoi usarli insieme! Un Command può implementare un Template Method al suo interno.

Quindi no, Command non è "più generale" - sono strumenti diversi per problemi diversi.

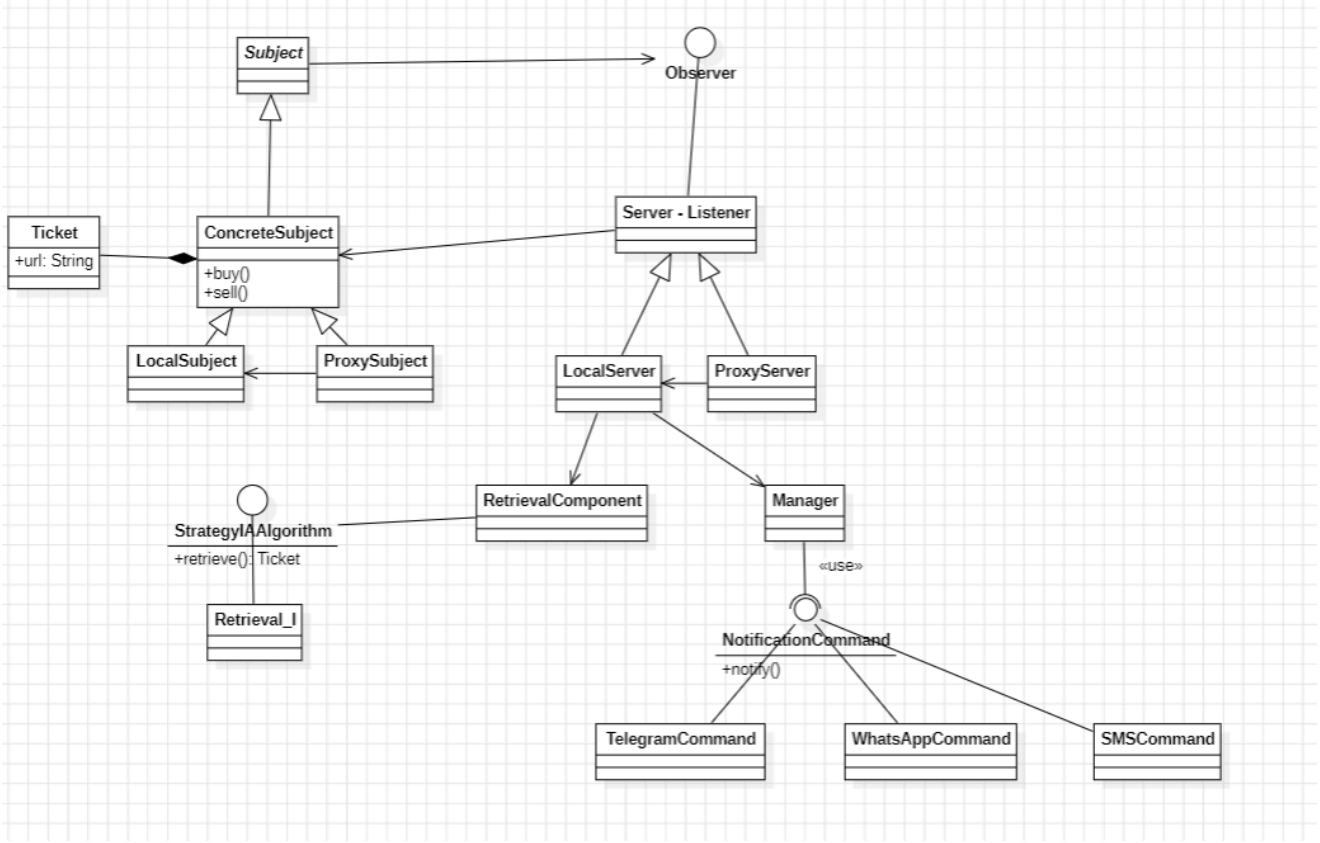
An **association** almost always implies that one object has the other object as a field/property/attribute (terminology differs).

A **dependency** typically (but not always) implies that an object accepts another object as a method parameter, instantiates, or uses another object. A **dependency** is very much implied by an **association**.

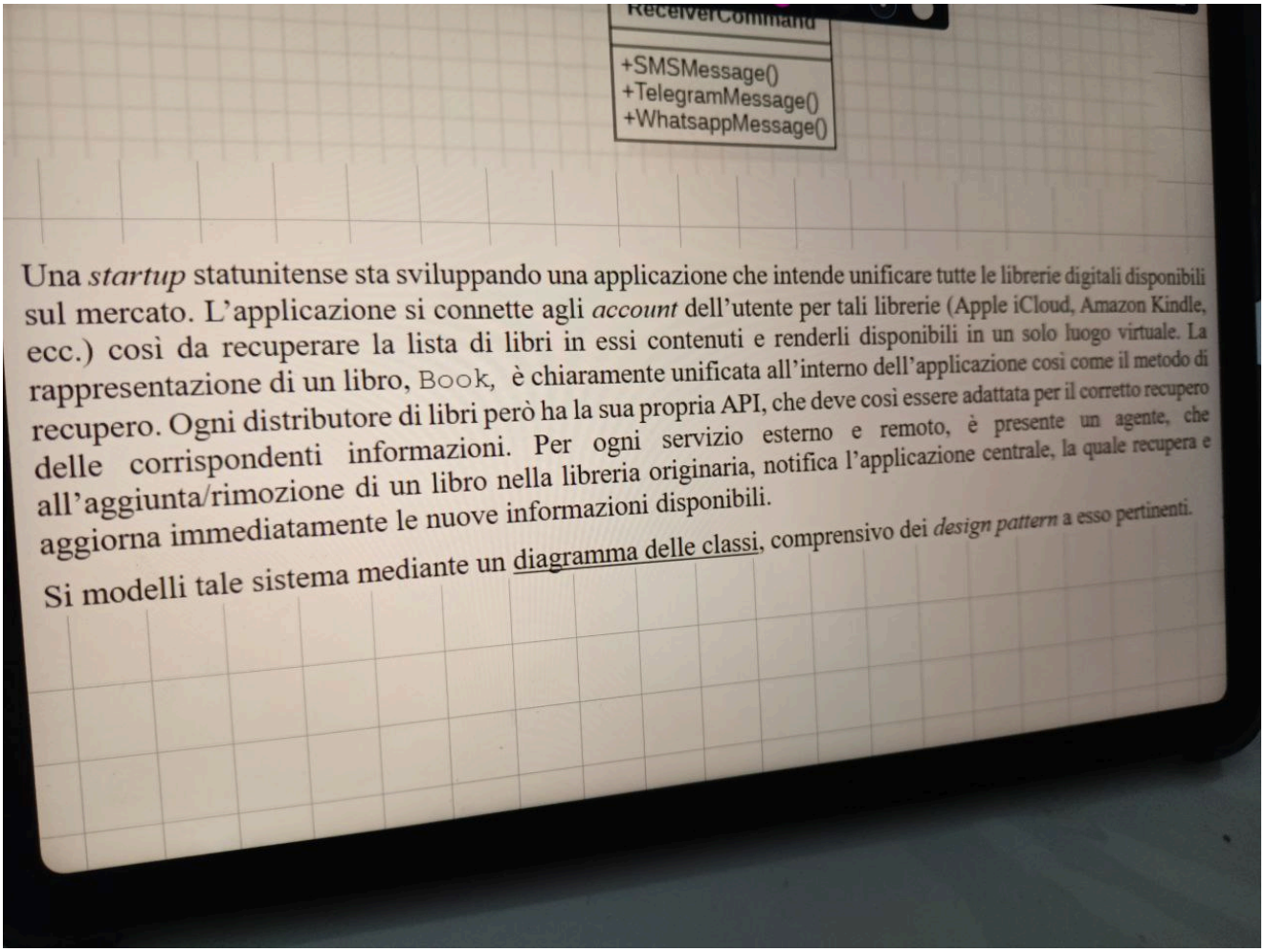
Modellazione 1 - Ticketoo

Gli eventi *live* sono particolarmente esposti al fenomeno della rivendita di biglietti. È normale infatti che una persona che abbia comprato un biglietto per un evento con mesi di anticipo non si trovi più nelle condizioni di potervi partecipare. Ticketoo è una delle piattaforme che offrono servizi di rivendita. L'applicazione *mobile* di Ticketoo consente di vendere un *e-ticket* ancora valido, già precedentemente acquistato, fornendo il corrispondente URL. Il *server* riceve queste informazioni dialogando con l'applicazione come se entrambe stessero eseguendo sulla stessa macchina. A quel punto, un componente dedicato recupera l'*e-ticket*, il quale poi successivamente viene elaborato da un algoritmo di IA, che ne estrae il codice identificativo. Questo algoritmo necessita di costante aggiornamento a causa delle modifiche frequentemente applicate dai *provider* di *e-ticket* per evitare le contraffazioni. Quando l'*e-ticket* sarà stato venduto, il sistema invierà un messaggio di notifica al corrispondente venditore. I *media* possibili per queste notifiche sono SMS, *Whatsapp*, o *Telegram*. Il messaggio contiene un *link* alla pagina *web* relativa alla vendita. Per tutti e tre i *media* citati, il sistema utilizza la medesima interfaccia di invio.

Si modelli il sistema sopra descritto mediante un diagramma delle classi, comprensivo dei *design pattern* a esso pertinenti.

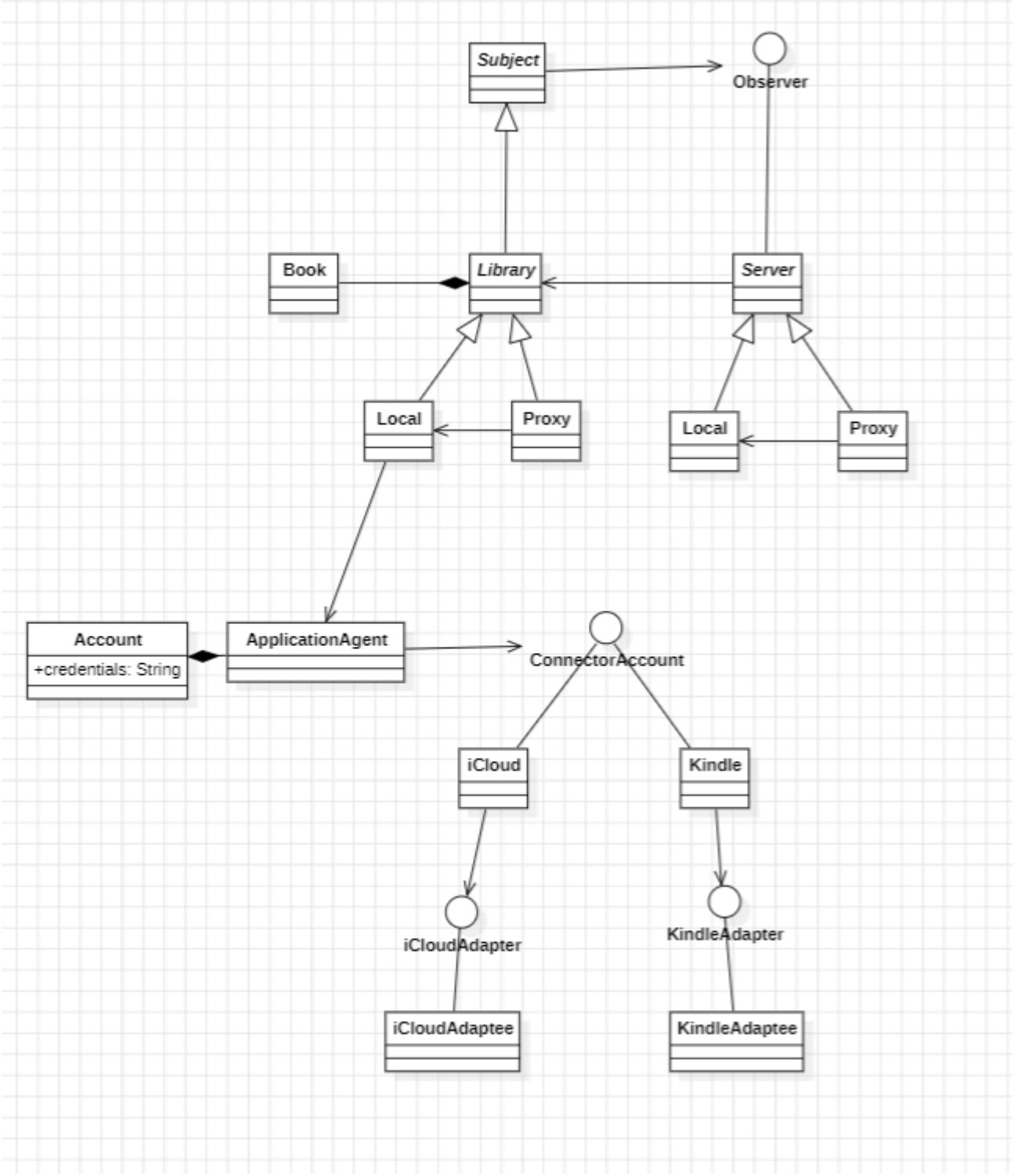


Modellazione 2 - Startup USA



Una *startup* statunitense sta sviluppando una applicazione che intende unificare tutte le librerie digitali disponibili sul mercato. L'applicazione si connette agli *account* dell'utente per tali librerie (Apple iCloud, Amazon Kindle, ecc.) così da recuperare la lista di libri in essi contenuti e renderli disponibili in un solo luogo virtuale. La rappresentazione di un libro, `Book`, è chiaramente unificata all'interno dell'applicazione così come il metodo di recupero. Ogni distributore di libri però ha la sua propria API, che deve così essere adattata per il corretto recupero delle corrispondenti informazioni. Per ogni servizio esterno e remoto, è presente un agente, che all'aggiunta/rimozione di un libro nella libreria originaria, notifica l'applicazione centrale, la quale recupera e aggiorna immediatamente le nuove informazioni disponibili.

Si modelli tale sistema mediante un diagramma delle classi, comprensivo dei *design pattern* a esso pertinenti.


Subject
Observer
Book
Library
Server
Local
Proxy
Local
Proxy
Account
+credentials: String
ApplicationAgent
ConnectorAccount
iCloud
Kindle
iCloudAdapter
KindleAdapter
iCloudAdaptee
KindleAdaptee