



**Università  
di Catania**

**UNIVERSITY OF CATANIA**

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

A MASTER OF SCIENCE DEGREE IN COMPUTER SCIENCE.

---

*Alfio Spoto*

Ingegneria dei sistemi distribuiti

---

APPUNTI LEZIONI ISD  
I

---

---

Academic Year 2024 - 2025

# Contents

<b>1 Il Design Pattern Proxy</b>	<b>1</b>
1.1 Intento e Motivazione . . . . .	1
1.2 Soluzione e Struttura . . . . .	1
1.2.1 Componenti . . . . .	1
1.2.2 Esempio: Il Virtual Proxy . . . . .	2
1.3 Approfondimento sulle Varianti . . . . .	3
1.3.1 Protection Proxy . . . . .	3
1.3.2 Proxy Multipli . . . . .	4
1.3.3 Altre Varianti Comuni . . . . .	4
1.4 Conseguenze Generali: Pro e Contro . . . . .	5
1.4.1 Benefici (Pro) . . . . .	5
1.4.2 Svantaggi e Considerazioni (Contro) . . . . .	5
<b>2 Il Design Pattern Reference Monitor</b>	<b>7</b>
2.1 Intento e Problema . . . . .	7
2.2 Soluzione e Struttura . . . . .	7
2.2.1 Componenti . . . . .	7
2.2.2 Focus: La classe Request . . . . .	8
2.3 Flusso di Esecuzione . . . . .	8
2.4 Conseguenze: Pro e Contro . . . . .	9
2.4.1 Benefici (Pro) . . . . .	9
2.4.2 Svantaggi e Considerazioni (Contro) . . . . .	9
2.5 Implementazione e Relazione con il Proxy . . . . .	9
<b>3 Role-Based Access Control (RBAC)</b>	<b>11</b>
3.1 Intento e Problema . . . . .	11
3.2 Soluzione: L'Astrazione del Ruolo . . . . .	11
3.3 Struttura e Componenti . . . . .	12
3.3.1 Componenti Chiave . . . . .	12
3.3.2 Focus: Il Ruolo della Classe Session . . . . .	12
3.4 Flusso di Esecuzione . . . . .	13
3.4.1 Controllo dei Permessi . . . . .	13
3.4.2 Creazione e Configurazione . . . . .	13
3.5 Conseguenze: Pro e Contro . . . . .	14
3.5.1 Benefici (Pro) . . . . .	14
3.5.2 Svantaggi e Considerazioni (Contro) . . . . .	14

<b>4 Autenticazione Basata su Token</b>	<b>15</b>
4.1 Contesto e Problema . . . . .	15
4.2 Soluzione: Il Flusso di Autenticazione . . . . .	15
4.2.1 Componenti del Flusso . . . . .	15
4.2.2 Flusso di Esecuzione . . . . .	16
4.3 Analisi Dettagliata del Token . . . . .	16
4.3.1 Definizione e Formati . . . . .	16
4.3.2 Contenuto Tipico (Claims) . . . . .	17
4.4 Conseguenze e Considerazioni . . . . .	17
4.4.1 Benefici (Pro) . . . . .	17
4.4.2 Svantaggi e Considerazioni (Contro) . . . . .	17
<b>5 Autenticazione e Protezione dei Servizi</b>	<b>18</b>
5.1 Il Design Pattern Authenticator . . . . .	18
5.1.1 Intento e Problema . . . . .	18
5.1.2 Soluzione e Struttura . . . . .	18
5.1.3 Flusso di Esecuzione . . . . .	19
5.1.4 Conseguenze e Varianti . . . . .	20
5.2 Gestione Sicura delle Password: Hashing . . . . .	20
5.2.1 Proprietà di una Funzione di Hashing . . . . .	20
5.2.2 La Necessità di Funzioni "Lente" . . . . .	21
5.2.3 Hashing con Salt . . . . .	21
5.3 Implementazione Pratica: Scrypt e Guava . . . . .	21
5.3.1 La Libreria Scrypt per l'Hashing . . . . .	21
5.3.2 Protezione da Attacchi DoS con Rate-Limiting . . . . .	22
5.4 Progettazione di un Flusso Completo . . . . .	23
5.4.1 Flusso di Log In . . . . .	23
5.4.2 Flusso per Richieste Successive . . . . .	24
5.5 Conseguenze Generali: Pro e Contro . . . . .	24
5.5.1 Benefici (Pro) . . . . .	25
5.5.2 Svantaggi e Considerazioni (Contro) . . . . .	25

# Chapter 1

## Il Design Pattern Proxy

### 1.1 Intento e Motivazione

L'intento principale del design pattern Proxy è quello di definire un sostituto o surrogato (surrogate) per un altro oggetto, al fine di controllare gli accessi all'oggetto target. In questo schema, i client comunicano con il surrogato anziché interagire direttamente con l'oggetto di destinazione.

La motivazione fondamentale è che l'accesso all'oggetto target (il **RealSubject**) dovrebbe essere trasparente e semplice per il client. Ad esempio, si può voler nascondere al client che un'immagine pesante viene creata solo nel momento in cui serve effettivamente, per non complicare la logica del client e ridurre i tempi di caricamento iniziali.

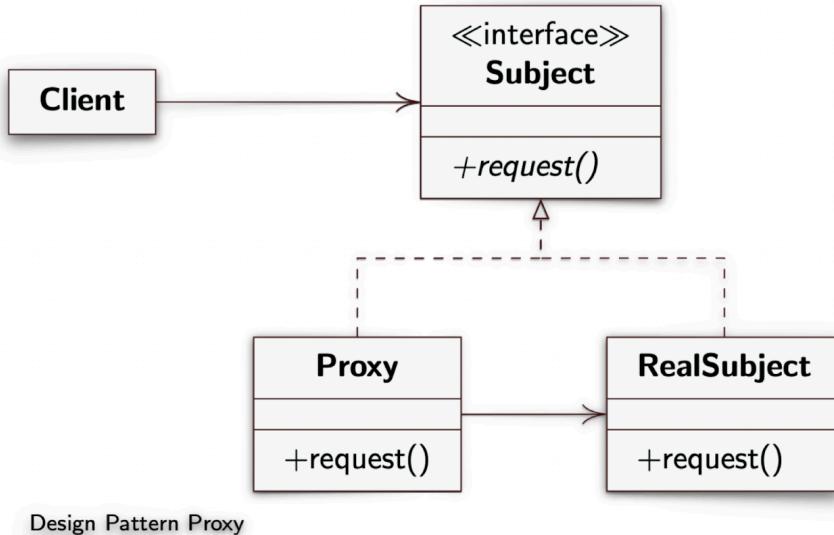
### 1.2 Soluzione e Struttura

La soluzione consiste nell'usare un oggetto Proxy che si frappone tra gli utilizzatori (Client) e l'oggetto target (RealSubject). Questo Proxy offre la stessa interfaccia dell'oggetto target, permettendo al client di interagire con esso in modo trasparente. Il Proxy può quindi aggiungere computazioni utili prima o dopo la chiamata al metodo reale, come ad esempio il controllo degli accessi.

#### 1.2.1 Componenti

Il pattern è composto dai seguenti elementi principali:

- **Subject:** Definisce un'interfaccia comune sia per il **RealSubject** che per il **Proxy**. Questo è ciò che permette al **Proxy** di essere usato al posto dell'oggetto reale.
- **RealSubject:** È l'oggetto target, quello che fornisce il servizio effettivo che il client desidera utilizzare.
- **Proxy:** Contiene un riferimento che permette l'accesso al **RealSubject**; fornisce la stessa interfaccia del **Subject** e gestisce l'accesso corretto all'oggetto reale.
- **Client:** È l'utilizzatore che usa il **Proxy** per accedere al servizio.

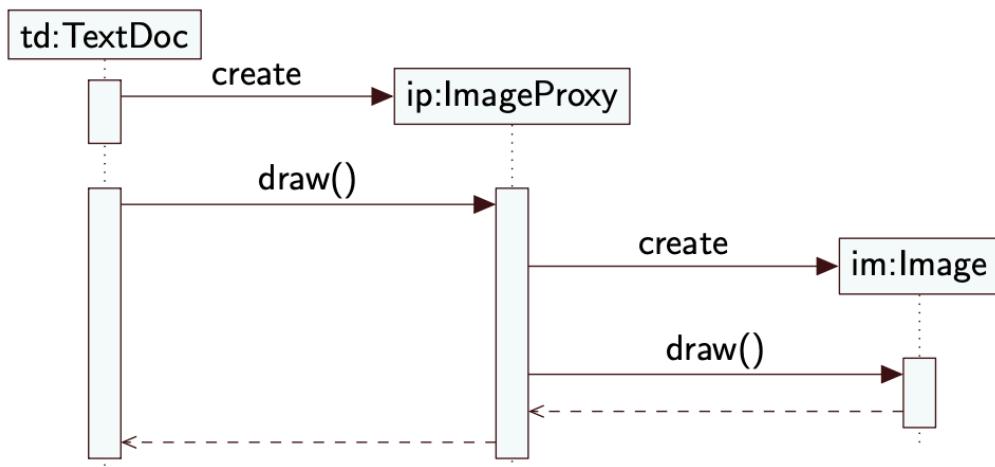


**Figure 1.1:** Diagramma UML delle classi del pattern Proxy generico. Il Client dipende solo dall'interfaccia Subject, implementata sia da Proxy che da RealSubject.

### 1.2.2 Esempio: Il Virtual Proxy

Un esempio classico di utilizzo è il **Virtual Proxy**. In questo scenario, l'oggetto **RealSubject** (es. un'immagine) è costoso da creare. Il **Proxy** (es. **ImageProxy**) rimanda la creazione dell'oggetto reale fino a quando non è strettamente necessario (es. quando il metodo **draw()** viene invocato).

Come mostrato nel diagramma di sequenza, quando il **TextDoc** (Client) invoca **draw()** sull'**ImageProxy**, è il proxy che si occupa di creare l'istanza di **Image** (RealSubject) solo la prima volta, per poi inoltrare la chiamata **draw()**.



**Figure 1.2:** Diagramma di sequenza di un Virtual Proxy per il caricamento di immagini.

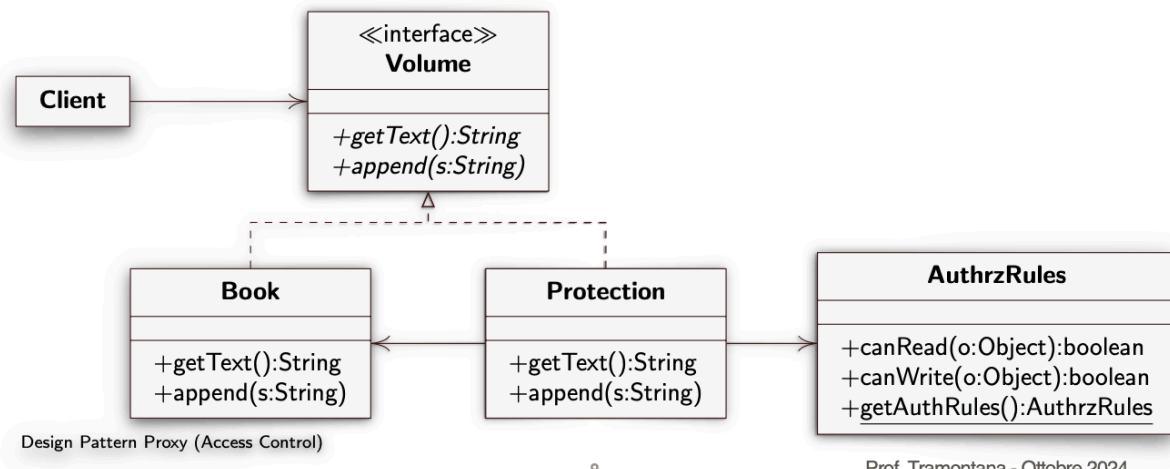
## 1.3 Approfondimento sulle Varianti

Il pattern Proxy non è monolitico. L'indirezione introdotta può essere usata per diversi scopi. Analizziamo le varianti più significative basate sul materiale di studio.

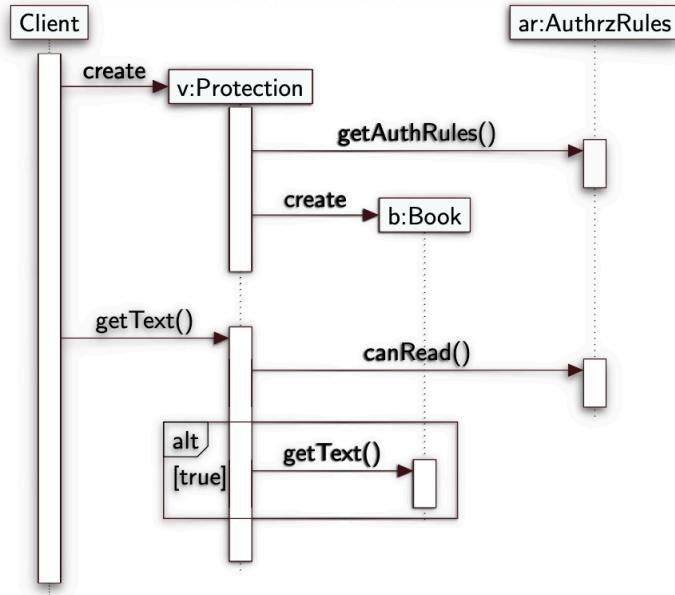
### 1.3.1 Protection Proxy

Il **Protection Proxy** viene utilizzato per implementare, in modo separato, politiche di protezione degli accessi a certi oggetti. Il proxy agisce come un controllore, filtrando le richieste.

Prima di inoltrare la chiamata al **RealSubject** (es. **Book**), il proxy (**Protection**) consulta un set di regole (es. **AuthrzRules**) per verificare se il client ha i permessi necessari (es. `canRead()` o `canWrite()`). Le richieste non autorizzate non vengono inoltrate all'oggetto reale.



**Figure 1.3:** Diagramma UML delle classi di un *Protection Proxy*. Il proxy **Protection** controlla **AuthrzRules** prima di accedere a **Book**.

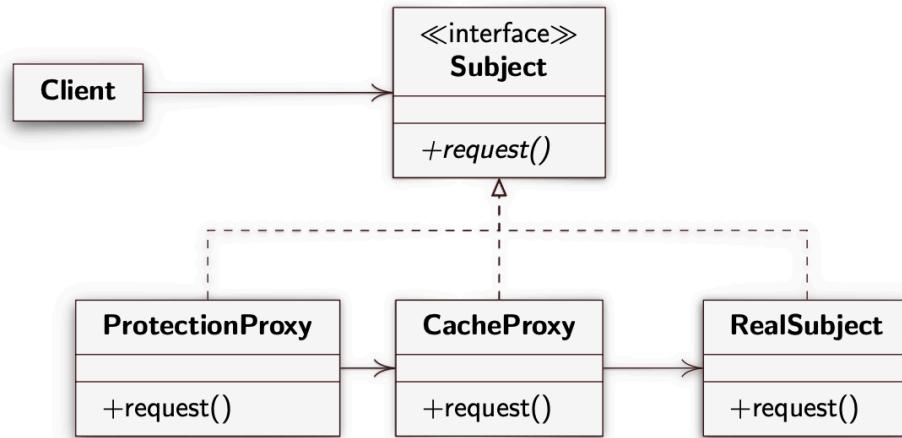


**Figure 1.4:** Diagramma di sequenza di un *Protection Proxy*. La chiamata `getText()` del client viene prima validata dal proxy tramite `canRead()`.

### 1.3.2 Proxy Multipli

È possibile avere vari Proxy che agiscono sullo stesso *RealSubject*, spesso "incatenati" (chaining). Ogni proxy aggiunge un comportamento specifico.

Come mostrato nello schema, una richiesta del client potrebbe passare prima attraverso un *ProtectionProxy* (per il controllo accessi) e, se autorizzata, passare a un *CacheProxy* (per l'ottimizzazione), che infine colpirà il *RealSubject* solo se il dato non è in cache.



**Figure 1.5:** Diagramma UML che mostra la possibilità di combinare più proxy (es. *ProtectionProxy* e *CacheProxy*) per lo stesso *RealSubject*.

### 1.3.3 Altre Varianti Comuni

Oltre a quelle analizzate, le slide menzionano altre varianti note:

- **Remote Proxy:** Nasconde al client il fatto che l'oggetto reale risiede su un host remoto. Il proxy si occupa di serializzare la richiesta, inviarla sulla rete e deserializzare la risposta.
- **Cache Proxy:** Memorizza temporaneamente i risultati di operazioni onerose per ottimizzare le prestazioni.
- **Copy-on-Write:** Ottimizza la copia di oggetti. Il proxy rimanda la duplicazione fisica dell'oggetto al momento in cui questo viene effettivamente modificato.

## 1.4 Conseguenze Generali: Pro e Contro

L'applicazione del pattern Proxy ha implicazioni significative sul design del software. La conseguenza principale è che **il Proxy introduce un livello di indirezione**, e da questa singola decisione scaturiscono tutti i vantaggi e gli svantaggi del pattern.

### 1.4.1 Benefici (Pro)

- **Trasparenza per il Client:** L'accesso all'oggetto target rimane trasparente e semplice per il client. Il client non deve cambiare il modo in cui chiama gli oggetti che usa, poiché il Proxy e il RealSubject condividono la stessa interfaccia Subject.
- **Ottimizzazione (Creazione Onerosa):** Permette di ottimizzare la creazione di oggetti costosi. Il *Virtual Proxy* è l'esempio perfetto, creando l'oggetto RealSubject solo quando è strettamente necessario (es. *lazy loading*).
- **Separazione delle Responsabilità (Sicurezza):** Consente di implementare politiche di protezione degli accessi (come autenticazione e autorizzazione) o di gestione della concorrenza (sincronizzazione tramite lock) in modo **separato** dalla logica di business del RealSubject.
- **Trasparenza di Locazione (Remote Proxy):** Permette di nascondere al client la complessità della comunicazione di rete (serializzazione, gestione della connessione, ecc.), disaccoppiando il client dalla locazione fisica degli oggetti remoti.
- **Ottimizzazione delle Prestazioni (Cache Proxy):** Un Cache Proxy può memorizzare temporaneamente i risultati per ridurre il numero di chiamate costose, specialmente se usato in congiunzione con un Remote Proxy.
- **Ottimizzazione della Memoria (Copy-on-Write):** Abilita l'ottimizzazione *copy-on-write*. Questa tecnica rimanda la costosa operazione di copia fisica di un oggetto al solo momento in cui questo deve essere modificato, evitando la copia se l'oggetto viene solo letto.

### 1.4.2 Svantaggi e Considerazioni (Contro)

- **Latenza Aggiuntiva:** L'indirezione stessa non è gratuita. Introduce un piccolo (spesso trascurabile) overhead sulle prestazioni. Ogni chiamata deve passare attraverso un livello logico aggiuntivo (il proxy) prima di raggiungere, eventualmente, l'oggetto reale.

- **Aumento della Complessità del Design:** L'uso del pattern introduce nuove classi e un livello di astrazione in più (l'interfaccia `Subject`, la/le classi `Proxy`). Questo può rendere il design più complesso da capire e manutenere rispetto a una chiamata diretta.
- **Proliferazione delle Classi:** In un sistema complesso, si può correre il rischio di dover implementare una classe `Proxy` corrispondente per *ciascuna* classe `RealSubject` che si vuole gestire. Questo può portare a un aumento significativo del numero di classi nel progetto.
- **Mitigazione alla Proliferazione:** Le slide stesse, tuttavia, suggeriscono una mitigazione a questo svantaggio. L'utilizzo di approcci più avanzati, come la programmazione ad aspetti (Aspect-Oriented Programming), può ridurre il numero di classi `Proxy` esplicite necessarie per implementare funzionalità trasversali (come la sicurezza o il logging).

# Chapter 2

## II Design Pattern Reference Monitor

### 2.1 Intento e Problema

In un sistema che gestisce dati o risorse, è fondamentale imporre restrizioni di accesso. L'intento del pattern **Reference Monitor** è proprio quello di definire un'astrazione che intercetta tutte le richieste e ne controlla la conformità con le autorizzazioni definite.

Il problema che risolve è semplice ma critico: se le regole di autorizzazione definite non vengono imposte ad ogni accesso, è come non averle affatto. Senza un controllo, un utente (o **Subject**) potrebbe effettuare qualsiasi azione non consentita, come leggere un file protetto.

### 2.2 Soluzione e Struttura

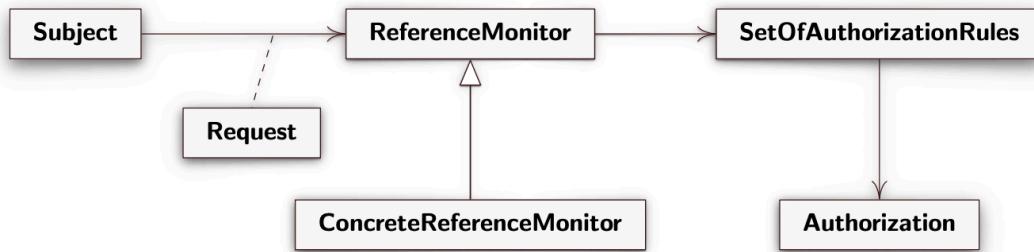
La soluzione consiste nel definire una componente, il **ReferenceMonitor**, che agisce come un "Policy Enforcement Point". Questo componente intercetta e valuta *tutte* le richieste, controlla la loro conformità con le autorizzazioni, prende una decisione e inoltra solo le richieste conformi.

#### 2.2.1 Componenti

La struttura del pattern include i seguenti elementi:

- **Subject**: L'entità (es. un utente o un processo) che richiede di accedere al **ProtectionObject**.
- **ProtectionObject**: L'oggetto o la risorsa che deve essere protetta.
- **Request**: Un oggetto che incapsula i dettagli della richiesta da parte del **Subject**.
- **ReferenceMonitor**: Il componente centrale che intercetta la richiesta e cerca tra le regole quella appropriata.
- **ConcreteReferenceMonitor**: Un'implementazione specifica del monitor, ad esempio per un certo tipo di risorsa come i file.
- **SetOfAuthorizationRules**: Un insieme che organizza e contiene tutte le regole di autorizzazione.

- **Authorization:** La singola regola di autorizzazione.



**Figure 2.1:** Diagramma UML delle classi del pattern Reference Monitor.

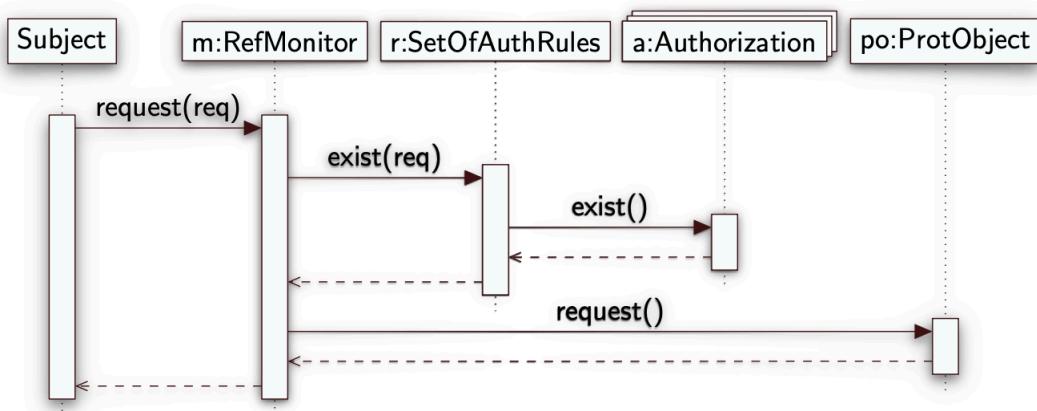
## 2.2.2 Focus: La classe Request

Nel diagramma UML, la classe **Request** è modellata come una *association class*. Questa è una scelta di design deliberata. La richiesta non è un attributo né del **Subject** né del **ReferenceMonitor**, ma piuttosto **caratterizza l'associazione** tra di loro. È un oggetto a sé stante che lega i due componenti, qualificando la natura dell'interazione (es. "cosa si sta chiedendo").

## 2.3 Flusso di Esecuzione

Il diagramma di sequenza mostra la dinamica di un controllo di autorizzazione:

1. Il **Subject** invia una **request(req)** al **m:RefMonitor**.
2. Il **RefMonitor** non agisce immediatamente. Interroga il **r:SetOfAuthRules** per verificare se esiste (**exist(req)**) un'autorizzazione valida per quella specifica richiesta.
3. Il **SetOfAuthRules** può a sua volta controllare le singole **a:Authorization**.
4. Solo se l'autorizzazione esiste e la richiesta è conforme, il **RefMonitor** inoltra la **request()** al **po:ProtObject** (l'oggetto protetto), che esegue l'azione.



**Figure 2.2:** Diagramma di sequenza che mostra la validazione di una richiesta.

## 2.4 Conseguenze: Pro e Contro

### 2.4.1 Benefici (Pro)

- **Imposizione Centralizzata:** Se *tutte* le richieste sono intercettate dal monitor, si possono imporre in modo centralizzato e affidabile tutte le regole di accesso definite, garantendo la coerenza della sicurezza.

### 2.4.2 Svantaggi e Considerazioni (Contro)

- **Degrado delle Prestazioni:** La conseguenza negativa più ovvia è che controllare *ogni singola* richiesta può far degradare le prestazioni del sistema.
- **Mitigazione 1 (Controllo a Monte):** Una tecnica per mitigare il calo di prestazioni è "tirar fuori" i controlli dal flusso di esecuzione principale. Ad esempio, si può inserire un solo controllo di sicurezza a monte (es. al momento dell'apertura di un file) e non ripeterlo per ogni singola operazione successiva (es. ogni `read` o `write` sul file).
- **Mitigazione 2 (Caching):** Un'altra ottimizzazione consiste nel tenere in memoria (in una cache) le decisioni prese precedentemente, associandole alla richiesta e al **Protection Object**. Questo evita di dover rivalutare le regole per richieste identiche e ripetute.

## 2.5 Implementazione e Relazione con il Proxy

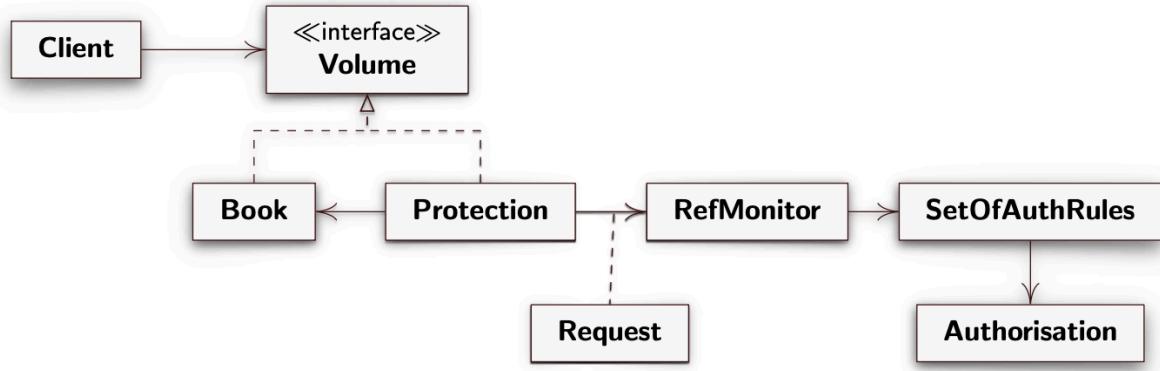
Una considerazione implementativa cruciale rimane: come si può **assicurare** che nessuna chiamata arrivi a un **ProtectionObject** senza prima passare dal **ReferenceMonitor**?

La risposta fornita dalle slide è chiara: **implementare un Protection Proxy** per le risorse da proteggere. Il **ReferenceMonitor** è un concetto astratto (la logica di decisione), mentre il **ProtectionProxy** è il meccanismo concreto di intercettazione.

L'esempio finale, che unisce i pattern, mappa i ruoli come segue:

- **Client** è il Subject.
- **Book** è il ProtectionObject.
- **Protection** è il Proxy.
- **RefMonitor** è il ReferenceMonitor.

In questa architettura, la classe **Protection** agisce da Proxy: intercetta la chiamata del **Client** e, prima di inoltrarla all'oggetto reale **Book**, *delega* la decisione di sicurezza chiamando il **RefMonitor** al suo interno.



**Figure 2.3:** Diagramma UML di un'implementazione del Reference Monitor tramite un Protection Proxy.

# Chapter 3

## Role-Based Access Control (RBAC)

### 3.1 Intento e Problema

Il pattern **Role-Based Access Control (RBAC)** è un modello di sicurezza di alto livello. Il suo intento è descrivere come gli utenti possono acquisire diritti di accesso non in base alla loro identità individuale, ma in base alle **funzioni del loro lavoro** o ai compiti che gli sono stati assegnati.

È un pattern specificamente pensato per ambienti complessi, caratterizzati da un gran numero di utenti, una grande varietà di risorse e molti tipi diversi di informazioni.

Il problema che RBAC risolve è la **complessità amministrativa**. In un sistema di grandi dimensioni, assegnare permessi a ogni singolo utente richiederebbe di immagazzinare e gestire un numero esorbitante di regole di autorizzazione individuali. Per gli amministratori, tenere traccia di chi può fare cosa diventerebbe rapidamente ingestibile e fonte di errori.

### 3.2 Soluzione: L'Astrazione del Ruolo

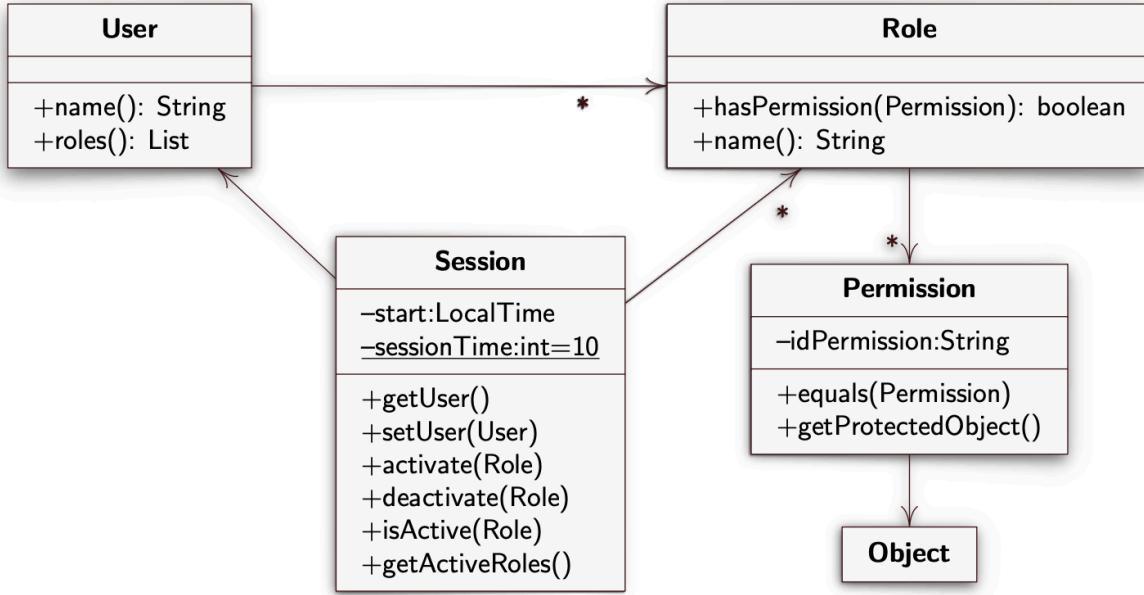
La soluzione di RBAC è tanto semplice quanto potente: introduce un livello di astrazione fondamentale, il **Ruolo (Role)**. L'assegnazione dei permessi non è più diretta tra utente e risorsa, ma è mediata dal ruolo, seguendo questo flusso:

1. I permessi (es. "scrivere file") sono assegnati ai ruoli (es. "Editore").
2. Gli utenti (es. "Alice") sono assegnati ai ruoli (es. "Editore").
3. Di conseguenza, gli utenti acquisiscono i permessi *attraverso* i ruoli a cui appartengono.

Questo approccio permette di mappare direttamente le politiche dell'organizzazione ("I manager approvano le ferie") in regole del sistema ("Il ruolo 'Manager' ha il permesso 'ApproveLeave'"). Inoltre, supporta nativamente il principio del *need-to-know* (minimo privilegio).

### 3.3 Struttura e Componenti

Il diagramma UML delle classi mostra un modello disaccoppiato e flessibile per implementare questa logica.



**Figure 3.1:** Diagramma UML delle classi del pattern RBAC.

#### 3.3.1 Componenti Chiave

I componenti principali del modello RBAC sono:

- **User**: Rappresenta un utente registrato nel sistema.
- **Role**: Descrive un ruolo predefinito, che corrisponde a una funzione o un compito lavorativo (es. "Security Officer", "Participant").
- **Permission**: Definisce il tipo di accesso permesso. È una classe (e non una semplice stringa) per incapsulare la logica di confronto (tramite `equals()`) e per mantenere un riferimento all'oggetto che protegge (`getProtectedObject()`).
- **Protection Object**: L'oggetto o la risorsa da proteggere (rappresentato genericamente come `Object` nel diagramma).
- **Session**: Un componente cruciale che mappa un utente a un insieme di ruoli *attivi* che gli sono stati assegnati.

#### 3.3.2 Focus: Il Ruolo della Classe Session

La classe **Session** è fondamentale per implementare il principio del minimo privilegio. Un utente può avere molti ruoli (es. "Manager", "Sviluppatore", "Revisore"), ma all'interno di una specifica sessione di lavoro ne "attiva" solo quelli che gli servono in quel momento. Il controllo dei permessi viene fatto **solo sui ruoli attivi** in quella sessione, non su tutti i ruoli che l'utente possiede.

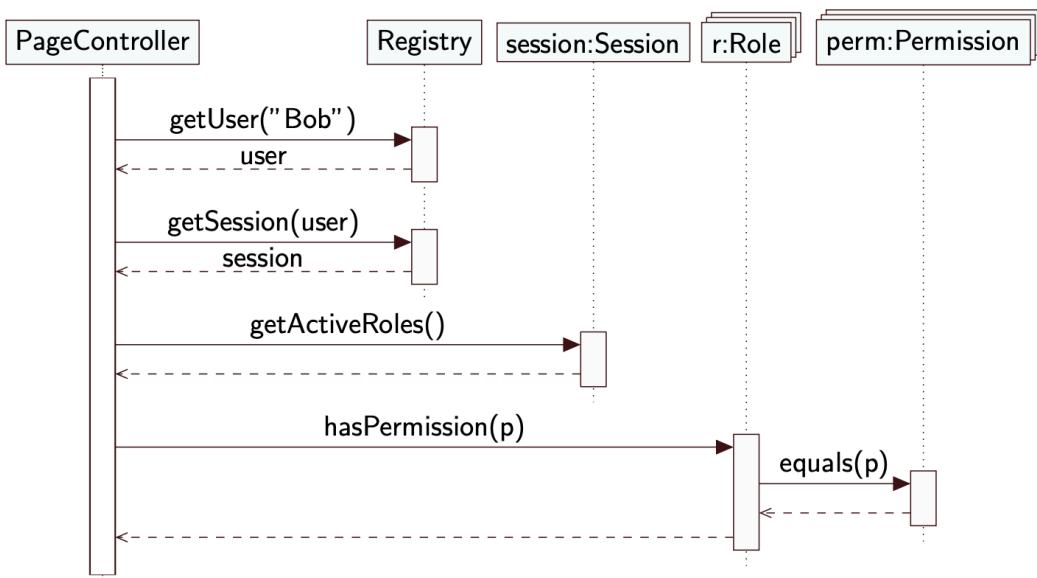
## 3.4 Flusso di Esecuzione

I diagrammi di sequenza mostrano la dinamica del pattern in due momenti chiave: il controllo dei permessi e la configurazione iniziale.

### 3.4.1 Controllo dei Permessi

Questo flusso mostra cosa accade quando un utente (es. "Bob") tenta di eseguire un'azione:

1. Un **PageController** (o un gestore di richieste) riceve la richiesta.
2. Ottiene l'oggetto **user** e la sua **session** corrente da un **Registry** centrale.
3. Chiede alla **session** quali sono i ruoli attivi (`getActiveRoles()`).
4. Per ogni **r:Role** attivo, il sistema chiede al ruolo: `hasPermission(p)?`, dove '**p**' è il permesso richiesto.
5. Il **Role**, a sua volta, controlla se uno dei suoi **perm:Permission** corrisponde a quello richiesto (tramite `equals(p)`).
6. Se anche solo **un** ruolo attivo possiede il permesso, l'azione è autorizzata.

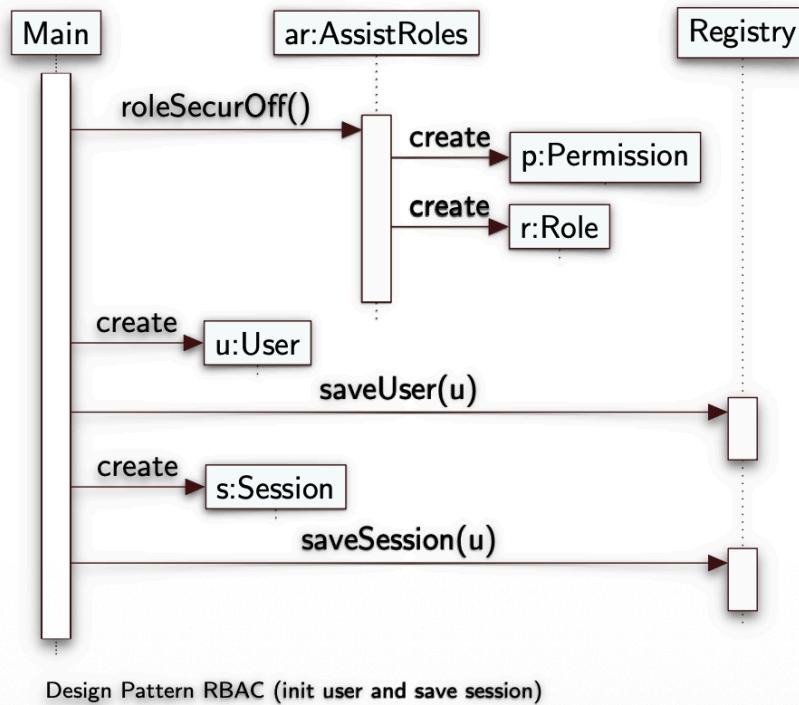


**Figure 3.2:** Diagramma di sequenza che mostra il controllo di un permesso.

### 3.4.2 Creazione e Configurazione

Questo flusso mostra come il sistema viene inizializzato:

1. Un processo principale (**Main**) o un amministratore avvia la configurazione.
2. Utilizza una classe helper (**ar:AssistRoles**) per creare le istanze di **p:Permission** e **r:Role** e associarle.
3. Crea le istanze di **u:User** e **s:Session**.
4. Salva gli utenti e le sessioni nel **Registry** per poterli recuperare in seguito.



**Figure 3.3:** Diagramma di sequenza della creazione e configurazione iniziale.

## 3.5 Conseguenze: Pro e Contro

### 3.5.1 Benefici (Pro)

- **Riduzione della Complessità Amministrativa:** È il vantaggio principale. Gli amministratori gestiscono la sicurezza a un livello più astratto (i ruoli), che sono in numero molto minore rispetto agli utenti.
- **Mappatura Organizzativa:** Le politiche di sicurezza dell'organizzazione ("I manager approvano...") possono essere mappate direttamente sulla definizione dei ruoli, rendendo il sistema più leggibile e coerente con la realtà aziendale.
- **Gestione Semplice del Personale:** La gestione del ciclo di vita dei dipendenti è drasticamente semplificata. Un nuovo assunto riceve i ruoli appropriati; un cambio di mansione richiede solo un cambio di ruoli; un utente che lascia l'organizzazione viene semplicemente disattivato o rimosso dalle sue associazioni di ruolo.

### 3.5.2 Svantaggi e Considerazioni (Contro)

- **Complessità Iniziale:** L'implementazione di RBAC non è banale. Richiede un'attenta analisi organizzativa per definire correttamente i ruoli e i permessi, e la configurazione iniziale (come visto nel diagramma di setup) può essere laboriosa.
- **Rischio di "Esplosione dei Ruoli":** Se non gestito correttamente, il sistema può soffrire di "Role Explosion", una situazione in cui vengono creati così tanti ruoli specifici da ritornare alla complessità iniziale (un ruolo per ogni utente).

# Chapter 4

## Autenticazione Basata su Token

### 4.1 Contesto e Problema

In un sistema distribuito, le richieste inviate ai vari servizi (lato server) devono essere scrupolosamente vagilate. È un requisito fondamentale che solo gli utenti autenticati e autorizzati possano accedere ed utilizzare i servizi.

Questo introduce due sfide principali che un'architettura di sicurezza deve risolvere:

1. **Autorizzazione delle Singole Richieste:** È necessario un meccanismo per autorizzare ogni singola richiesta proveniente da un utente, ma allo stesso tempo bisogna evitare di chiedere continuamente le credenziali (username e password) all'utente, cosa che comprometterebbe l'usabilità del sistema.
2. **Centralizzazione dell'Identità:** Per motivi di sicurezza, manutenibilità e coerenza, è necessario far sì che la prova di identità e la verifica delle credenziali siano gestite da un **singolo server** dedicato (un "authority"), invece di replicare questa logica in ogni singolo servizio.

### 4.2 Soluzione: Il Flusso di Autenticazione

La soluzione a questo problema è un'architettura di **autenticazione basata su token**. Il principio è semplice: una volta che l'utente si è autenticato una prima volta, gli viene fornito un "gettone" (il token) che può usare per tutte le sue successive richieste ai servizi, come prova della sua identità e delle sue autorizzazioni.

#### 4.2.1 Componenti del Flusso

Il flusso mostrato nelle slide coinvolge quattro componenti principali:

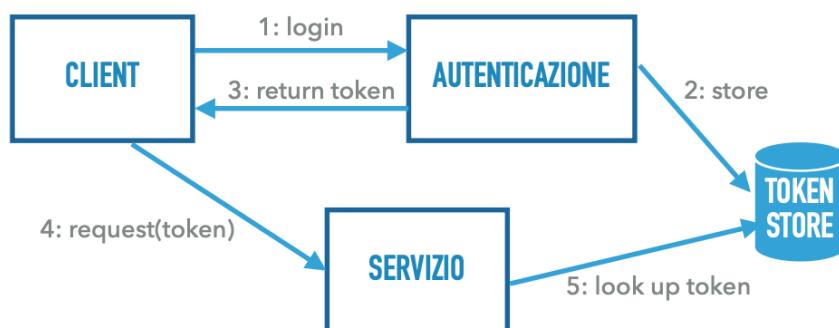
- **CLIENT:** L'applicazione front-end usata dall'utente (es. un browser web o un'app mobile).
- **AUTENTICAZIONE:** Un server dedicato, l'unico responsabile della verifica delle credenziali dell'utente (es. username e password).

- **SERVIZIO:** Uno o più server che espongono la logica di business e le risorse protette (es. un'API per i dati utente).
- **TOKEN STORE:** Un database centralizzato che memorizza i token generati, per permettere ai servizi di verificarne la validità.

### 4.2.2 Flusso di Esecuzione

Il processo si svolge nei seguenti passi:

1. **Login:** Il **CLIENT** invia le credenziali dell'utente (es. username e password) al server di **AUTENTICAZIONE**.
2. **Store:** Il server di **AUTENTICAZIONE** verifica le credenziali. Se sono valide, genera un token unico e lo memorizza nel **TOKEN STORE**.
3. **Return Token:** Il server di **AUTENTICAZIONE** restituisce il token al **CLIENT**. Il client memorizzerà questo token per le richieste future.
4. **Request with Token:** Quando il **CLIENT** ha bisogno di accedere a una risorsa protetta, invia la richiesta al **SERVIZIO** allegando il token (spesso in un header HTTP, come `Authorization`).
5. **Token Lookup:** Il **SERVIZIO** riceve la richiesta e, prima di eseguirla, estrae il token. Contatta il **TOKEN STORE** per verificare se il token esiste ed è valido (`look up token`). Se il token è valido, il **SERVIZIO** esegue la richiesta.



**Figure 4.1:** Flusso di un sistema di autenticazione basato su token con validazione tramite TOKEN STORE.

## 4.3 Analisi Dettagliata del Token

### 4.3.1 Definizione e Formati

Un token di accesso è, a tutti gli effetti, una **credenziale** che può essere usata per accedere a servizi. Il suo scopo primario è informare il servizio che chiunque possieda quel token è stato (precedentemente) autorizzato ad accedere.

Le slide identificano due formati principali per un token:

- **Token "Opaque" (Opaco)**: Una stringa apparentemente casuale e incomprensibile (es. kPoPMRYrCEoY06s5). Questo tipo di token non contiene informazioni leggibili e *richiede* che il servizio esegua un *look up* (come nel passo 5) per recuperare le informazioni ad esso associate (es. a quale utente appartiene).
- **Token Strutturato (es. JWT)**: Un oggetto con una struttura definita, come un **JSON Web Token (JWT)**. Questi token sono "self-contained" e possono contenere le informazioni (dette "claims") direttamente al loro interno.

### 4.3.2 Contenuto Tipico (Claims)

Indipendentemente dal formato, un token (o i dati ad esso associati nel Token Store) tipicamente indica:

- **L'utente** identificato (chi è l'utente).
- **L'emittente** (chi ha emesso il token, es. il server di autenticazione).
- Il **timestamp** al momento dell'emissione.
- La **durata** (una data di scadenza, dopo la quale il token non è più valido).
- L'**audience** (a chi è rivolto, ovvero per quali servizi è valido).

## 4.4 Conseguenze e Considerazioni

### 4.4.1 Benefici (Pro)

L'introduzione di un'architettura a token risolve direttamente i problemi esposti:

- **Centralizzazione dell'Autenticazione**: La logica di verifica delle credenziali è gestita da un solo server, come richiesto. I singoli servizi non devono più conoscere le password degli utenti; devono solo sapere come validare un token.
- **Migliore User Experience**: Si evita di chiedere continuamente le credenziali all'utente. Il login avviene una sola volta e il token viene riutilizzato per un determinato periodo di tempo (la durata del token).
- **Sicurezza per Richiesta**: Il token permette di autorizzare (o negare) le singole richieste ai servizi in modo granulare.

### 4.4.2 Svantaggi e Considerazioni (Contro)

- **Dipendenza dallo Store (per Token Opachi)**: Il flusso presentato nelle slide, che usa un *look up token*, crea una forte dipendenza tra ogni SERVIZIO e il TOKEN STORE. Questo store diventa un componente critico: se è lento o non disponibile, l'intero sistema si blocca.
- **Gestione della Scadenza**: I token devono avere una scadenza per motivi di sicurezza. Questo introduce la complessità di dover gestire il "refresh" del token (ottenere uno nuovo quando quello vecchio scade) in modo trasparente per l'utente.

# Chapter 5

## Autenticazione e Protezione dei Servizi

Questo capitolo analizza il pattern **Authenticator**, un componente fondamentale per la sicurezza dei sistemi, e le tecniche pratiche ad esso associate per la gestione delle password e la protezione da attacchi comuni.

### 5.1 Il Design Pattern Authenticator

#### 5.1.1 Intento e Problema

L'intento del pattern **Authenticator** è quello di **verificare che il soggetto** che intende accedere al sistema sia effettivamente chi dice di essere.

Il **problema** che risolve è come prevenire l'accesso da parte di impostori in sistemi che contengono risorse di valore (dati medicali, informazioni di business, ecc.). La soluzione a questo problema deve bilanciare diverse "forze":

- **Flessibilità:** Il sistema deve poter gestire una varietà di utenti e risorse con diversi livelli di riservatezza.
- **Dependability:** L'autenticazione deve essere affidabile e sicura, usando un protocollo robusto.
- **Costo:** Esiste un compromesso tra il livello di sicurezza e il costo di implementazione.
- **Prestazioni:** Se l'autenticazione è frequente, deve essere efficiente.
- **Frequenza:** Bisogna evitare che i soggetti si debbano autenticare troppo frequentemente, per non compromettere l'usabilità.

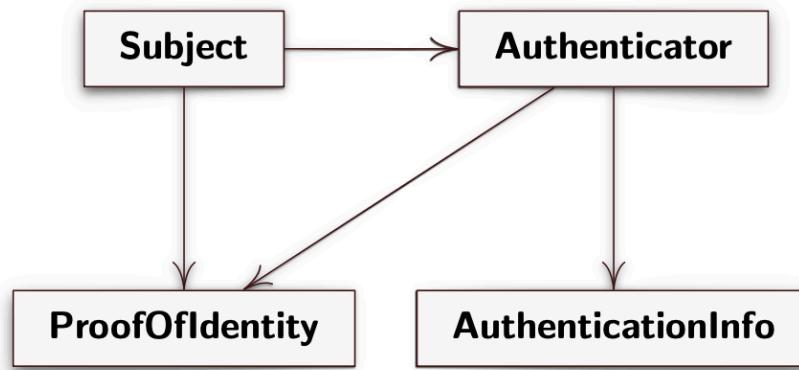
#### 5.1.2 Soluzione e Struttura

La soluzione proposta è l'uso di un **Singolo Punto di Accesso** (*Single Point of Access*). Questo componente riceve tutte le interazioni iniziali di un soggetto con il sistema e applica un protocollo per verificare l'identità del soggetto.

La struttura di questa soluzione è composta da quattro elementi:

- **Subject:** Il soggetto (es. utente) che richiede l'accesso al sistema.

- **Authenticator:** Il singolo punto di accesso che riceve la richiesta e applica il protocollo di verifica.
- **AuthenticationInfo:** L'informazione nota usata per la verifica (es. una password hashata, una chiave biometria).
- **ProofOfIdentity:** Un "gettone" creato dall'Authenticator se l'autenticazione ha successo. Viene assegnato al Subject per indicare che è un utente legittimo.

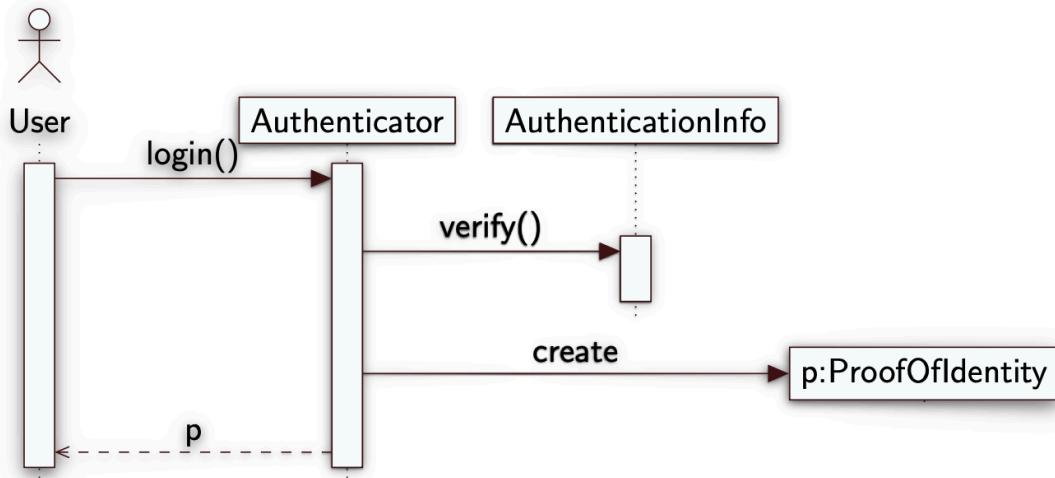


**Figure 5.1:** Diagramma UML delle classi del pattern Authenticator.

### 5.1.3 Flusso di Esecuzione

Il diagramma di sequenza mostra la dinamica di autenticazione:

1. L'User (un Subject) invia una richiesta di `login()` all'Authenticator.
2. L'Authenticator esegue il metodo `verify()`, utilizzando l'AuthenticationInfo (es. l'hash della password salvato nel database) per convalidare le credenziali fornite.
3. Se la verifica ha successo, l'Authenticator crea (`create`) una nuova istanza di `p:ProofOfIdentity`.
4. Questa "prova di identità" (`p`) viene restituita all'User, che la userà per le richieste successive.



**Figure 5.2:** Diagramma di sequenza del processo di autenticazione.

### 5.1.4 Conseguenze e Varianti

L'applicazione di questo pattern porta a diverse conseguenze:

- **Flessibilità (Pro):** Variando il protocollo (la logica in `Authenticator`) e l'informazione (`AuthenticationInfo`), si possono gestire molti tipi di autenticazione: ciò che l'utente *conosce* (password), ciò che *possiede* (ID card), o ciò che è (biometria).
- **Dependability (Pro):** L'informazione di autenticazione è separata e può essere memorizzata in un'area protetta (es. con accesso in sola lettura).
- **Performance e Frequenza (Pro):** Al momento della verifica, si produce una *proof* (come un **token**). Questa "prova" è ciò che abbiamo analizzato nel capitolo precedente e serve proprio a evitare di dover ripetere l'autenticazione per ogni richiesta.
- **Single Sign On (Variante):** Questo pattern è alla base del *Single Sign On (SSO)*, un processo in cui la verifica dell'identità (il `ProofOfIdentity` o "credenziale") può essere usata su vari domini e servizi per un certo intervallo di tempo.

## 5.2 Gestione Sicura delle Password: Hashing

Un'implementazione comune di `AuthenticationInfo` è l'hash della password dell'utente. È fondamentale che la password non venga *mai* conservata in chiaro.

### 5.2.1 Proprietà di una Funzione di Hashing

La funzione di hashing usata per le password deve avere proprietà specifiche:

- **Deterministica:** Lo stesso input deve produrre sempre lo stesso output.
- **Irreversibile:** Deve essere computazionalmente impossibile risalire all'input (la password) a partire dall'output (l'hash).

- **Lunghezza Fissa:** L'output deve avere una lunghezza fissa, indipendentemente dalla lunghezza dell'input.
- **Effetto Valanga:** Un cambiamento minimo nell'input (es. un singolo carattere) deve produrre un hash completamente diverso.

### 5.2.2 La Necessità di Funzioni "Lente"

Per la sicurezza delle password, la funzione di hashing deve essere **intenzionalmente lenta**. Questo serve a rendere impraticabili gli attacchi *brute-force* (o a dizionario). Se un attaccante ottiene il database degli hash, proverà ad "indovinare" la password applicando la funzione di hashing a miliardi di password comuni (prese da un dizionario) fino a trovare una corrispondenza.

- Un tempo di 100 ms per generare un hash è accettabile per il singolo login di un utente.
- Per un attaccante con un dizionario di  $10^9$  password, questo tempo diventa proibitivo:  $100 \times 10^{-3}s \times 10^9 = 10^8$  secondi, che corrispondono a circa **3 anni e 2 mesi** per un singolo hash.

### 5.2.3 Hashing con Salt

La lentezza da sola non basta. Se due utenti usano la stessa password ("password123"), avranno lo stesso hash. Un attaccante può pre-calcolare gli hash di miliardi di password e salvarli in tabelle ottimizzate, chiamate *rainbow tables*, che permettono di trovare la password corrispondente a un hash in pochi secondi.

La soluzione è il **Salt**:

- Il "salt" è un valore random unico, generato per ogni utente al momento della registrazione.
- Questo valore viene aggiunto alla password *prima* di calcolare l'hash.
- In questo modo, anche se due utenti hanno la stessa password, i loro hash saranno completamente diversi (perché i loro salt sono diversi).
- Il salt viene memorizzato insieme all'hash e utilizzato durante la verifica per poter ricalcolare l'hash corretto.

## 5.3 Implementazione Pratica: Scrypt e Guava

### 5.3.1 La Libreria Scrypt per l'Hashing

**Scrypt** è una funzione di hashing progettata per essere "lenta" in modo efficace, poiché richiede deliberatamente molte risorse di CPU e, soprattutto, di memoria. I suoi parametri principali sono:

- **N:** Indica il costo computazionale (CPU e memoria).
- **r:** Indica la dimensione del blocco di memoria.

- **p**: Indica il fattore di parallelizzazione.

Nelle librerie moderne come SCryptUtil, la gestione del salt è automatica:

- **Creazione Hash**: final String hash = SCryptUtil.scrypt(passwd, 32768, 8, 1);
- **Verifica Hash**: SCryptUtil.check(passwd, authenticationInfo);

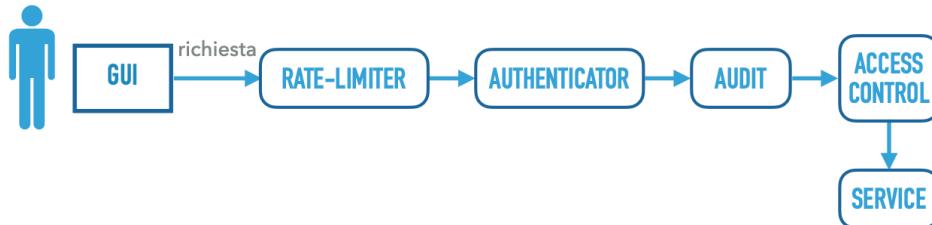
Il metodo `check` estrae automaticamente il salt dall'hash `authenticationInfo` conservato, per poter eseguire il confronto corretto.

### 5.3.2 Protezione da Attacchi DoS con Rate-Limiting

Un attacco **Denial of Service (DoS)** mira a rendere un servizio inutilizzabile per gli utenti legittimi, spesso generando un traffico enorme che sovraccarica il server. Un attacco **DDoS (Distributed DoS)** usa molte macchine per lo stesso scopo.

Poiché anche l'autenticazione (specialmente con funzioni lente come Scrypt) consuma risorse, è fondamentale proteggerla. La soluzione è il **Rate-Limiting**:

- Si limita la frequenza delle richieste che possono essere processate.
- Il Rate-Limiter deve essere il **primo componente** a processare una richiesta, *prima* dell'autenticazione, per bloccare il traffico eccessivo prima che consumi risorse.



**Figure 5.3:** Flusso di una richiesta con Rate-Limiting anteposto all'Authenticator.

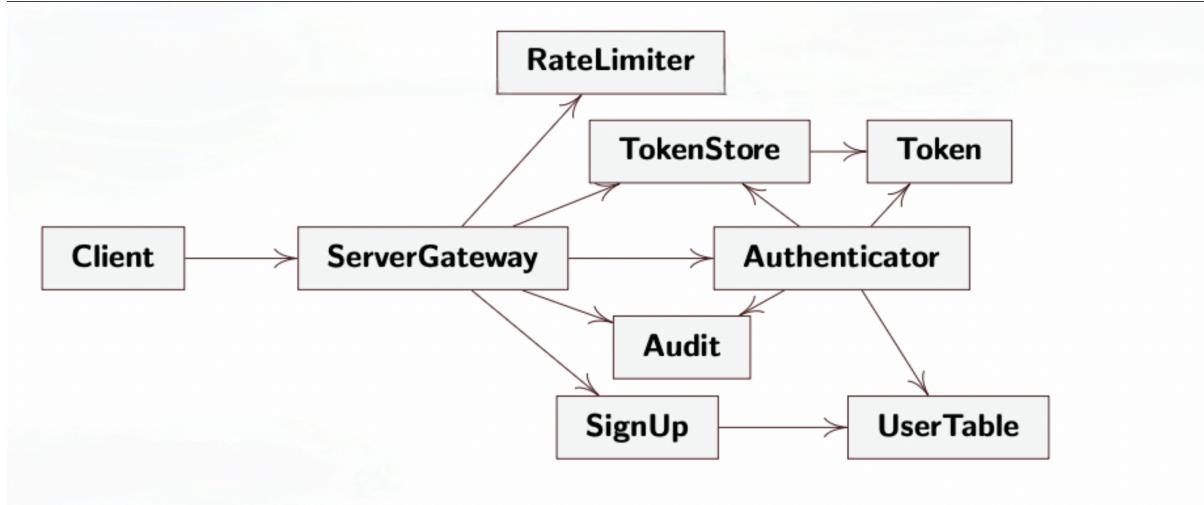
Questa strategia fa parte della "Defence in Depth" (difesa a più strati): il rate-limiter viene applicato al gateway (punto di ingresso) e può essere ri-applicato su ogni singolo server per ulteriore sicurezza.

La libreria Guava di Google fornisce una classe `RateLimiter` per questo scopo:

- **Creazione**: `RateLimiter r = RateLimiter.create(2);` (permette 2 richieste al secondo).
- **Acquisizione Bloccante**: `r.acquire();` (mette in attesa la richiesta se il limite è superato).
- **Acquisizione non Bloccante**: `r.tryAcquire(10, TimeUnit.MICROSECONDS);` (tenta di acquisire, ma rinuncia dopo un timeout).

## 5.4 Progettazione di un Flusso Completo

Combinando tutti questi concetti, si ottiene un'architettura di autenticazione robusta. Il `ServerGateway` agisce come punto di ingresso unico.

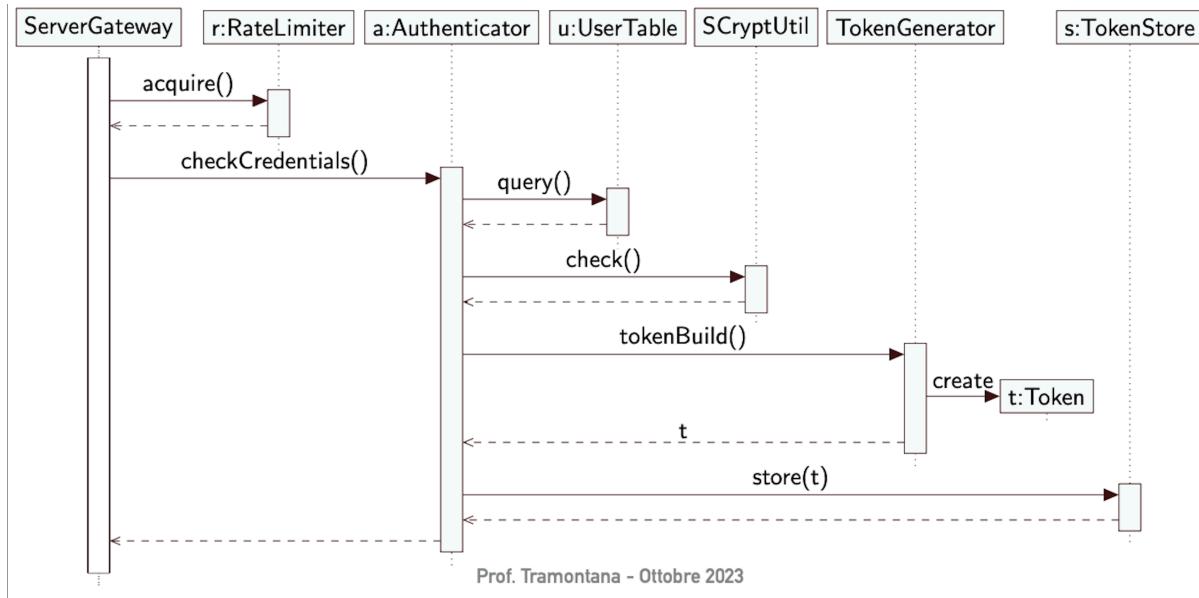


**Figure 5.4:** Diagramma delle classi di un'architettura di autenticazione completa.

### 5.4.1 Flusso di Log In

Il diagramma di sequenza per il login mostra l'orchestrazione di tutti i componenti:

1. Il `ServerGateway` riceve la richiesta.
2. Chiama `acquire()` sul `r:RateLimiter`.
3. Inoltra le credenziali all'`a:Authenticator`.
4. L'`Authenticator` fa una `query()` su `u:UserTable` per trovare l'hash dell'utente.
5. Usa `SCryptUtil.check()` per verificare la password.
6. Se la password è valida, usa `TokenGenerator` per creare un token.
7. Infine, salva (`store(t)`) il nuovo token nel `s:TokenStore`.

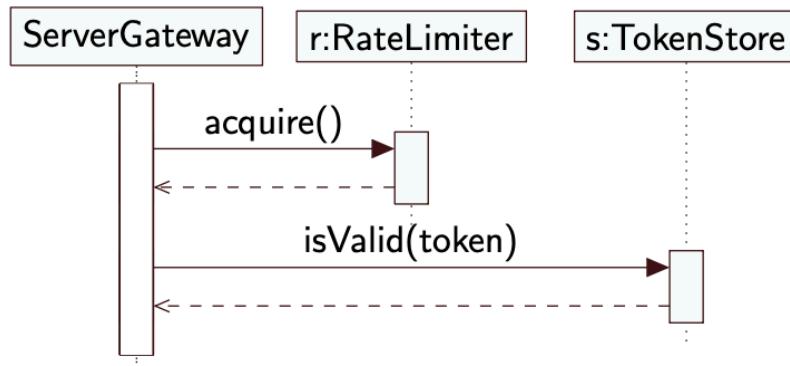


**Figure 5.5:** Diagramma di sequenza del processo di log in.

#### 5.4.2 Flusso per Richieste Successive

Per tutte le richieste successive al login, il flusso è più snello (come visto nel Capitolo 4):

1. Il **ServerGateway** riceve la richiesta (che contiene il token).
2. Chiama **acquire()** sul **r:RateLimiter**.
3. Inoltra il token al **s:TokenStore** per la validazione (**isValid(token)**).
4. Se il token è valido, la richiesta viene inoltrata al servizio di business.



**Figure 5.6:** Diagramma di sequenza per richieste successive al log in.

### 5.5 Conseguenze Generali: Pro e Contro

L'adozione del pattern **Authenticator** e delle tecniche di hashing e rate-limiting associate ha impatti significativi sul design del sistema, introducendo benefici in termini di sicurezza e flessibilità, ma anche nuove considerazioni implementative.

### 5.5.1 Benefici (Pro)

- **Flessibilità (Authenticator):** Il pattern disaccoppia la logica di autenticazione. Variando l'implementazione dell'Authenticator e il formato dell'AuthenticationInfo, il sistema può supportare in modo flessibile molteplici metodi di autenticazione (password, biometria, smart card) senza impattare il resto dell'applicazione.
- **Sicurezza (Authenticator):** Poiché l'AuthenticationInfo è un componente separato, può essere isolato e memorizzato in un'area protetta (es. un database con accesso ristretto), migliorando la sicurezza del dato sensibile.
- **Efficienza (Authenticator):** Come conseguenza diretta, il pattern abilita l'uso di token. Al momento della verifica dell'identità, l'Authenticator produce una ProofOfIdentity (un token), che evita all'utente di doversi autenticare ad ogni singola richiesta, bilanciando sicurezza e usabilità.
- **Resistenza a Attacchi Brute-Force (Hashing Lento):** L'uso di funzioni di hashing *intenzionalmente lente* (come Scrypt) rende gli attacchi a dizionario o brute-force computazionalmente impraticabili, anche se un attaccante dovesse rubare il database degli hash.
- **Resistenza a Rainbow Table (Salt):** L'uso di un "salt" unico per ogni utente vanifica l'efficacia delle *rainbow tables* (tabelle di hash pre-calcolati), poiché lo stesso hash non può essere riutilizzato per password identiche di utenti diversi.
- **Protezione da DoS (Rate-Limiting):** Il Rate-Limiter è la prima linea di difesa contro attacchi DoS, bloccando il traffico malevolo *prima* che possa consumare risorse costose (come la CPU per l'hashing o le connessioni al database).
- **Robustezza (Defence in Depth):** L'architettura stessa (Gateway -> Rate-Limiter -> Authenticator) è un esempio di "Difesa a più strati". Il fallimento di un singolo strato (es. un Rate-Limiter mal configurato) non compromette immediatamente l'intero sistema, poiché la difesa è distribuita.

### 5.5.2 Svantaggi e Considerazioni (Contro)

- **Costo di Performance (Hashing Lento):** La lentezza è un'arma a doppio taglio. Se da un lato ferma gli attaccanti, introduce comunque un ritardo (es. 100ms) per l'utente legittimo durante il login. Questo tempo è un compromesso tra sicurezza e reattività.
- **Complessità di Gestione (Salt):** L'uso del "salt" aggiunge un piccolo strato di complessità. Il salt, generato casualmente, deve essere memorizzato in modo sicuro insieme all'hash e recuperato correttamente durante la fase di verifica.
- **Impatto sull'Usabilità (Rate-Limiting):** Un Rate-Limiter configurato in modo troppo aggressivo può bloccare utenti legittimi. L'implementazione (es. `acquire()` di Guava) blocca l'utente e lo mette in attesa, peggiorando la sua esperienza. È necessario un bilanciamento e, spesso, un meccanismo di `tryAcquire` con timeout per respingere la richiesta piuttosto che bloccarla indefiniteamente.



**Università  
di Catania**

**UNIVERSITY OF CATANIA**

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

A MASTER OF SCIENCE DEGREE IN COMPUTER SCIENCE.

---

*Alfio Spoto*

Ingegneria dei sistemi distribuiti

---

APPUNTI LEZIONI ISD  
II

---

---

Academic Year 2024 - 2025

# Contents

<b>1</b>	<b>Remote Proxy e Comunicazione Distribuita</b>	<b>1</b>
1.1	Intento e Problema . . . . .	1
1.1.1	La Sfida della Trasparenza . . . . .	1
1.2	Soluzione: Il Remote Proxy . . . . .	2
1.3	Architettura Completa a Due Lati . . . . .	2
1.3.1	Diagrammi di Sequenza . . . . .	3
1.4	Disaccoppiamento Avanzato: Forwarder-Receiver . . . . .	3
1.5	Conseguenze: Pro e Contro . . . . .	4
1.5.1	Benefici (Pro) . . . . .	4
1.5.2	Svantaggi e Considerazioni (Contro) . . . . .	4
<b>2</b>	<b>Strategie di Distribuzione: Remote Facade e DTO</b>	<b>6</b>
2.1	Il Design Pattern Remote Facade . . . . .	6
2.1.1	Intento e Problema . . . . .	6
2.1.2	Soluzione . . . . .	6
2.1.3	Flusso di Esecuzione . . . . .	7
2.1.4	Altre Responsabilità del Facade . . . . .	8
2.2	Il Design Pattern Data Transfer Object (DTO) . . . . .	8
2.2.1	Intento e Problema . . . . .	8
2.2.2	Soluzione . . . . .	9
2.2.3	L'Assembler . . . . .	9
2.3	Architettura di Esempio (RMI) . . . . .	10
2.3.1	Implementazione (RMI) . . . . .	10
2.4	Conseguenze Generali: Pro e Contro . . . . .	11
2.4.1	Benefici (Pro) . . . . .	11
2.4.2	Svantaggi e Considerazioni (Contro) . . . . .	12
<b>3</b>	<b>Gestione dello Stato della Sessione</b>	<b>13</b>
3.1	Il Dilemma: Stateful vs. Stateless . . . . .	13
3.2	Pattern 1: Client Session State . . . . .	13
3.2.1	Intento e Soluzione . . . . .	13
3.2.2	Esempio di Implementazione . . . . .	14
3.2.3	Conseguenze: Pro e Contro . . . . .	15
3.3	Pattern 2: Server Session State . . . . .	15
3.3.1	Intento e Soluzione . . . . .	15
3.3.2	Clustering e Failover . . . . .	15
3.3.3	Esempio di Implementazione . . . . .	15

3.3.4	Conseguenze: Pro e Contro . . . . .	16
3.4	Pattern 3: Database Session State . . . . .	16
3.4.1	Intento e Soluzione . . . . .	16
3.4.2	Gestione dei Dati "Pending" . . . . .	17
3.4.3	Conseguenze: Pro e Contro . . . . .	17
3.5	Pattern Correlato: Serialized LOB . . . . .	17
3.5.1	Intento e Problema . . . . .	18
3.5.2	Soluzione e Conseguenze . . . . .	18
<b>4</b>	<b>Il Design Pattern Memento</b>	<b>19</b>
4.1	Intento e Problema . . . . .	19
4.2	Soluzione e Struttura . . . . .	19
4.2.1	Partecipanti . . . . .	20
4.3	Conseguenze: Pro e Contro . . . . .	20
4.3.1	Benefici (Pro) . . . . .	20
4.3.2	Svantaggi e Considerazioni (Contro) . . . . .	20
<b>5</b>	<b>Il Design Pattern Idempotent Receiver</b>	<b>22</b>
5.1	Intento e Problema . . . . .	22
5.2	Soluzione: L'Operazione Idempotente . . . . .	22
5.3	Flusso di Esecuzione . . . . .	23
5.4	Struttura di Esempio . . . . .	23
5.5	Gestione dello Storage (Cleanup) . . . . .	24
5.6	Conseguenze: Pro e Contro . . . . .	24
5.6.1	Benefici (Pro) . . . . .	24
5.6.2	Svantaggi e Considerazioni (Contro) . . . . .	24

# Chapter 1

## Remote Proxy e Comunicazione Distribuita

Nei capitoli precedenti abbiamo introdotto il pattern Proxy come un surrogato generico. Questo capitolo analizza in dettaglio una delle sue varianti più cruciali nei sistemi distribuiti: il **Remote Proxy**.

### 1.1 Intento e Problema

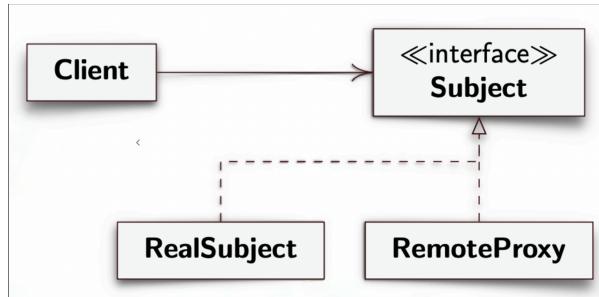
Il **contesto** è quello di un sistema distribuito in cui un **Client** necessita di accedere a servizi di un componente (il **RealSubject**) che risiede su un altro host.

Il **problema** è che l'accesso diretto da parte del client all'oggetto remoto non è appropriato per diverse ragioni:

- **Accoppiamento di Rete:** I client non dovrebbero conoscere i dettagli di basso livello come gli indirizzi di rete (IP, porta) o i protocolli di comunicazione inter-processo (IPC) per contattare il servizio.
- **Trasparenza Sintattica:** L'accesso al servizio remoto deve essere trasparente e semplice. Il **Client** non deve cambiare la sintassi usata per le chiamate; dovrebbe poter chiamare `oggetto.metodo()` come se l'oggetto fosse locale.

#### 1.1.1 La Sfida della Trasparenza

L'obiettivo della trasparenza ne introduce un altro: la **trasparenza delle prestazioni**. Una comunicazione da un host a un altro è ordinariamente di grandezza più lenta di una chiamata di metodo locale (le slide indicano 20.000 ns contro 1 ns). Se il **Remote Proxy** nasconde *tropppo bene* questa differenza (piena trasparenza), può oscurare le penalità nelle prestazioni. Questo è pericoloso, poiché può indurre il programmatore del **Client** a usare le chiamate remote in modo inefficiente (ad esempio, dentro un ciclo) pensando che siano "economiche" come quelle locali.



**Figure 1.1:** Diagramma UML di un Remote Proxy. LocalBook è il proxy per Book e usa SockCommunicator per la rete.

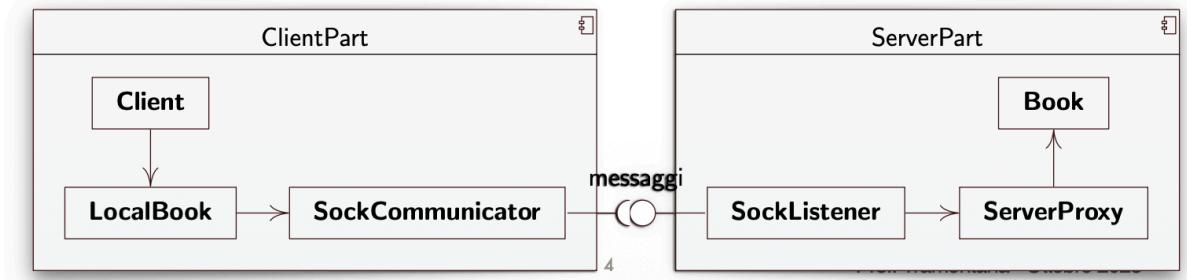
## 1.2 Soluzione: Il Remote Proxy

La soluzione è che il **Client** comunichi con un **Remote Proxy** locale (nell'esempio, **LocalBook**). Questo proxy ha la stessa interfaccia del **RealSubject** (es. **Volume**) e si occupa di:

1. Tenere traccia della posizione (indirizzo IP, porta) e dell'identità del **RealSubject** (es. **Book**).
2. Implementare la comunicazione tra le due macchine (es. usando socket).
3. **Tradurre** le chiamate a metodi (es. `getText()`) in messaggi di rete (marshalling) da inviare all'host remoto.
4. Ricevere le risposte dall'host remoto, tradurle nuovamente (unmarshalling) e restituirle al **Client**.

## 1.3 Architettura Completa a Due Lati

Per far funzionare questo meccanismo, non è sufficiente un proxy solo sul lato client. È necessaria un'architettura simmetrica con componenti su entrambi i lati della connessione.



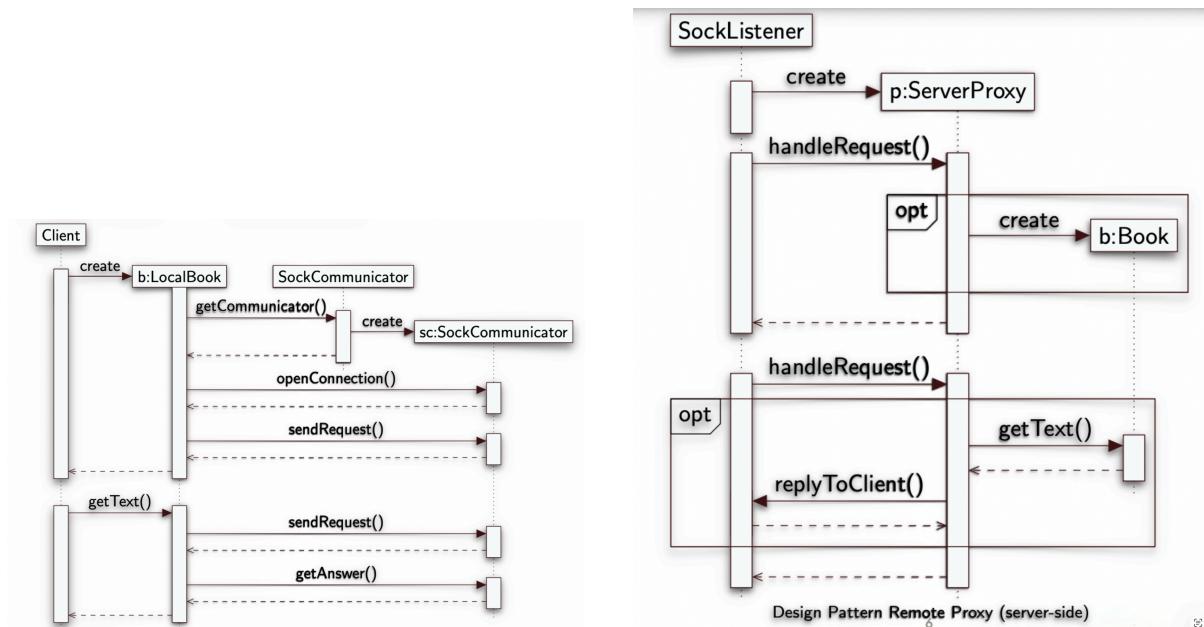
**Figure 1.2:** Diagramma delle componenti dell'architettura Remote Proxy.

- **Client Part:** È composta dal **Client** e dal **Remote Proxy** (es. **LocalBook**). Il **LocalBook** traduce le chiamate a metodi in messaggi, che vengono inviati tramite un **SockCommunicator**.
- **Server Part:** È composta dal **RealSubject** (es. **Book**) e da due componenti lato server:

- **SockListener:** Un componente che rimane costantemente in ascolto sulla rete per messaggi in arrivo.
- **Server Proxy:** Quando un messaggio arriva, il SockListener lo passa a un Server Proxy (a volte chiamato "skeleton"), che ha il compito opposto: traduce il messaggio di rete in una chiamata di metodo sull'oggetto Book reale.

### 1.3.1 Diagrammi di Sequenza

Il flusso di una chiamata `getText()` è diviso in due parti:



**Figure 1.3:** Flusso di una chiamata: lato client (sinistra) e lato server (destra).

- **Client-Side (Sinistra):** Il Client chiama `getText()` sul **b:LocalBook** (il proxy). Il proxy, a sua volta, chiama `sendRequest()` sul **sc:SockCommunicator** e attende una risposta con `getAnswer()`.
- **Server-Side (Destra):** Il **SockListener** riceve la richiesta e la affida al **p:ServerProxy** (`handleRequest()`). Il proxy chiama `getText()` sul **b:Book** reale, ottiene il risultato e lo invia indietro al client (`replyToClient()`).

## 1.4 Disaccoppiamento Avanzato: Forwarder-Receiver

L'architettura vista finora, sebbene efficace, crea una dipendenza tra i proxy (**LocalBook**, **ServerProxy**) e il meccanismo di comunicazione specifico (**SockCommunicator**, **SockListener**). Se volessimo cambiare la tecnologia di rete (es. da Sockets a code di messaggi AMQP), dovremmo modificare i proxy.

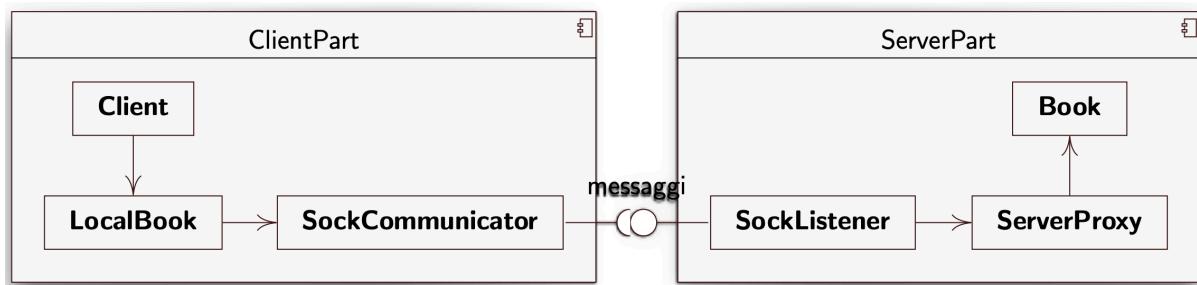
Per risolvere questo, si introduce il pattern **Forwarder-Receiver**.

- **Problema:** I meccanismi IPC (TCP/IP, Sockets) sono efficienti ma introducono dipendenze dal sistema operativo e dai protocolli di rete.

- **Soluzione:** Disaccoppiare i *peer* (le classi che comunicano) dai meccanismi di comunicazione sottostanti.

In questo modello, i ruoli sono mappati come segue:

- **Peer:** Sono le classi responsabili dei compiti dell'applicazione. Nel nostro caso, i Peer sono **LocalBook** e **ServerProxy**.
- **Forwarder:** Incapsula la logica di *invio* dei messaggi. Mappa i nomi logici dei peer ai loro indirizzi fisici. Nel nostro caso, il **SockCommunicator** è il Forwarder.
- **Receiver:** Incapsula la logica di *ricezione* dei messaggi. Converte lo stream di dati in un messaggio per il suo Peer. Nel nostro caso, il **SockListener** è il Receiver.



**Figure 1.4:** Mapping tra Remote Proxy e Forwarder-Receiver.

## 1.5 Conseguenze: Pro e Contro

### 1.5.1 Benefici (Pro)

- **Trasparenza Sintattica:** Il Client è disaccoppiato dalla natura remota del servizio. Utilizza la stessa sintassi di chiamata (`oggetto.metodo()`) sia per oggetti locali che remoti, rendendo il codice più semplice.
- **Disaccoppiamento dalla Locazione:** Il Client non conosce i dettagli di rete (indirizzo IP, porta) del RealSubject. Questa logica è incapsulata nel Remote Proxy.
- **Disaccoppiamento dal Protocollo (con F-R):** L'uso congiunto con il pattern Forwarder-Receiver porta il disaccoppiamento a un livello superiore. I Peer (i proxy) sono disaccoppiati dal meccanismo IPC sottostante. Si può cambiare la tecnologia di rete (es. da sockets a code di messaggi) modificando solo le classi Forwarder e Receiver.
- **Analisi delle Prestazioni:** Paradossalmente, sebbene il proxy introduca latenza, rende anche *più facile analizzare* i ritardi della comunicazione di rete, poiché la logica è centralizzata in un unico punto.

### 1.5.2 Svantaggi e Considerazioni (Contro)

- **Latenza di Rete:** È lo svantaggio più ovvio e inevitabile. Una chiamata di metodo remoto è ordinariamente più lenta di una chiamata locale.

- **Rischio della Trasparenza Eccessiva:** Come menzionato, se il proxy nasconde troppo bene la latenza, il programmatore del client può essere indotto a scrivere codice inefficiente (es. chiamate remote in un ciclo) che "uccide" le prestazioni, non rendendosi conto del costo di ogni chiamata.
- **Aumento della Complessità:** Il pattern è significativamente più complesso di una chiamata locale. Introduce molte nuove classi (almeno 4 in più: Proxy client, Proxy server, Forwarder, Receiver) e molti nuovi punti di fallimento (rete, marshalling, unmarshalling, risoluzione dei nomi).
- **Nomi vs. Locazioni:** Le slide evidenziano una distinzione importante. Un Remote Proxy "puro" deve conoscere la *locazione* fisica del **RealSubject**. L'uso di **Forwarder-Receiver** migliora questo: il **Peer** (proxy) deve conoscere solo il *nome* logico del peer destinatario, mentre il **Forwarder** si occupa della risoluzione da nome a indirizzo fisico.

# Chapter 2

## Strategie di Distribuzione: Remote Facade e DTO

Nei sistemi distribuiti, una chiamata di metodo da un host a un altro è estremamente lenta (es. 20.000 ns) rispetto a una chiamata interna allo stesso processo (es. 1 ns). Questo divario di prestazioni impone un cambiamento radicale nel modo in cui le interfacce vengono progettate.

- **Oggetti Locali:** Hanno interfacce *fine-grained* (a grana fine), con molti piccoli metodi (es. `getStreet()`, `setStreet()`, `getCity()`, `setCity()`). Questo approccio, tipico della programmazione OO, promuove la flessibilità.
- **Oggetti Remoti:** Devono avere interfacce *coarse-grained* (a grana grossa). Poiché ogni chiamata remota è lenta, è fondamentale combinare più operazioni in un'unica chiamata (es. `setAddressData(street, city, zip)`).

Il problema è che programmare con interfacce coarse-grained è difficile, complica il codice e riduce la flessibilità. Questo capitolo introduce due pattern che risolvono questo dilemma.

### 2.1 Il Design Pattern Remote Facade

#### 2.1.1 Intento e Problema

L'intento del **Remote Facade** è fornire un'interfaccia *coarse-grained* (a grana grossa) che funge da facciata per un insieme di oggetti *fine-grained* (a grana fine), con l'obiettivo di migliorare l'efficienza sulla rete.

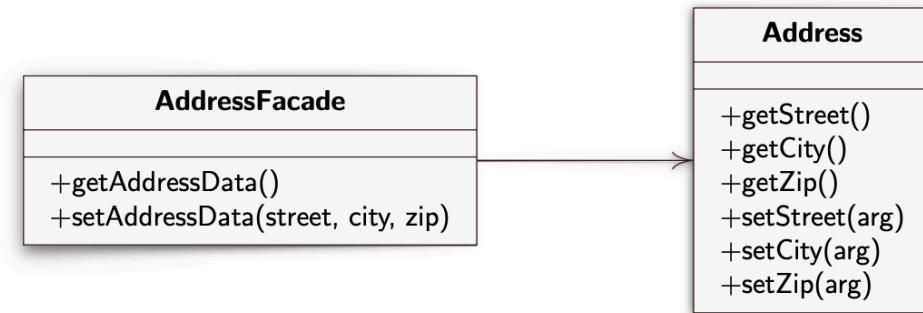
Il problema, come accennato, è che le chiamate inter-processo sono costose. Un'interfaccia fine-grained remota genererebbe un traffico di rete insostenibile (es. una chiamata per ogni campo di un form). D'altra parte, un'interfaccia coarse-grained complica la programmazione e rende il codice meno chiaro.

#### 2.1.2 Soluzione

La soluzione è una netta separazione di responsabilità:

1. **Oggetti di Dominio (Fine-Grained)**: La logica di business complessa rimane in oggetti piccoli e fine-grained (es. `Address`), progettati per collaborare *all'interno dello stesso processo*. Nessuno di questi oggetti ha un'interfaccia remota.
2. **Remote Facade (Coarse-Grained)**: Si introduce una nuova classe, il `RemoteFacade` (es. `AddressFacade`), il cui unico scopo è fornire un'interfaccia remota coarse-grained.

Questa facciata **non contiene logica di dominio**. Il suo solo compito è **tradurre** le chiamate coarse-grained che riceve dalla rete nelle molteplici chiamate fine-grained necessarie sugli oggetti di dominio locali.

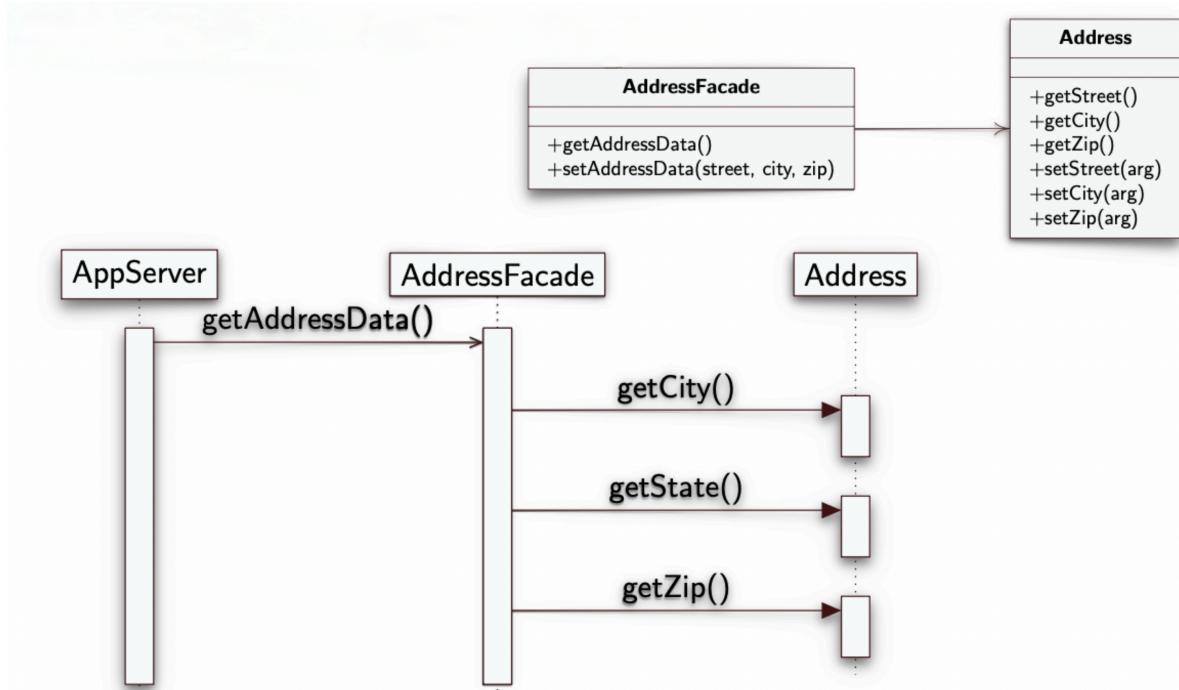


**Figure 2.1:** Confronto tra l'interfaccia coarse-grained del `AddressFacade` e quella fine-grained dell'oggetto di dominio `Address`.

### 2.1.3 Flusso di Esecuzione

Il `RemoteFacade` agisce come un traduttore. Spesso sostituisce una moltitudine di metodi `get` e `set` con un singolo "bulk accessor" (es. `getAddressData()`) e un singolo "bulk setter" (es. `setAddressData(...)`).

Come mostra il diagramma di sequenza, una singola chiamata remota `getAddressData()` dall'`AppServer` all'`AddressFacade` viene tradotta dal facade in molteplici chiamate locali (e veloci) agli oggetti di dominio (es. `getCity()`, `getState()`, `getZip()` sull'oggetto `Address`).



**Figure 2.2:** Diagramma di sequenza di un `RemoteFacade` che traduce una chiamata coarse-grained in più chiamate fine-grained.

### 2.1.4 Altre Responsabilità del Facade

Poiché il `RemoteFacade` è il punto di ingresso (il "gateway") per tutte le chiamate remote, diventa un posto naturale per aggiungere altre responsabilità trasversali:

- **Controlli di Sicurezza:** È il punto ideale per avviare controlli di sicurezza, verificando tramite un'Access Control List (ACL) se l'utente può eseguire quel metodo.
- **Gestione delle Transazioni:** Un metodo del facade può avviare una transazione del database, orchestrare le varie chiamate agli oggetti di dominio e infine eseguire il `commit` (o `rollback`) prima di restituire la risposta. Questo è cruciale: la transazione non deve mai rimanere aperta quando si ritorna al client, a causa dei tempi lunghi e imprevedibili della rete.

## 2.2 Il Design Pattern Data Transfer Object (DTO)

Il `RemoteFacade` risolve il problema delle *chiamate* multiple, ma solleva una nuova domanda: come si trasferiscono i *dati* in modo efficiente? Un metodo come `getAddressData()` come restituisce i dati? E `setAddressData(...)` come li riceve?

### 2.2.1 Intento e Problema

L'intento del **Data Transfer Object (DTO)** è di essere un oggetto che **trasporta dati fra processi**, con lo scopo di ridurre ulteriormente il numero di chiamate a metodo.

Il problema è che, anche con un facade, trasferire i dati è scomodo:

- Usare tanti parametri nei metodi è complesso e poco manutenibile.

- Il valore di ritorno di un metodo è, di solito, singolo.
- Non si possono inviare gli oggetti di dominio (es. `Address`) direttamente sulla rete, perché sono spesso complessi, connessi ad altri oggetti, e impossibili da serializzare.

### 2.2.2 Soluzione

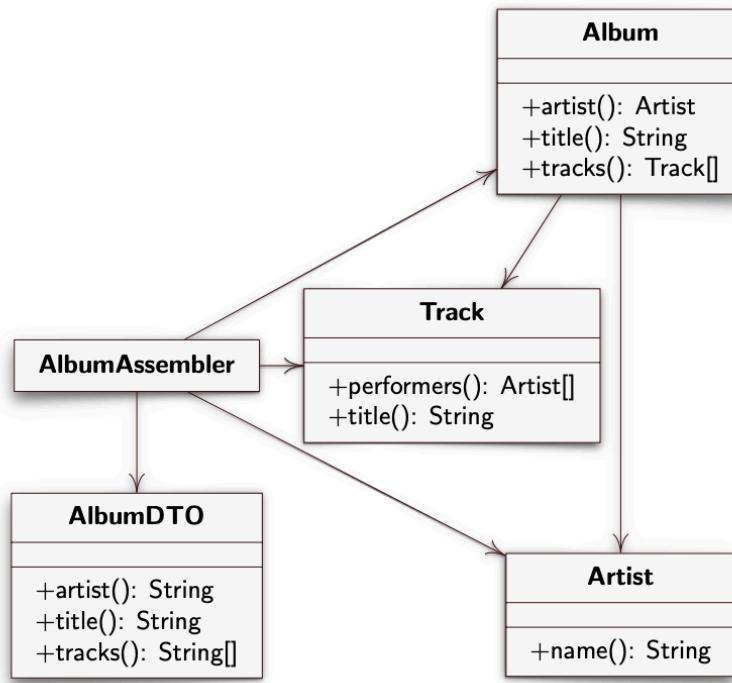
La soluzione è creare una classe `DTO` (es. `AlbumDTO`), un oggetto "stupido" il cui unico scopo è contenere i dati da trasferire.

- Un `DTO` contiene solo campi (dati primitivi, stringhe, o altri `DTO`) e i loro metodi `getter` e `setter`.
- **Non contiene logica di business.**
- Deve essere facilmente **serializzabile** (in binario, XML, JSON, ecc.) per poter viaggiare sulla rete.

### 2.2.3 L'Assembler

Per spostare i dati tra gli oggetti di dominio (ricchi e complessi) e i `DTO` (semplici e trasportabili), si usa un oggetto **Assembler** (es. `AlbumAssembler`).

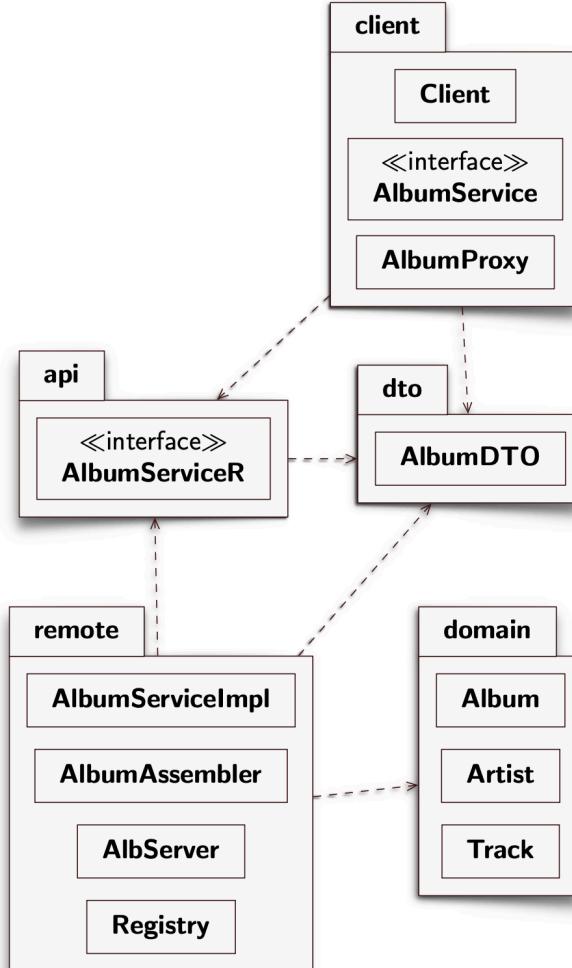
- **In uscita:** L'Assembler legge i dati da uno o più oggetti di dominio (es. `Album`, `Artist`) e li "assembra" in un unico `AlbumDTO` da inviare al client.
- **In ingresso:** L'Assembler riceve un `AlbumDTO` dal client, lo "smonta" e usa i dati per creare o aggiornare i veri oggetti di dominio.



**Figure 2.3:** Diagramma UML che mostra il ruolo dell'`AlbumAssembler` nel mediare tra gli oggetti di dominio (`Album`, `Track`, `Artist`) e il `AlbumDTO`.

## 2.3 Architettura di Esempio (RMI)

Le slide mostrano un'architettura completa che combina tutti questi pattern e li implementa utilizzando **Remote Method Invocation (RMI)** di Java come tecnologia di rete.



**Figure 2.4:** Architettura di un'applicazione che usa Remote Facade e DTO.

L'architettura è divisa in package:

- **Lato Server:** Contiene i package **domain** (gli oggetti fine-grained), **remote** (il RemoteFacade, l'Assembler), **api** (l'interfaccia remota) e **dto** (i DTO).
- **Lato Client:** Contiene i package **client** (l'applicazione), **api** (l'interfaccia remota) e **dto** (i DTO).

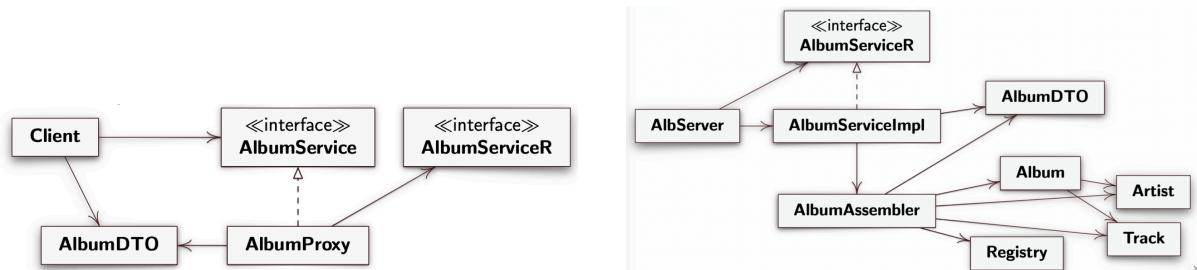
Come si può notare, l'interfaccia **api** e i **dto** sono gli unici componenti **condivisi** tra client e server.

### 2.3.1 Implementazione (RMI)

RMI è una tecnologia che permette di chiamare metodi su un oggetto che risiede in un altro processo o su un altro host. L'obiettivo è fare in modo che la chiamata sembri locale. Il

funzionamento di un sistema basato su RMI è interamente costruito sull'applicazione congiunta dei pattern **Remote Proxy**, **Remote Facade** e **Data Transfer Object (DTO)**. Con RMI, la comunicazione avviene tramite un'interfaccia Java (es. `AlbumServiceR`) che estende `java.rmi.Remote`.

- **Lato Server:** La classe `AlbServer` crea un'istanza del facade (`AlbumServiceImpl`) e la registra su un *RMI Registry* (un servizio di naming).
- **Lato Client:** Il **Client** non contatta direttamente il server. Esegue un *lookup* sul Registry per ottenere un riferimento all'oggetto remoto. RMI gli restituisce un **Proxy** (un `RemoteProxy` generato automaticamente da RMI) che il client usa in modo trasparente.



**Figure 2.5:** Architettura RMI lato server (sinistra) e lato client (destra).

Per schermare ulteriormente il **Client** dalla complessità di RMI (come la gestione delle `RemoteException`), l'esempio introduce un ulteriore **Proxy** (il pattern Proxy!) chiamato `AlbumProxy` sul lato client. Questo proxy implementa un'interfaccia locale "pulita" (`AlbumService`) e incapsula al suo interno la logica di lookup RMI e la gestione delle eccezioni di rete.

## 2.4 Conseguenze Generali: Pro e Contro

L'adozione congiunta dei pattern **Remote Facade** e **Data Transfer Object (DTO)** è una strategia di progettazione fondamentale per i sistemi distribuiti. Risolve il problema centrale della latenza di rete, ma introduce un nuovo insieme di compromessi (trade-off) architetturali.

### 2.4.1 Benefici (Pro)

- **Efficienza della Rete (Remote Facade):** È il vantaggio principale. Il facade, con la sua interfaccia *coarse-grained*, riduce drasticamente il numero di chiamate remote necessarie per un'operazione, raggruppando più interazioni in una sola.
- **Conservazione del Design OO (Remote Facade):** Questo è un beneficio cruciale. Il facade permette al *domain model* (gli oggetti di business come `Address` o `Album`) di rimanere *fine-grained*, flessibile e aderente ai principi OO. La "brutta" interfaccia *coarse-grained* è relegata a un livello separato (il facade) e non "inquinà" la logica di business.
- **Efficienza dei Dati (DTO):** Il DTO complementa il facade. Risolve il problema di come trasferire i dati in modo efficiente in una singola chiamata, aggregando più

informazioni in un unico oggetto serializzabile e risolvendo il problema dei parametri multipli o dei valori di ritorno complessi.

- **Separazione delle Responsabilità (Remote Facade):** Il facade separa nettamente le responsabilità di distribuzione dalla logica di business. Il facade gestisce la rete, la traduzione e, come visto, diventa il punto di ingresso ideale per responsabilità trasversali come la **sicurezza** (controlli ACL) e la **gestione delle transazioni** (avvio, commit, rollback).
- **Disaccoppiamento Cliente-Server (DTO):** Il DTO disaccoppia il client dall'implementazione interna del modello di dominio del server. Il client deve conoscere solo la struttura "piatta" del DTO, non le complesse interconnessioni tra gli oggetti di dominio (che, tra l'altro, sarebbero difficili o impossibili da serializzare e inviare).

#### 2.4.2 Svantaggi e Considerazioni (Contro)

- **Duplicazione dei Dati (DTO):** È lo svantaggio più evidente. Si crea una duplicazione delle classi che rappresentano i dati. Ora esistono sia gli oggetti di dominio (es. `Album`) sia i DTO (es. `AlbumDTO`) che, in gran parte, contengono le stesse informazioni.
- **Manutenzione dell'Assembler (DTO):** Come conseguenza diretta della duplicazione, è necessario un nuovo componente, l'**Assembler**, il cui unico scopo è tradurre e copiare i dati tra gli oggetti di dominio e i DTO. Questo è spesso codice *boilerplate* (ripetitivo) che deve essere scritto, testato e, soprattutto, **manutenuto**. Ogni volta che si aggiunge un campo all'oggetto `Album`, bisogna ricordarsi di aggiornare anche il `AlbumDTO` e l'`AlbumAssembler`.
- **Rigidità del Facade (Remote Facade):** Un'interfaccia coarse-grained è, per definizione, meno flessibile. È progettata per le esigenze specifiche di un client. Se un nuovo client ha bisogno di una combinazione di dati leggermente diversa, l'amministratore si trova di fronte a una scelta scomoda: aggiungere un nuovo metodo al facade (aumentandone la complessità), creare un nuovo facade, o costringere il client a chiamare un metodo che restituisce più dati del necessario.
- **Fragilità della Serializzazione (DTO):** Come menzionato nelle slide, alcuni formati di serializzazione (come quello binario nativo di Java) sono fragili. Se il server aggiunge o rimuove un campo dal DTO e il client non viene aggiornato contemporaneamente, l'applicazione client fallirà con un errore di deserializzazione. (Formati come XML o JSON sono generalmente più tolleranti a questi cambiamenti).

# Chapter 3

## Gestione dello Stato della Sessione

Una **sessione** è un'interazione a lungo termine tra un client e un server. Può consistere in una o più richieste, tipicamente iniziando con un login e terminando con un logout o un abbandono.

### 3.1 Il Dilemma: Stateful vs. Stateless

Il problema fondamentale nella gestione delle sessioni è decidere *dove* conservare lo "stato".

- **Server Stateless (Senza Stato):** Un server stateless non mantiene (conserva) nessuno stato tra le richieste. Ogni richiesta è indipendente e contiene tutte le informazioni necessarie per essere processata. Quando la risposta è data, lo stato dell'oggetto che ha processato la richiesta è inutile e viene scartato.
- **Server Stateful (Con Stato):** Un server stateful tiene traccia di ciò che il client ha fatto in precedenza (es. la lista di pagine visitate).

Un server con molti utenti dovrebbe essere **stateless** per evitare un enorme uso di risorse. Un server stateful, infatti, deve usare lo stesso oggetto per servire tutte le richieste dello stesso utente. Se si hanno 100 utenti, servono 100 oggetti server in memoria, ognuno dei quali passa la maggior parte del tempo (es. il 90%) in attesa della prossima richiesta.

Tuttavia, funzionalità comuni come il **carrello della spesa** (Shopping Cart) richiedono per loro natura uno stato che persista tra le richieste. Questo è lo **Stato della Sessione**. Le slide identificano tre scelte di base per conservare questo stato: sul Client, sul Server, o sul Database.

### 3.2 Pattern 1: Client Session State

#### 3.2.1 Intento e Soluzione

L'intento di questo pattern è di conservare l'intero stato della sessione **sul client**.

In questo approccio, il server rimane completamente stateless. Il client ha la responsabilità di inviare tutti i dati di sessione (l'intero "carrello") con *ogni singola richiesta*.

Il server processa la richiesta, modifica lo stato se necessario, e restituisce *tutto* il nuovo stato al client insieme alla risposta.

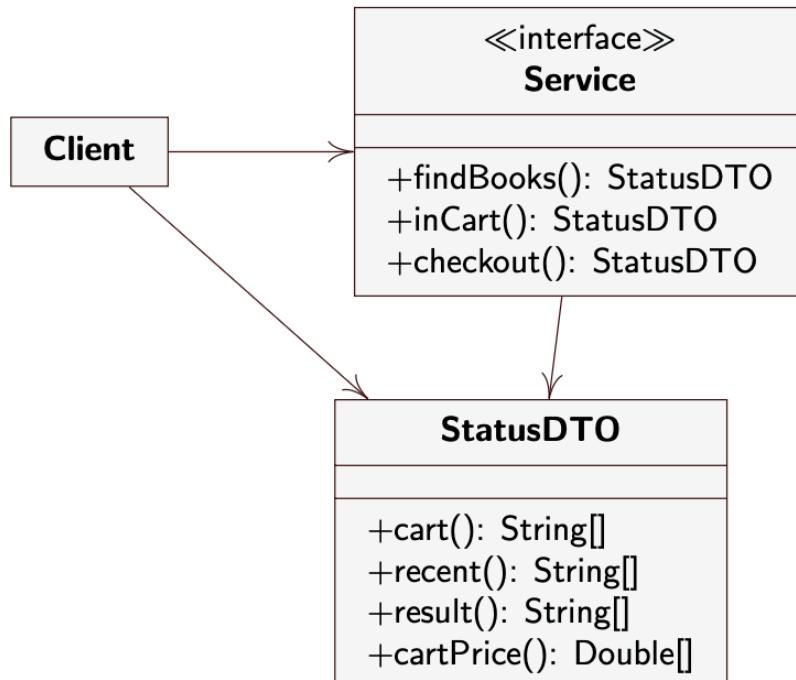
Per un'interfaccia HTML, ci sono tre modi per implementarlo:

- **Parametri dell'URL:** Funziona bene per pochi dati (es. un ID di sessione), ma la dimensione dell'URL è limitata.
- **Campi Nascosti (Hidden Fields):** Lo stato della sessione (es. serializzato in XML) viene inserito in un tag `<input type="hidden">` e inviato con il form.
- **Cookie:** I cookie vengono inviati avanti e indietro automaticamente. Funzionano solo per un singolo nome di dominio e se l'utente li disabilita, il sito smette di funzionare.

### 3.2.2 Esempio di Implementazione

L'esempio nelle slide mostra l'uso di un **Data Transfer Object** (DTO) per gestire lo stato.

- L'oggetto **StatusDTO** contiene tutti i dati della sessione (il carrello, le ricerche recenti, il prezzo).
- **Ogni metodo** dell'interfaccia **Service** (es. `findBooks()`, `inCart()`) è costretto a ricevere e/o restituire l'intero **StatusDTO**.
- Questo mostra chiaramente che è il client a gestire lo stato e a passarlo al server ad ogni chiamata.



**Figure 3.1:** Diagramma UML del pattern Client Session State. Lo **StatusDTO** viene passato in ogni chiamata.

### 3.2.3 Conseguenze: Pro e Contro

- **Pro (Scalabilità):** È il vantaggio principale. Il server è stateless, quindi qualsiasi server in un cluster può gestire la prossima richiesta. Questo abilita un *failover* perfetto: se un server va giù, il client ripete la richiesta a un altro server (invia lo stato) e la sessione continua senza interruzioni.
- **Pro (Risorse):** Il server non consuma memoria per tenere traccia delle sessioni inattive.
- **Contro (Performance):** Con grandi quantità di dati (un carrello con molti oggetti), il tempo e la banda per trasferire l'intero stato avanti e indietro ad ogni richiesta possono diventare proibitivi.
- **Contro (Sicurezza):** I dati di sessione inviati al client possono essere **visti e alterati** dall'utente. Questo costringe il server a *ri-convalidare* tutti i dati (es. i prezzi nel carrello) ad ogni singola richiesta.
- **Contro (Performance/Sicurezza):** Per mitigare il punto precedente, si possono criptare i dati. Ma criptare e decriptare l'intero stato ad ogni richiesta può introdurre un notevole sovraccarico (overhead) di CPU.

## 3.3 Pattern 2: Server Session State

### 3.3.1 Intento e Soluzione

L'intento è di conservare lo stato della sessione **sul server**, tipicamente in memoria o serializzato su disco.

Al client viene inviato solo un piccolo **ID di sessione** (spesso tramite cookie). Quando il client fa una richiesta, invia solo quell'ID. Il server usa l'ID per recuperare l'oggetto di sessione corretto da una Map in memoria e servire la richiesta.

Il problema principale è la **memoria limitata**. Per risolvere questo, si usa la **passivazione**: l'oggetto di sessione viene serializzato (es. in binario o XML) e salvato su uno storage (es. file system o database).

### 3.3.2 Clustering e Failover

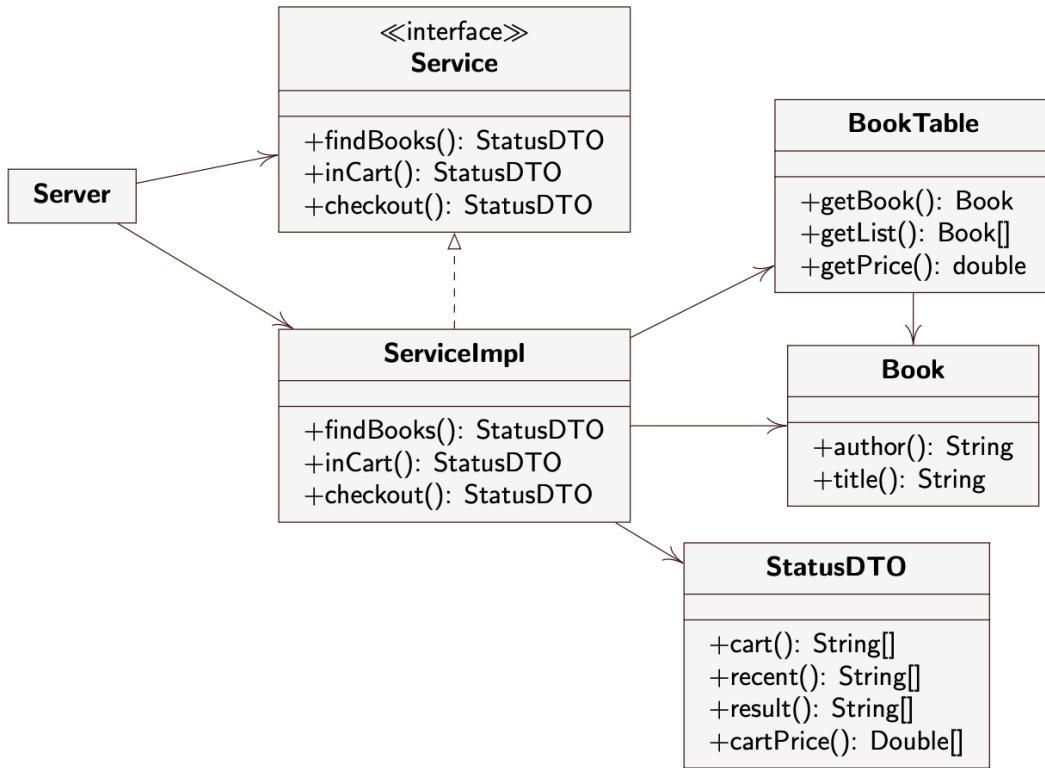
Se lo storage di sessione serializzato è *condiviso* (es. un file system di rete o un database centrale), questo pattern abilita il **Clustering** (distribuzione di richieste a più server) e il **Failover** (se un host va giù, un altro può recuperare lo stato dallo storage condiviso).

### 3.3.3 Esempio di Implementazione

L'esempio mostra un'architettura molto diversa dal precedente:

- L'interfaccia **Service** (lato client) ha metodi semplici (es. `findBooks()`) che non passano più lo stato. Il client è "ignaro".
- È il **ServiceImpl** (lato server) che ora contiene la logica `getSession()`, `isExpired()`, `saveToDisk()`.

- Il server mantiene un oggetto **Status** (con `cart`, `recent`, `timestamp`) in memoria e lo salva/legge tramite **DiskSer** (che rappresenta la serializzazione).



**Figure 3.2:** Diagramma UML del pattern Server Session State. La logica di sessione è tutta interna al server.

### 3.3.4 Conseguenze: Pro e Contro

- Pro (Semplicità):** È un pattern molto semplice da implementare (specialmente se si ha un solo server e memoria sufficiente).
- Pro (Sicurezza):** Lo stato non lascia mai il server, quindi non può essere visto o manipolato dal client.
- Contro (Scalabilità/Risorse):** È il principale svantaggio. Il server deve consumare memoria per ogni sessione attiva. In un cluster, la gestione del failover (la "passivazione" e il recupero dello stato) può diventare molto complicata.
- Contro (Manutenzione):** Il server deve occuparsi della gestione delle sessioni abbandonate (es. cancellando periodicamente i vecchi file di sessione o gli oggetti in memoria).

## 3.4 Pattern 3: Database Session State

### 3.4.1 Intento e Soluzione

L'intento è di conservare i dati di sessione **come dati dedicati su un database**. Questo pattern cerca di unire i vantaggi dei due precedenti: il server rimane stateless (come nel

Client Session State), ma i dati rimangono sicuri sul server (come nel Server Session State).

Il flusso è il seguente:

1. Il client invia una richiesta con un **ID di sessione**.
2. Il server usa l'ID come chiave per **recuperare i dati** della sessione dal database.
3. Il server processa la richiesta (modificando i dati in memoria).
4. Il server **salva i dati aggiornati** sul database prima di rispondere.

### 3.4.2 Gestione dei Dati "Pending"

I dati di sessione (es. un carrello) sono "intermedi" o *pending* (in sospeso) e non dovrebbero essere mischiati con i dati permanenti (es. ordini confermati). Ci sono due soluzioni:

- **Soluzione 1 (Invasiva):** Aggiungere un campo (es. `isPending` o `sessionId`) alle tabelle reali. Questo è "invasivo" perché costringe le altre applicazioni a conoscere e gestire questo flag.
- **Soluzione 2 (Migliore):** Usare un set di **tabelle separate** solo per i dati di sessione. Quando la sessione è confermata (es. `checkout`), i dati vengono copiati nelle tabelle permanenti. Questo non è invasivo e facilita l'applicazione di regole di validazione diverse tra dati pending e dati permanenti.

### 3.4.3 Conseguenze: Pro e Contro

- **Pro (Scalabilità e Failover):** È il vantaggio principale. I server applicativi diventano completamente **stateless** (lo stato è nel DB), il che abilita il *pooling* di oggetti e rende il clustering e il failover estremamente facili.
- **Pro (Semplicità di Programmazione):** La gestione dello stato (es. la gestione della concorrenza) viene delegata al database.
- **Contro (Performance):** È lo svantaggio principale. Ogni singola richiesta dell'utente **richiede almeno un accesso al database** (spesso una lettura e una scrittura). Questo può creare un collo di bottiglia sul database.
- **Mitigazione (Cache):** Il problema delle prestazioni può essere ridotto utilizzando una cache (es. Redis, Memcached) per i dati di sessione, evitando l'accesso al disco per ogni richiesta.
- **Contro (Manutenzione):** Come per il Server Session State, è necessario un meccanismo (es. un daemon) per trovare e cancellare i dati delle sessioni vecchie o abbandonate, usando un *timestamp* di ultima interazione.

## 3.5 Pattern Correlato: Serialized LOB

### 3.5.1 Intento e Problema

Il pattern **Serialized LOB (Large Object)** è una tecnica di persistenza che può essere usata per implementare il *Server Session State* o il *Database Session State*.

L'intento è di conservare un intero **grafo di oggetti** (come un complesso oggetto di sessione) serializzandolo dentro un **singolo grande oggetto (LOB)** in un campo del database.

Il problema che risolve è che i sistemi OO sono grafi complessi di piccoli oggetti. Mappare queste informazioni in uno schema relazionale è difficile e *lento*, poiché manipolare lo schema richiede molti JOIN SQL.

### 3.5.2 Soluzione e Conseguenze

La soluzione è serializzare l'intero grafo in un singolo **BLOB** (Binary Large Object) o **CLOB** (Character Large Object) e salvarlo in una sola riga e colonna del database.

- **Pro:** È estremamente veloce in lettura e scrittura, poiché evita tutti i JOIN.
- **Contro (Oltre alla Serializzazione):** Il database non può "vedere" dentro il LOB. È impossibile eseguire una query SQL come `SELECT * WHERE cart.item.price > 10`.
- **Contro (Duplicazione Dati):** Bisogna fare attenzione a non duplicare i dati. Se il LOB dei dettagli del cliente viene copiato in ogni riga della tabella `Ordini`, si crea un'enorme duplicazione. È meglio mettere il LOB in una tabella `Clienti` e usare un link.
- **Contro (Fragilità):** La serializzazione binaria è fragile. Se si aggiunge o rimuove un campo da una classe nel grafo, si potrebbe non essere più in grado di deserializzare i vecchi LOB (errore a runtime), a meno che il campo non sia dichiarato `transient`.

# Chapter 4

## Il Design Pattern Memento

### 4.1 Intento e Problema

L'intento del pattern **Memento** è quello di catturare ed esternalizzare lo stato interno di un oggetto, **senza violare l'incapsulamento**, in modo che l'oggetto possa essere riportato a questo stato in un momento successivo.

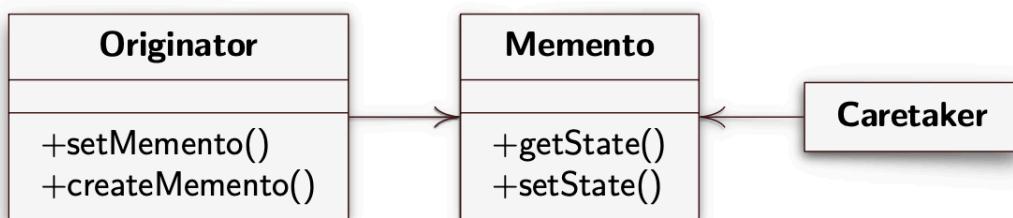
Il **problema** che risolve è comune nei sistemi software. Spesso è necessario registrare lo stato interno di un oggetto, ad esempio per implementare meccanismi di *checkpoint* o di *undo* (annulla), che permettono di ripristinare uno stato precedente in caso di errore o operazione fallita.

Tuttavia, un principio fondamentale della programmazione Object-Oriented è l'incapsulamento: gli oggetti normalmente nascondono il loro stato, rendendolo inaccessibile dall'esterno. Esporre pubblicamente lo stato (es. con metodi **getter** e **setter** per ogni campo interno) per permetterne il salvataggio violerebbe l'incapsulamento, compromettendo l'affidabilità e l'estensibilità del codice.

### 4.2 Soluzione e Struttura

La soluzione è introdurre un oggetto, chiamato **Memento**, che ha il compito di immagazzinare una copia dello stato interno di un altro oggetto (detto **Originator**).

Il flusso è gestito da tre partecipanti:



**Figure 4.1:** Diagramma UML delle classi del pattern Memento.

### 4.2.1 Partecipanti

- **Originator (Creatore)**: È l'oggetto che possiede lo stato interno che si desidera salvare (es. un editor di testo, un oggetto di sessione). È l'unico responsabile della creazione di un **Memento** che contiene una copia del suo stato interno attuale (`createMemento()`). In seguito, l'**Originator** può usare un oggetto **Memento** per ripristinare il suo stato interno (`setMemento()`).
- **Memento (Ricordo)**: È l'oggetto che immagazzina lo stato interno dell'**Originator**. La struttura e i dati contenuti nel **Memento** sono una scelta implementativa del solo **Originator**. La caratteristica fondamentale del **Memento** è che **protegge i dati** da accessi esterni: idealmente, solo l'**Originator** che lo ha creato può accedere ai suoi contenuti.
- **Caretaker (Custode)**: È l'oggetto "cliente" che ha bisogno di salvare lo stato (es. il gestore del comando "undo"). Il **Caretaker** richiede all'**Originator** di creare un **Memento** e poi lo **custodisce** (es. in una lista). Crucialmente, il **Caretaker** **non usa e non esamina mai** il contenuto del **Memento**. Per lui è una "scatola nera" opaca.

Quando l'utente richiede un "undo", il **Caretaker** recupera l'ultimo **Memento** che ha salvato e lo restituisce all'**Originator** (tramite `setMemento()`), il quale sa come leggerlo per ripristinare il suo stato precedente.

## 4.3 Conseguenze: Pro e Contro

L'uso del pattern Memento ha implicazioni dirette sulla progettazione, la performance e l'uso della memoria.

### 4.3.1 Benefici (Pro)

- **Preservazione dell'Incapsulamento**: È il vantaggio principale. Il pattern evita di esporre informazioni che solo l'**Originator** dovrebbe conoscere (il suo stato interno), pur permettendo a queste informazioni di essere immagazzinate all'esterno dell'oggetto stesso.
- **Semplificazione dell'Originator**: L'**Originator** non deve gestire la complessità di salvare più versioni del proprio stato. Questa responsabilità è delegata al **Caretaker** (es. uno stack di **Memento** per la cronologia di undo). L'**Originator** sa solo come creare un **Memento** e come ripristinarsi da esso.

### 4.3.2 Svantaggi e Considerazioni (Contro)

- **Costo di Performance e Memoria**: Questo è lo svantaggio principale. Se l'**Originator** ha una grande quantità di stato, o se il salvataggio avviene molto di frequente, le prestazioni possono deteriorarsi e la quantità di memoria (o spazio su disco) utilizzata può diventare molto grande. Il **Caretaker** è responsabile della cancellazione dei **Memento**, ma non ha idea di quanto "pesino".
- **Difficoltà di Accesso**: Può essere tecnicamente difficile, a seconda del linguaggio di programmazione, garantire che *solo* l'**Originator** possa accedere allo stato del

Memento, mantenendolo nascosto a tutti gli altri, incluso il Caretaker.

- **Ottimizzazione (Stato Incrementale):** Una possibile mitigazione al problema delle prestazioni è non salvare l'intero stato ogni volta. Se i Memento vengono usati in una sequenza prevedibile (come per l'undo/redo), un Memento può conservare solo il *cambiamento incrementale* (il "delta") rispetto allo stato precedente, riducendo drasticamente il suo peso.

# Chapter 5

## Il Design Pattern Idempotent Receiver

### 5.1 Intento e Problema

L'intento del pattern **Idempotent Receiver** (Ricevitore Idempotente) è quello di identificare le richieste dei client in modo univoco, così da poter **ignorare le richieste duplicate** quando il client effettua un nuovo tentativo (retry).

Il **problema** è uno scenario inevitabile nei sistemi distribuiti:

1. Un client invia una richiesta (es. "Addebita 100 Euro a Bob").
2. Il client, a causa di un problema di rete o di un crash del server, non riceve una risposta.
3. Per il client è **impossibile sapere** se:
  - a) La richiesta non è mai arrivata al server.
  - b) La richiesta è arrivata, il server l'ha elaborata (Bob ha pagato!), ma la *risposta* è andata persa.
4. Nel dubbio, per garantire che l'operazione venga eseguita, il client **deve inviare nuovamente** la richiesta.

Se si verifica lo scenario (b), il server riceverà una richiesta duplicata. Se il server non è idempotente, elaborerà la richiesta una seconda volta e, nell'esempio, Bob pagherà 200 Euro.

Questo problema è aggravato dal fatto che molti sistemi affidabili usano un modello di consegna **"at-least-once"** (**almeno una volta**). Per garantire che un messaggio non vada perso, il sistema preferisce inviarlo più volte piuttosto che rischiare di non inviarlo affatto. Di conseguenza, la ricezione di duplicati è un evento *inevitabile* e deve essere gestito.

### 5.2 Soluzione: L'Operazione Idempotente

La soluzione è progettare il ricevitore (il server) affinché sia **idempotente**. Una operazione idempotente è un'operazione che può essere eseguita tante volte ottenendo lo

stesso effetto che si avrebbe eseguendola una sola volta.

- **Non idempotente:**  $x = x + 10$  (Eseguito due volte, aggiunge 20).
- **Idempotente:**  $x = 10$  (Eseguito due volte,  $x$  è sempre 10).

Il ricevitore deve quindi far sì che l'elaborazione di un messaggio più volte produca lo stesso effetto della sua elaborazione una sola volta.

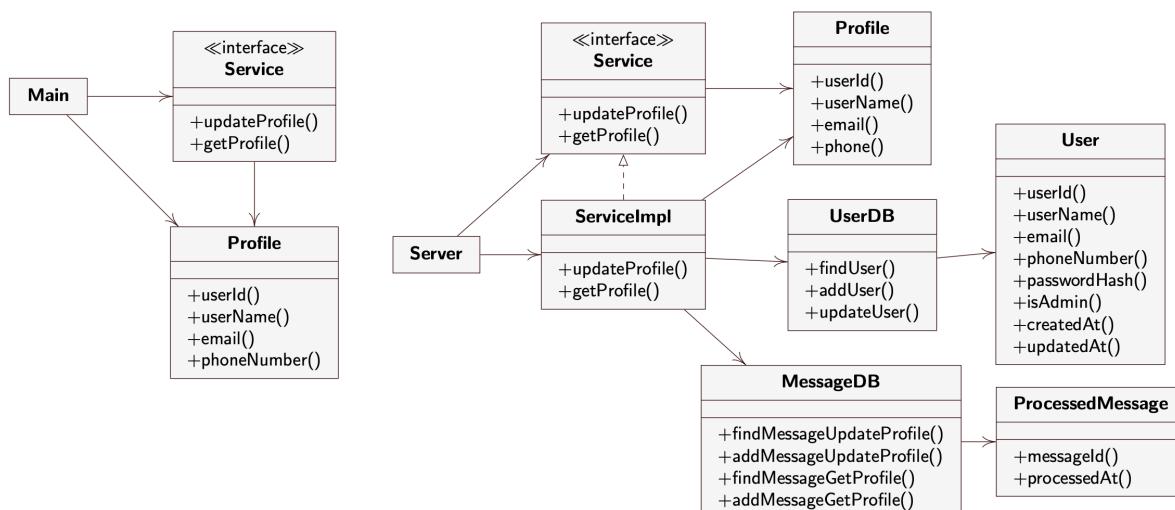
## 5.3 Flusso di Esecuzione

Il pattern non rende l'operazione di business (come "addebitare") magicamente idempotente, ma introduce una logica di *controllo* per prevenire la doppia esecuzione.

1. Il **client** (il mittente) deve creare un **ID Messaggio Univoco** per ogni richiesta (es. un numero sequenziale, un UUID, o un hash del contenuto).
2. Il **server** (il ricevitore) deve mantenere un **MessageDB**, ovvero un "registro" (su database o in memoria) di tutti gli ID dei messaggi che ha già elaborato.
3. Quando il server riceve una richiesta, **per prima cosa** controlla se l'ID di quel messaggio è già presente nel **MessageDB**.
4. **Caso A (ID Trovato):** La richiesta è un duplicato. Il server **non esegue** nuovamente la logica di business. Recupera la risposta che aveva salvato la prima volta e la invia di nuovo al client.
5. **Caso B (ID non Trovato):** La richiesta è nuova. Il server esegue la logica di business, salva l'ID del messaggio nel **MessageDB**, salva la risposta, e infine invia la risposta al client.

## 5.4 Struttura di Esempio

Il diagramma UML delle slide mostra un'implementazione pratica di questo flusso.



**Figure 5.1:** Diagramma UML di un Idempotent Receiver per un servizio di profili utente.

L'oggetto `ServiceImpl` (il `MessageReceiver`) orchestra la logica. Quando riceve una chiamata `updateProfile()`:

- **Passo 1 (Controllo):** Interroga il `MessageDB` (il registro) per vedere se l'ID di quella richiesta `updateProfile` è già stato processato (es. `findMessageUpdateProfile()`).
- **Passo 2a (Duplicato):** Se lo trova, recupera e restituisce la risposta salvata.
- **Passo 2b (Nuovo):** Se non lo trova, esegue l'aggiornamento reale sullo `UserDB` (il database di business), e *prima* di rispondere, salva l'ID e il risultato nel `MessageDB` (es. `addMessageUpdateProfile()`).

## 5.5 Gestione dello Storage (Cleanup)

Questa soluzione introduce un nuovo problema: il `MessageDB` cresce all'infinito. È necessaria una strategia di pulizia:

- **Timeout:** Le richieste già elaborate e conservate vengono cancellate dal server dopo un certo periodo di tempo (es. 24 ore).
- **Numerazione Sequenziale:** Se le richieste del client sono numerate in modo progressivo (1, 2, 3...), quando il server riceve la richiesta n. 100, può tranquillamente cancellare dal `MessageDB` tutte le richieste salvate con numero minore di 100 provenienti da quel client.

## 5.6 Conseguenze: Pro e Contro

### 5.6.1 Benefici (Pro)

- **Garanzia "Exactly-Once":** Questo pattern permette di simulare un'elaborazione "exactly-once" (esattamente una volta), anche se si opera in un sistema che garantisce solo una consegna "at-least-once" (almeno una volta).
- **Affidabilità e Consistenza:** Migliora drasticamente l'affidabilità e la consistenza dei dati dell'applicazione (es. Bob non paga due volte).
- **Integrazione Microservizi:** È un pattern fondamentale nelle architetture a microservizi. Se più microservizi comunicano tra loro (e quindi possono fallire e ri-tentare), possono condividere un `MessageDB` (es. su Redis o un database) per garantire che le operazioni tra di loro siano idempotenti.

### 5.6.2 Svantaggi e Considerazioni (Contro)

- **Overhead di Performance e Storage:** È lo svantaggio principale. Il server ora ha un carico di lavoro aggiuntivo per *ogni singola richiesta*: deve eseguire almeno una ricerca (lookup) sul `MessageDB` e occupare più memoria o spazio su disco per salvare gli ID.
- **Complessità di Gestione (Cleanup):** Introduce la necessità di gestire la scadenza e la cancellazione delle richieste registrate, per evitare che lo storage cresca all'infinito.

- **Accoppiamento con il Client:** Il pattern richiede che il client "partecipi" attivamente, generando e inviando un ID di richiesta univoco.



**Università  
di Catania**

**UNIVERSITY OF CATANIA**

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

A MASTER OF SCIENCE DEGREE IN COMPUTER SCIENCE.

---

*Alfio Spoto*

Ingegneria dei sistemi distribuiti

---

APPUNTI LEZIONI ISD  
III

---

---

Academic Year 2024 - 2025

# Contents

<b>1 Ottimizzazione della Comunicazione: Request Batch</b>	<b>1</b>
1.1 Intento e Problema nel Contesto Distribuito . . . . .	1
1.1.1 Intento del Pattern . . . . .	1
1.1.2 Il Problema della Granularità Fine . . . . .	1
1.2 Soluzione Architetturale e Meccanismo . . . . .	1
1.2.1 Componenti Strutturali del Pattern . . . . .	2
1.2.2 Fasi del Flusso Operativo . . . . .	2
1.3 Conseguenze: Pro e Contro . . . . .	3
1.3.1 Benefici (Pro) . . . . .	3
1.3.2 Compromessi e Svantaggi (Contro) . . . . .	3
<b>2 Consistency e Fault Tolerance: Leader and Followers</b>	<b>5</b>
2.1 Majority Quorum . . . . .	5
2.1.1 Intento e Motivazione . . . . .	5
2.1.2 Il Problema della Consistenza . . . . .	5
2.1.3 Soluzione: Calcolo del Quorum . . . . .	6
2.1.3.1 Quorum Flessibili . . . . .	6
2.2 Leader and Followers . . . . .	6
2.2.1 Intento . . . . .	6
2.2.2 Flusso delle Richieste . . . . .	6
2.2.3 Elezione del Leader . . . . .	6
2.2.3.1 Criteri di Scelta . . . . .	7
2.3 Heart Beat . . . . .	7
2.3.1 Intento e Meccanismo . . . . .	7
2.3.2 Configurazione dei Timer . . . . .	7
2.3.3 Trade-off . . . . .	7
2.4 Generation Clock (Epoca) . . . . .	7
2.4.1 Problema: Il Leader Zombie . . . . .	7
2.4.2 Soluzione . . . . .	8
<b>3 Il Pattern Request Pipeline</b>	<b>9</b>
3.1 Intento e Problema . . . . .	9
3.2 Soluzione e Funzionamento . . . . .	9
3.3 Implementazione e Meccanismi di Controllo . . . . .	9
3.3.1 Saturazione e Limiti (Inflight Requests) . . . . .	10
3.3.2 Gestione dell'Ordine e dei Fallimenti . . . . .	10
3.4 Esempio Tecnico: CompletableFuture in Java . . . . .	10

3.5	Conseguenze . . . . .	10
3.5.1	Vantaggi (Pro) . . . . .	10
3.5.2	Svantaggi e Sfide (Contro) . . . . .	10
<b>4</b>	<b>Programmazione Asincrona: CompletableFuture</b>	<b>11</b>
4.1	Evoluzione dai Future a CompletableFuture . . . . .	11
4.2	Dettaglio Tecnico dei Metodi . . . . .	11
4.2.1	1. Avvio dell'Esecuzione Asincrona . . . . .	11
4.2.2	2. Pipeline di Trasformazione (Callback) . . . . .	12
4.2.3	3. Composizione e Combinazione . . . . .	12
4.2.4	4. Controllo Manuale e Stato . . . . .	13
4.2.5	5. Coordinamento Multiplo . . . . .	13
4.2.6	6. Recupero del Risultato (Blocking) . . . . .	14
4.3	Gestione dei Workflow: Decomposizione e Coordinamento a Step . . . . .	14
4.3.1	Analisi Architetturale della Struttura a Tre Step . . . . .	15
4.4	Strategie di Composizione: Analisi Comparativa . . . . .	15
4.5	Sintesi del Pattern Request Pipeline . . . . .	16
<b>5</b>	<b>Design Pattern per la Resilienza e la Stabilità</b>	<b>17</b>
5.1	Il Concetto di Stabilità nei Sistemi Distribuiti . . . . .	17
5.2	Timeout . . . . .	17
5.2.1	Meccanismo e Implementazione . . . . .	17
5.2.2	Pattern Gateway e QueryObject . . . . .	18
5.2.3	Strategie di Ritentativo (Retry) . . . . .	18
5.3	Circuit Breaker . . . . .	18
5.3.1	Modellazione e Implementazione: Il State Pattern . . . . .	19
5.3.2	Dinamica degli Stati e Transizioni . . . . .	20
5.4	Bulkheads (Paratie) . . . . .	21
5.4.1	Strategie di Partizionamento e Granularità . . . . .	21

# Chapter 1

## Ottimizzazione della Comunicazione: Request Batch

### 1.1 Intento e Problema nel Contesto Distribuito

#### 1.1.1 Intento del Pattern

L'intento del pattern **Request Batch** (o Aggregazione di Richieste) è quello di **ottimizzare l'utilizzo delle risorse di rete e computazionali** aggregando un insieme di richieste individuali in un'unica operazione logica e fisica.

#### 1.1.2 Il Problema della Granularità Fine

Nei sistemi distribuiti, la comunicazione tra processi remoti (inter-process communication, IPC) o tra nodi di rete è affetta da due costi principali:

1. **Latenza di Rete (Round-Trip Time, RTT):** Il tempo necessario affinché un messaggio raggiunga il destinatario e la risposta ritorni. Questo tempo è un costo fisso che si ripete per ogni singola richiesta.
2. **Overhead di Elaborazione:** Ogni singola richiesta (anche piccola) genera un overhead computazionale legato all'apertura/chiusura della connessione, alla gestione dei socket e, soprattutto, alle operazioni di *serializzazione* e *deserializzazione* dei dati (marshalling/unmarshalling) sia sul client che sul server.

Quando un sistema è soggetto a un numero elevato di **richieste di piccole dimensioni** (come 10.000 richieste per leggere singoli dati da un database), la ripetizione di questi costi per ogni transazione singola causa una **scarsa efficienza** e un **throughput limitato**.

Il Request Batch risolve questo problema riducendo il numero di *round-trip* totali e consolidando l'overhead di rete e serializzazione su un'unica operazione di massa.

### 1.2 Soluzione Architetturale e Meccanismo

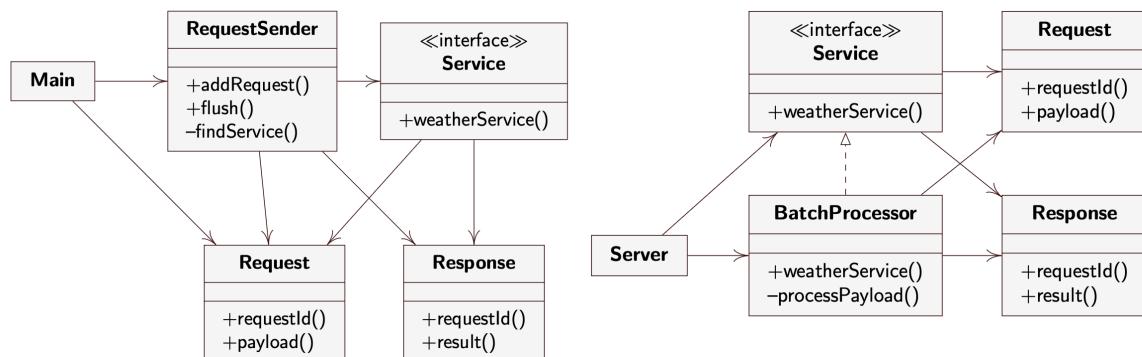
La soluzione si basa sull'introduzione di un livello di *buffer* sul lato client, o in un proxy intermedio, che accumula le richieste per un breve periodo di tempo o fino al raggiungi-

mento di una soglia definita, prima di inviarle come un unico lotto.

### 1.2.1 Componenti Strutturali del Pattern

Il pattern richiede l'implementazione dei seguenti componenti, che definiscono l'interfaccia e la logica di gestione:

- **Request Sender:** (Lato Client) Componente che espone i metodi pubblici come `+addRequest()` e `+flush()` per aggiungere una richiesta alla coda e forzarne l'invio. Si occupa anche dell'inizializzazione del tracciamento (`requestId`). Logica responsabile della costruzione del pacchetto di dati serializzato (il Batch) pronto per la trasmissione.
- **Request:** Oggetto dati fondamentale contenente un identificatore univoco (`requestId`) e il carico utile della richiesta (`payload`).
- **Batch Processor (Receiver):** (Lato Server) È l'implementazione del servizio remoto (spesso implementa l'interfaccia `Service`). Riceve il batch aggregato, lo elabora internamente e produce un unico batch di risposte.
- **Response:** Oggetto dati che contiene l'identificatore univoco (`requestId`) e il risultato dell'operazione (`result`).
- **Response Splitter:** (Lato Client) Logica che scomponete la risposta aggregata e la indirizza ai destinatari corretti.



**Figure 1.1:** Diagramma Strutturale UML dei Componenti del Pattern Request Batch (Riferimento: Slide 7).

### 1.2.2 Fasi del Flusso Operativo

Il meccanismo di Request Batch si articola nelle seguenti fasi :

1. **Accumulo (Client-Side):** Le richieste individuali vengono temporaneamente accodate dal `Request Sender`. La soglia di accumulo (tempo o numero massimo di richieste) deve essere bilanciata per ottimizzare la banda di rete e minimizzare la latenza indotta.
2. **Identificazione e Tracciamento:** Ad ogni richiesta accodata è associato un **ID univoco** (`requestId`). Questo ID è cruciale per permettere al client di mappare la risposta aggregata al richiedente originale.

3. **Invio Aggregato (Batching):** L'insieme delle richieste accumulate viene assemblato in un unico oggetto Batch e inviato come singola richiesta remota al server.
4. **Elaborazione Server-Side:** Il Batch Processor sul lato server riceve il lotto, lo disaggrega nelle richieste originali, le processa e consolida i risultati individuali in un'unica Response aggregata.
5. **Smistamento (Response Splitting):** Il client riceve la Response aggregata e lo Response Splitter utilizza l'requestId associato a ciascun risultato per smistare e restituire le risposte corrette ai rispettivi oggetti Request Sender che le avevano avviate.

## 1.3 Conseguenze: Pro e Contro

L'adozione del Request Batch è un *trade-off* che bilancia l'ottimizzazione del sistema a livello globale contro un potenziale degrado della latenza individuale.

### 1.3.1 Benefici (Pro)

I vantaggi sono di natura prestazionale e infrastrutturale:

- **Riduzione dell'Overhead di Rete:** Il numero totale di messaggi inviati e ricevuti è drasticamente ridotto. Di conseguenza, si abbassa l'overhead generato dalla gestione di connessioni separate.
- **Diminuzione del Round-Trip Time Totale:** Per un dato numero di operazioni, il tempo totale è inferiore poiché si riduce il numero di cicli di attesa (RTT) tra client e server.
- **Miglioramento del Throughput:** Il sistema è in grado di processare un volume maggiore di operazioni nell'unità di tempo grazie a un migliore utilizzo delle risorse del server e della banda di rete.
- **Efficienza della Serializzazione:** L'overhead di serializzazione/deserializzazione (costoso in termini di CPU) è consolidato in un'unica operazione sul batch, piuttosto che essere ripetuto per ogni singola piccola richiesta.

### 1.3.2 Compromessi e Svantaggi (Contro)

Il pattern introduce dei compromessi che devono essere gestiti:

- **Latenza di Accumulo (Blocking Latency):** Il principale svantaggio. Una richiesta individuale deve attendere di essere aggregata da un Request Sender (tempo o conteggio massimo) prima di essere inviata. Questo introduce una latenza artificiale e imprevedibile per il singolo utente.
- **Aumento della Complessità del Codice:** Sia lato client (per l'assemblaggio e lo smistamento delle risposte) che lato server (per la disaggregazione e l'elaborazione atomica o parziale del batch) la logica di gestione diventa più complessa.
- **Rischio di Fallimento del Batch:** Se la logica del Batch Processor non è robusta, il fallimento di una singola richiesta all'interno del lotto può compromettere

## *CHAPTER 1. OTTIMIZZAZIONE DELLA COMUNICAZIONE: REQUEST BATCH4*

l'intera operazione, causando la perdita (o la necessità di re-elaborazione) di tutte le altre richieste valide.

# Chapter 2

## Consistency e Fault Tolerance: Leader and Followers

### 2.1 Majority Quorum

#### 2.1.1 Intento e Motivazione

Il pattern **Majority Quorum** nasce con l'intento di evitare situazioni di *split-brain*, ovvero impedire che due gruppi di server prendano decisioni indipendenti e conflittuali. Per ogni decisione presa (lettura o scrittura), si richiede il consenso di una maggioranza di nodi.

#### 2.1.2 Il Problema della Consistenza

Nei sistemi distribuiti, garantire sia *Safety* (il sistema è sempre nello stato corretto) che *Liveness* (il sistema progredisce sempre) è complesso. Si potrebbe erroneamente pensare che un semplice quorum di scrittura sia sufficiente a garantire la consistenza forte. Tuttavia, analizziamo il seguente scenario di fallimento descritto nelle slide:

1. **Stato Iniziale:** Si ha un cluster di 3 server ( $S_1, S_2, S_3$ ) con una variabile  $x$  che vale inizialmente 1.
2. **Scrittura Parziale:** Uno scrittore invia l'aggiornamento  $x = 2$ . La scrittura ha successo **solo su  $S_1$** , ma fallisce su  $S_2$  e  $S_3$  (ad esempio per un timeout di rete).
3. **Lettura di  $C_1$  (Successo):** Il client  $C_1$  legge dai server  $S_1$  e  $S_2$ . Poiché  $S_1$  possiede il valore più recente ( $x = 2$ ),  $C_1$  ottiene correttamente il dato aggiornato.
4. **Fallimento:** Subito dopo, il server  $S_1$  (l'unico con il dato aggiornato) cade ("va giù").
5. **Lettura di  $C_2$  (Inconsistenza):** Un nuovo client  $C_2$  avvia una lettura. Non potendo contattare  $S_1$ , legge dai rimanenti  $S_2$  e  $S_3$ . Entrambi possiedono ancora il valore vecchio  $x = 1$ .

**Conclusione:** In questo caso,  $C_2$  legge un valore obsoleto ( $x = 1$ ) sebbene la sua lettura sia avvenuta temporalmente **dopo** che  $C_1$  aveva già letto il valore nuovo ( $x = 2$ ). Questo

viola la consistenza forte e dimostra che il quorum semplice non basta senza meccanismi aggiuntivi.

### 2.1.3 Soluzione: Calcolo del Quorum

Un cluster considera un aggiornamento confermato solo quando la **maggioranza dei nodi** lo ha validato.

- **Formula del Quorum:** Se un cluster ha  $n$  nodi, il quorum è  $n/2 + 1$ .
- **Tolleranza ai Fallimenti ( $f$ ):** Per tollerare  $f$  fallimenti, la dimensione del cluster deve essere  $2f + 1$ . Quindi per un cluster di dimensione  $n$  si ha una tolleranza ad  $f = \frac{n-1}{2}$  fallimenti.

È fondamentale notare che aumentare il numero di server non migliora linearmente la tolleranza ai guasti e riduce il throughput di scrittura (poiché bisogna attendere più conferme).

- **3 Server:** Quorum 2, tollera 1 fallimento.
- **4 Server:** Quorum 3, tollera **ancora solo 1 fallimento**.
- **5 Server:** Quorum 3, tollera 2 fallimenti.

#### 2.1.3.1 Quorum Flessibili

È possibile configurare quorum di dimensioni diverse per operazioni diverse, a patto che vi sia un'**intersezione non vuota** tra l'insieme di nodi di scrittura e quello di lettura. Poiché le letture sono tipicamente più frequenti, in un cluster di 5 nodi si potrebbe impostare un quorum di lettura pari a 2 e un quorum di scrittura pari a 4.

## 2.2 Leader and Followers

### 2.2.1 Intento

Il pattern **Leader and Followers** centralizza il coordinamento: un singolo server (Leader) è responsabile di coordinare la replicazione e prendere decisioni per l'intero cluster, propagandole agli altri server (Followers).

### 2.2.2 Flusso delle Richieste

- **Accettazione:** Il cluster accetta richieste solo se un Leader è stato eletto.
- **Gestione:** I client comunicano con il Leader. Se una richiesta arriva a un Follower, questo la deve propagare al Leader.

### 2.2.3 Elezione del Leader

Un server può trovarsi in tre stati: **Leader**, **Follower** o **Candidato** (Cerca Leader). L'elezione viene avviata all'avvio del sistema o quando il meccanismo di Heart Beat segnala che il Leader precedente non è più attivo.

### 2.2.3.1 Criteri di Scelta

Chi dovrebbe diventare il nuovo Leader?

1. **Il più aggiornato:** Si preferisce il server con il *Generation Clock* più alto e, a parità di generazione, quello con l'indice di log più recente nel Write-Ahead Log.
2. **Criteri di spareggio:** Se i server sono equamente aggiornati:
  - Si vota il server con **ID maggiore**.
  - Si utilizza un approccio basato su **timer casuali** (come nell'algoritmo RAFT), dove vince chi avvia l'elezione per primo.

Il Leader viene eletto solo se ottiene la **maggioranza dei voti** del cluster.

## 2.3 Heart Beat

### 2.3.1 Intento e Meccanismo

L'Heart Beat serve a dimostrare la **liveness** (disponibilità) di un server inviando messaggi periodici agli altri nodi. Questo permette di rilevare prontamente i fallimenti e avviare azioni correttive.

### 2.3.2 Configurazione dei Timer

La corretta configurazione si basa su tre grandezze che devono rispettare la disuguaglianza:

$$\text{Intervallo Timeout} > \text{Intervallo Richiesta} > \text{Round Trip Time (RTT)}$$

- **Intervallo di Richiesta:** Frequenza di invio del battito. Deve essere superiore al tempo di andata e ritorno (RTT) tra i server.
- **Intervallo di Timeout:** Tempo di attesa prima di dichiarare un server morto. È un multiplo dell'intervallo di richiesta.

**Esempio pratico:** Se RTT = 20ms, si può impostare l'Heartbeat a 100ms e il Timeout a 1s.

### 2.3.3 Trade-off

Un intervallo di heartbeat molto breve permette di rilevare velocemente i guasti, ma aumenta la probabilità di **falsi positivi** (falsi rilevamenti di fallimento dovuti a latenza di rete momentanea).

## 2.4 Generation Clock (Epoca)

### 2.4.1 Problema: Il Leader Zombie

Può accadere che un Leader venga temporaneamente disconnesso dalla rete (partitioning) ma il suo processo continui a girare. Nel frattempo, il cluster elegge un nuovo Leader. Quando la rete si ripristina, il vecchio Leader ("zombie") potrebbe inviare richieste di replica obsolete ai Followers, creando inconsistenze.

## 2.4.2 Soluzione

Si utilizza un **Generation Clock**, un numero monotonicamente crescente che identifica l'epoca (o generazione) corrente del cluster.

- **Incremento:** Il numero viene incrementato ad ogni nuova elezione.
- **Validazione:** Il Leader include la propria generazione in ogni richiesta (inclusi gli Heart Beat).
- **Reazione:** Se un Follower riceve una richiesta con una generazione inferiore alla propria, la respinge. Il vecchio Leader, ricevendo il rifiuto, comprende di essere obsoleto e degrada allo stato di Follower, aggiornando la propria generazione.

**Esempio:** Il Leader1 (Gen 1) si ferma per 5 secondi. Il cluster elegge un nuovo leader (Gen 2). Al ritorno, Leader1 invia comandi con Gen 1. I Followers (ora a Gen 2) respingono le richieste. Leader1 diventa Follower e adotta Gen 2.

# Chapter 3

## Il Pattern Request Pipeline

### 3.1 Intento e Problema

L'intento del pattern **Request Pipeline** è migliorare la latenza e il throughput di un sistema distribuito permettendo l'invio di richieste multiple su una connessione senza dover attendere la risposta alle richieste precedentemente inoltrate.

In un'architettura a cluster, l'utilizzo di un singolo canale di comunicazione può diventare un collo di bottiglia se implementato secondo il modello "stop-and-wait" (invio una richiesta, attendo la risposta, invio la successiva). Se si manda una sola richiesta alla volta, gran parte della capacità computazionale del server e della banda di rete viene sprecata, poiché il server rimarrà inattivo in attesa di nuovo lavoro e il client rimarrà bloccato in attesa del risultato.

### 3.2 Soluzione e Funzionamento

La soluzione consiste nel permettere ai nodi di inviare flussi di richieste in modo asincrono. Per ottenere questo comportamento, la struttura del nodo mittente e del nodo ricevente deve essere specializzata:

- **Gestione a due Thread:** Il nodo mittente utilizza due thread separati: uno dedicato esclusivamente all'invio delle richieste sul canale di rete e un altro dedicato alla ricezione delle risposte.
- **Asincronia:** Il mittente non si blocca dopo l'invio. Il ricevente, dal canto suo, elabora la richiesta immediatamente o la inserisce in una coda locale per massimizzare l'occupazione della propria CPU.

### 3.3 Implementazione e Meccanismi di Controllo

L'adozione di una pipeline introduce sfide gestionali che richiedono meccanismi specifici per evitare il collasso del sistema o l'incoerenza dei dati.

### 3.3.1 Saturazione e Limiti (Inflight Requests)

Un rischio concreto è che il nodo ricevente venga saturato da un numero eccessivo di richieste. Per ovviare a questo, si impone un **limite al numero di richieste in evase** (in-flight).

- Ogni nodo può avere solo un numero massimo di richieste inviate per le quali non ha ancora ricevuto risposta.
- Una volta raggiunto tale limite, il mittente viene inibito dall'invio di ulteriori messaggi finché non si libera un posto nella pipeline (ovvero finché non arriva una risposta).
- Le richieste in eccesso possono essere temporaneamente memorizzate in una coda lato client.

### 3.3.2 Gestione dell'Ordine e dei Fallimenti

Poiché le risposte potrebbero tornare in un ordine diverso da quello di invio o alcune richieste potrebbero fallire e richiedere un nuovo invio (retry), è necessario un meccanismo di riordino. Il server fornisce un **identificativo unico per richiesta**, che permette al client di associare correttamente ogni risposta alla rispettiva richiesta e di gestire eventuali messaggi fuori ordine.

## 3.4 Esempio Tecnico: CompletableFuture in Java

Un modo moderno per implementare questo pattern in Java, specialmente in contesti RMI (Remote Method Invocation), è l'uso di **CompletableFuture**. Il client non attende il risultato della chiamata remota nel thread principale:

```
1 // Il thread chiamante non si blocca
2 var response = CompletableFuture.supplyAsync(() -> sendRequest(request));
```

In questo scenario, il thread fornito da `supplyAsync` gestisce l'attesa della risposta RMI, mentre il thread principale del client è libero di continuare l'esecuzione o inviare altre richieste, realizzando di fatto una pipeline logica.

## 3.5 Conseguenze

### 3.5.1 Vantaggi (Pro)

- **Massimizzazione del Throughput**: La coda del server rimane sempre piena, garantendo che le risorse hardware siano costantemente utilizzate.
- **Riduzione della Latenza Percepita**: Il tempo totale per completare  $N$  richieste è drasticamente inferiore rispetto all'invio sequenziale.

### 3.5.2 Svantaggi e Sfide (Contro)

- **Necessità di Risorse**: Richiede una gestione oculata dei thread e della memoria per le code delle richieste in sospeso.

# Chapter 4

## Programmazione Asincrona: CompletableFuture

### 4.1 Evoluzione dai Future a CompletableFuture

L’interfaccia `Future<V>` rappresenta un’astrazione per un risultato che sarà disponibile in un secondo momento. Sebbene permetta di avviare task asincroni tramite un `ExecutorService`, essa presenta un limite strutturale critico: per recuperare il valore è necessario invocare il metodo `get()`, che è di natura **bloccante**. Se il risultato non è ancora pronto, il thread chiamante viene sospeso dal sistema operativo, impedendogli di svolgere qualsiasi altro lavoro utile. Questo fenomeno è particolarmente dannoso nei sistemi distribuiti, dove le latenze di rete sono imprevedibili e il blocco di un thread può portare rapidamente all’esaurimento dei thread pool disponibili.

`CompletableFuture` risolve questo problema implementando l’interfaccia `CompletionStage`. Questa evoluzione permette di definire callback e pipeline di esecuzione che reagiscono al completamento dei task in modo reattivo: invece di *attendere* il dato, si definisce *cosa fare* non appena il dato sarà disponibile, mantenendo il sistema fluido e non bloccante.

### 4.2 Dettaglio Tecnico dei Metodi

La classe `CompletableFuture` fornisce un insieme completo di strumenti per gestire l’intero ciclo di vita di un’attività asincrona. I metodi possono essere categorizzati in base alla loro funzione: avvio dell’attività, trasformazione dei risultati, composizione di più future, coordinamento e recupero finale.

#### 4.2.1 1. Avvio dell’Esecuzione Asincrona

Il punto di ingresso per la programmazione asincrona con questa classe è il metodo statico che permette di sottomettere un task a un pool di thread.

- `supplyAsync(Supplier<U>)`:

È un metodo statico che accetta in ingresso un `Supplier<U>` (un’interfaccia funzionale che non prende argomenti e restituisce un valore di tipo `U`). Il metodo

avvia l'esecuzione del metodo `get()` del Supplier in un thread separato (tipicamente nel `ForkJoinPool.commonPool()`) e restituisce immediatamente un nuovo `CompletableFuture<U>`. Questo oggetto fungerà da handle per il risultato futuro dell'operazione, ovvero un riferimento che non contiene ancora il dato finale, ma che permette al thread chiamante di monitorare lo stato dell'attività, attenderne il completamento o registrare azioni da eseguire automaticamente alla disponibilità del risultato.

```

1 // Avvia un calcolo asincrono che restituisce un prezzo
2 var getPrice = CompletableFuture.supplyAsync(() -> p.computePrice("pen"))
   ;

```

### 4.2.2 2. Pipeline di Trasformazione (Callback)

Questi metodi permettono di elaborare il risultato di un task non appena questo diventa disponibile, senza bloccare il thread principale. Costituiscono la base delle pipeline reattive.

- **`thenApply(Function<T,U>):`**

Viene utilizzato per applicare una trasformazione sincrona al risultato di un `CompletableFuture`. Prende in ingresso una funzione che accetta il risultato del task precedente (di tipo T) e produce un nuovo valore (di tipo U). Restituisce un nuovo `CompletableFuture<U>` contenente il risultato trasformato.

```

1 // Trasforma il risultato (Response) in Stringa appena disponibile
2 CompletableFuture<String> textFuture =
3     future.thenApply(res -> res.toString());

```

- **`thenAccept(Consumer<T>):`**

Viene utilizzato per consumare il risultato finale senza produrre un nuovo valore di ritorno per step successivi. Prende in ingresso un `Consumer` e restituisce un `CompletableFuture<Void>`. È tipicamente usato alla fine di una catena di operazioni per effetti collaterali (es. logging, aggiornamento GUI).

```

1 // Stampa il risultato appena disponibile
2 future.thenAccept(res ->
3     System.out.println("Meteo ricevuto: " + res));

```

### 4.2.3 3. Composizione e Combinazione

Questi metodi gestiscono scenari complessi in cui più operazioni asincrone devono interagire tra loro.

- **`thenCompose(Function<T, CompletionStage<U> >):`**

Fondamentale per concatenare due task asincroni dipendenti (dove l'output del primo serve come input del secondo). A differenza di `thenApply`, la funzione passata restituisce a sua volta un `CompletableFuture`.

`thenCompose` si occupa di "appiattire" il risultato, evitando di ottenere un tipo annidato come `CompletableFuture<CompletableFuture<U>>` e restituendo direttamente `CompletableFuture<U>`.

```

1 // Concatenazione: ottieni utente -> poi ottieni ordini dell'utente
2 CompletableFuture<Orders> orders =
3     userFuture.thenCompose(user ->
4         service.getOrdersAsync(user));

```

Si nota che *userFuture* è un completable future il cui esito è atteso e senza il quale non potrebbe essere eseguita la chiamata a *getOrdersAsync()* poichè si necessita del parametro di input.

- **thenCombine(CompletionStage<U>, BiFunction<T,U,V>):**

Utilizzato per sincronizzare due computazioni indipendenti che possono essere eseguite in parallelo. Accetta un secondo `CompletableFuture` e una funzione combinatrice (`BiFunction`). La funzione viene eseguita solo quando **entrambi** i Future sono completati, producendo un risultato congiunto.

```

1 // Attende sia il prezzo che lo sconto, poi calcola il totale
2 priceFuture.thenCombine(
3     discountFuture,
4     (price, disc) -> price - disc
5 );

```

Anche qui *discountFuture* è un completable future il cui esito è ancora atteso.

#### 4.2.4 4. Controllo Manuale e Stato

Strumenti per monitorare o intervenire manualmente sul ciclo di vita del Future.

- **isDone():**

Restituisce un valore booleano (`true`) se il task è entrato in uno stato terminale, che può corrispondere a un completamento con successo, un completamento eccezionale o una cancellazione.

```

1 if (future.isDone()) {
2     // Il risultato può essere letto immediatamente senza blocchi
3 }

```

- **complete(T value):**

Permette di forzare manualmente il completamento di un `CompletableFuture` con un valore specifico. Se il Future è già stato completato (da un altro thread o dal normale flusso di esecuzione), questa chiamata viene ignorata (ritorna `false`). È utile per fornire valori di default o gestire scenari di fallback.

```

1 // Forza il completamento con un valore di fallback se i dati tardano
2 future.complete(new Response("Dati non disponibili"));

```

#### 4.2.5 5. Coordinamento Multiplo

Metodi statici per gestire collezioni di Future.

- **allOf(CompletableFuture<?>... cfs):**

Crea una barriera di sincronizzazione. Restituisce un `CompletableFuture<Void>`

che si completa solo quando **tutti** i Future passati come argomento sono terminati. Non aggrega i risultati singoli, ma serve a garantire che tutte le operazioni siano concluse.

```

1 CompletableFuture<Void> allTasks =
2     CompletableFuture.allOf(f1, f2, f3);
3
4 allTasks.thenRun(() ->
5     System.out.println("Elaborazione collettiva conclusa"));

```

- **anyOf(CompletableFuture<?>... cfs):**

Restituisce un `CompletableFuture<Object>` che si completa non appena il **primo** tra i Future passati termina (sia con successo che con eccezione). È utile per scenari speculativi o di ridondanza.

```

1 // Procede con il risultato del primo server che risponde
2 CompletableFuture<Object> firstResponder =
3     CompletableFuture.anyOf(s1, s2);

```

#### 4.2.6 6. Recupero del Risultato (Blocking)

Sebbene l'obiettivo sia evitare i blocchi, è talvolta necessario estrarre il valore finale per uscire dal contesto asincrono.

- **get():**

Metodo ereditato dall'interfaccia `Future`. È bloccante e costringe a gestire le eccezioni controllate (`Checked Exceptions`), rendendolo verboso. Può lanciare `InterruptedException` o `ExecutionException`.

```

1 try {
2     Response res = future.get();
3 } catch (InterruptedException | ExecutionException e) {
4     // Gestione obbligatoria dell'errore
5 }

```

- **join():**

Alternativa moderna a `get()`, progettata per l'uso con le Lambda Expression e gli Stream. Sebbene sia anch'esso bloccante, lancia eccezioni non controllate (`Runtime Exceptions`), rendendo il codice più pulito e conciso.

```

1 // Recupero del valore senza try-catch esplicito
2 Response res = future.join();

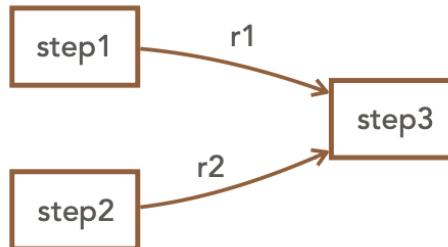
```

### 4.3 Gestione dei Workflow: Decomposizione e Coordinamento a Step

Nello sviluppo di sistemi distribuiti ad alte prestazioni, un'operazione complessa non viene quasi mai trattata come un'unità monolitica. Al contrario, essa viene decomposta in una serie di attività elementari (step) per massimizzare l'occupazione delle risorse hardware e minimizzare i tempi morti.

### 4.3.1 Analisi Architetturale della Struttura a Tre Step

La scomposizione di un workflow richiede un'analisi attenta delle dipendenze dei dati. Consideriamo il modello tipico a tre fasi illustrato nel diagramma:



**Figure 4.1:** Workflow asincrono multi-livello: r1 e r2 rappresentano i flussi di dati (Future) che alimentano lo step di aggregazione finale.

- **Step 1 e Step 2 (Parallelismo Fork):** Rappresentano attività computazionalmente indipendenti o chiamate I/O verso servizi diversi (ad esempio, il recupero di un prezzo da un database e il calcolo di uno sconto da un servizio esterno). Invece di eseguirli in sequenza, raddoppiando il tempo di attesa, vengono avviati simultaneamente tramite `supplyAsync`. Questo approccio "fork" permette di ridurre la latenza totale al tempo di esecuzione del task più lento tra i due, anziché alla loro somma.
- **Step 3 (Sincronizzazione Join):** Questo step funge da barriera di sincronizzazione (Join). In un sistema distribuito, non possiamo prevedere quale tra lo Step 1 e lo Step 2 terminerà per primo a causa del jitter di rete. Lo Step 3 è progettato per rimanere in stato di attesa logica (non bloccante): esso possiede la logica di business per combinare i risultati intermedi **r1** e **r2**, ma la sua esecuzione è vincolata alla disponibilità di entrambi i dati.

## 4.4 Strategie di Composizione: Analisi Comparativa

Il modo in cui gestiamo l'attesa dello Step 3 determina drasticamente la scalabilità del nostro nodo nel cluster.

1. **Approccio Bloccante (Attesa Attiva/Sincrona):** In questa configurazione, il thread chiamante avvia i task paralleli e poi invoca il metodo `join()` o `get()` su ciascun Future. Dal punto di vista ingegneristico, questa è una soluzione inefficiente: il thread rimane occupato in memoria (occupando stack e risorse del kernel) pur non producendo alcun calcolo. Se molte richieste seguono questo pattern, il thread pool si esaurisce rapidamente, portando al collasso del servizio anche se la CPU è teoricamente scarica.
2. **Composizione Non Bloccante (Inversione del Controllo):** Sfruttando i metodi come `thenCombine`, lo Step 3 viene registrato come una *callback*. In questo scenario, il thread principale "delega" l'esecuzione futura allo scheduler di Java e torna immediatamente libero di gestire nuove richieste in arrivo. Quando l'ultimo dei due task precedenti completa, sarà lo stesso thread del pool a farsi carico

dell'esecuzione dello Step 3. Questo garantisce che il sistema sia sempre reattivo (*responsive*).

## 4.5 Sintesi del Pattern Request Pipeline

L'infrastruttura fornita dai `CompletableFuture` non è fine a se stessa, ma costituisce il pilastro tecnologico per l'implementazione del pattern **Request Pipeline**. L'obiettivo è trasformare un canale di comunicazione da un modello "stop-and-wait" a un flusso continuo ad alto throughput.

I concetti chiave del pattern sono:

- **Asincronia di Invio:** Il client non sospende mai l'esecuzione in attesa di un ACK dal server, permettendo di saturare la banda disponibile con un flusso costante di richieste.
- **Disaccoppiamento dei Thread:** Si implementano thread dedicati alla scrittura sul socket e thread dedicati alla lettura delle risposte, evitando che un ritardo in una fase blocchi l'altra.
- **Gestione delle Richieste Inflight:** Per evitare di saturare la memoria del ricevente o la coda del server (fenomeno del *flooding*), si introduce un limite al numero di richieste inviate ma non ancora confermate. Questo meccanismo di **backpressure** garantisce la stabilità del sistema: il mittente si "autolimita" in base alla velocità effettiva di elaborazione del destinatario, riprendendo l'invio solo quando una risposta libera uno slot nella pipeline.

# Chapter 5

## Design Pattern per la Resilienza e la Stabilità

### 5.1 Il Concetto di Stabilità nei Sistemi Distribuiti

Per comprendere la resilienza, è necessario definire l'unità di lavoro fondamentale di un sistema distribuito: la **transazione**. Una transazione non è semplicemente una query, ma un'unità logica di lavoro che il sistema deve elaborare. Essa può comprendere molteplici aggiornamenti del database e integrazioni complesse con sistemi esterni, come ad esempio il processo in cui un cliente effettua un ordine che include la verifica remota della carta di credito.

Un sistema è un insieme di componenti hardware (host e rete), applicazioni e servizi che supportano queste transazioni. Un sistema si definisce **resiliente** quando continua a elaborare transazioni anche in presenza di guasti nei componenti che lo costituiscono. La stabilità, pertanto, non è una metrica puramente tecnica (server *up* o *down*), bensì funzionale: l'obiettivo è garantire che l'utente finale possa portare a termine le proprie operazioni nonostante le perturbazioni interne.

### 5.2 Timeout

L'interazione con la rete introduce inevitabilmente il rischio che un componente remoto smetta di rispondere o che un dispositivo di rete fallisca improvvisamente. In questo contesto, il codice applicativo non può permettersi di attendere indefinitamente una risposta che potrebbe non arrivare mai; prima o poi deve rinunciare.

#### 5.2.1 Meccanismo e Implementazione

Il **Timeout** è il meccanismo primario per l'isolamento dei guasti. Esso consente di interrompere l'attesa quando si sospetta un malfunzionamento, impedendo che un problema in un sottosistema si propaghi all'intera applicazione (effetto a cascata).

L'applicazione rigorosa dei timeout è critica non solo per le chiamate di rete, ma anche per la gestione delle risorse interne:

- **Problema delle API nascoste:** Molte librerie di alto livello utilizzano socket internamente ma nascondono i dettagli implementativi, spesso non esponendo configurazioni di timeout, il che rappresenta un rischio.
- **Pool di risorse (Thread Block):** Un pool (ad esempio di connessioni a un database) che si esaurisce blocca i thread richiedenti finché una risorsa non si libera. È quindi obbligatorio imporre un timeout su questa attesa per assicurare che i thread vengano sbloccati sia in caso di successo sia in caso di fallimento, evitando il blocco indefinito dell'applicazione.
- **Primitive Java:** Quando si utilizzano meccanismi di concorrenza, è necessario preferire le versioni con timeout:
  - Per `Object`: usare `wait(long timeout)`.
  - Per `BlockingQueue`: usare `poll()` e `offer()` con parametri temporali.
  - Per `Lock`: usare `tryLock()` con timeout.

### 5.2.2 Pattern Gateway e QueryObject

L'introduzione dei timeout rischia di inquinare la logica di business con codice ripetitivo dedicato alla gestione degli errori (*error-handling*). Per mitigare questo problema, è opportuno organizzare le operazioni in un insieme di primitive riutilizzabili.

Ad esempio, un'interazione con il database composta da acquisizione della connessione, esecuzione della query e gestione dei risultati può essere incapsulata in un **QueryObject** o gestita tramite un **Gateway** generico. Questo approccio centralizza la gestione dei timeout e facilita l'applicazione di pattern più complessi come il Circuit Breaker.

### 5.2.3 Strategie di Ritentativo (Retry)

Quando un'operazione fallisce a causa di un timeout, riprovare immediatamente è spesso la strategia sbagliata: se il problema è sistematico o dovuto a congestione, un retry immediato fallirà nuovamente e aggraverà il carico.

- Se il problema è transitorio, è preferibile attendere prima di riprovare.
- La strategia consigliata è accodare l'operazione e riprovare in un secondo momento (*delayed retry*), permettendo al sistema di recuperare.

## 5.3 Circuit Breaker

Ispirato al fusibile elettrico che previene incendi dovuti al surriscaldamento ( $I^2R$ ), il **Circuit Breaker** è un componente architettonico che protegge il sistema evitando l'invio di richieste verso sottosistemi che sono già in stato di errore.

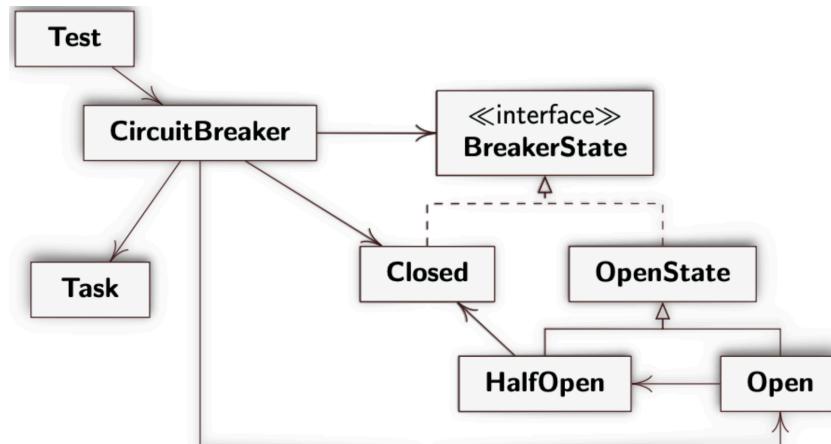
La differenza fondamentale rispetto al pattern *Retry* risiede nell'obiettivo: mentre il *Retry* mira a superare problemi transitori (spesso aggravando il carico), il Circuit Breaker mira a proteggere le risorse. Bloccando preventivamente le operazioni verso un servizio degradato, si ottengono due benefici:

- **Protezione del Server:** Si concede al sottosistema in difficoltà il tempo necessario per recuperare, senza essere bombardato da continue richieste.
- **Protezione del Client (Resource Conservation):** Si evita che i thread del client rimangano bloccati in attesa di timeout TCP/HTTP su un servizio morto. Questo previene l'esaurimento del Thread Pool del client e il conseguente blocco dell'intera applicazione ("Cascading Failure").

### 5.3.1 Modellazione e Implementazione: Il State Pattern

La natura del Circuit Breaker è intrinsecamente quella di una macchina a stati finiti. Dal punto di vista ingegneristico, l'approccio migliore per implementare questa logica evitando complessi blocchi condizionali (come lunghe catene di `if-else` nidificati) è l'utilizzo del **State Design Pattern**.

Come mostrato nel diagramma delle classi seguente, il comportamento varia dinamicamente in base allo stato interno dell'oggetto, sfruttando il polimorfismo per alterare la risposta del sistema alle richieste:



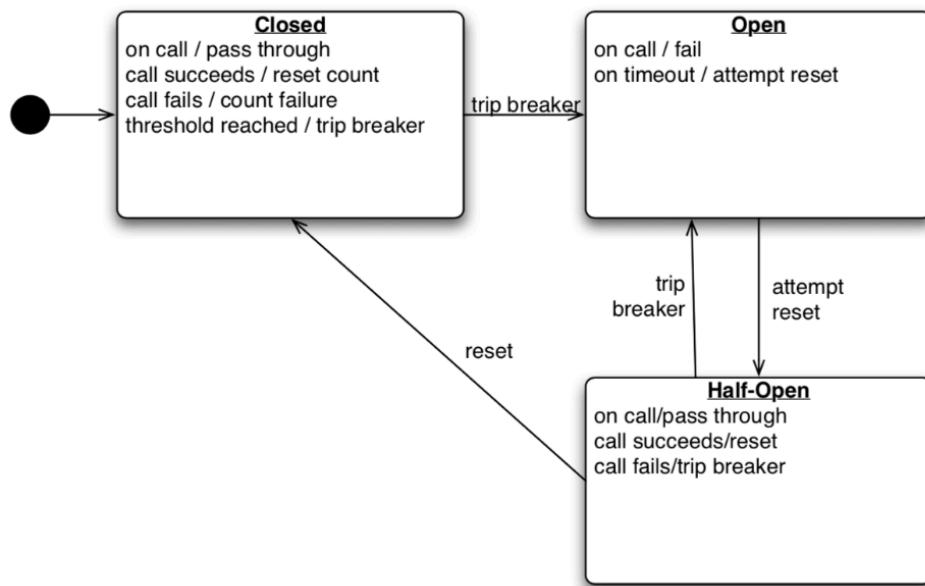
**Figure 5.1:** Diagramma UML delle classi per l'implementazione del Circuit Breaker tramite State Pattern. La classe **CircuitBreaker** funge da Context, mentre **BreakerState** definisce l'interfaccia comune per gli stati concreti.

L'architettura si compone di tre elementi chiave:

- **Context (CircuitBreaker):** È la classe "involutro" utilizzata dai client. Essa non implementa la logica di transizione, ma mantiene un riferimento all'istanza dello stato corrente (**BreakerState**) e delega ad esso l'esecuzione di ogni richiesta (es. `state.executeRequest()`).
- **State Interface (BreakerState):** Definisce il contratto che ogni stato deve rispettare, tipicamente includendo metodi per gestire l'esecuzione della logica di business e metodi per notificare successi o fallimenti.
- **Concrete States:** Le classi **Closed**, **Open** e **HalfOpen** incapsulano la logica specifica. Ad esempio, solo la classe **Closed** eseguirà effettivamente la chiamata remota, mentre **Open** solleverà un'eccezione immediata.

### 5.3.2 Dinamica degli Stati e Transizioni

Il comportamento del sistema transita attraverso tre stati logici ben definiti. La correttezza di queste transizioni determina la resilienza del sistema.



**Figure 5.2:** Diagramma di stato (State Machine): logica delle transizioni basata su eventi di successo, fallimento e timeout.

Analizziamo la logica operativa encapsulata in ciascuno stato:

1. **Closed (Chiuso - Operatività Normale):** È lo stato iniziale. Qui il Circuit Breaker agisce come un pass-through trasparente.
  - **Logica:** Ogni richiesta viene inoltrata al servizio esterno.
  - **Monitoraggio:** Il componente monitora l'esito delle chiamate. In caso di successo, resetta eventuali contatori di errore.
  - **Transizione:** In caso di fallimento (eccezione o timeout), incrementa un contatore interno. Se il numero di errori (o la percentuale, in implementazioni più avanzate come le *sliding windows*) supera una soglia predefinita (*threshold*), l'oggetto innesca la transizione verso lo stato **Open** ("Trip the breaker").
2. **Open (Aperto - Blocco di Sicurezza):** Rappresenta lo stato di protezione attiva.
  - **Fail Fast:** L'oggetto intercetta ogni chiamata e fallisce *immediatamente*, lanciando un'eccezione specifica (es. `CircuitBreakerOpenException`) senza tentare alcuna connessione di rete. Questo libera istantaneamente il thread del client.
  - **Timer:** Al momento dell'ingresso in questo stato, viene avviato un timer interno (es. 60 secondi).
  - **Transizione:** Allo scadere del timeout, lo stato transita automaticamente verso **Half-Open** per verificare se il sistema remoto è tornato disponibile.

3. **Half-Open (Semi-Aperto - Sonda)**: È lo stato più delicato, fungendo da "canarino". Il sistema deve verificare la disponibilità del servizio senza rischiare di sovraccaricarlo nuovamente.

- **Logica "Sonda"**: Permette il passaggio di una *singola* richiesta (o un numero molto limitato) verso il sistema critico, mentre tutte le altre richieste concorrenti continuano a fallire (comportamento Open).
- **Transizione su Successo**: Se la richiesta sonda ha esito positivo, si assume che il problema sia risolto. Il Circuit Breaker torna allo stato **Closed** e azzera i contatori.
- **Transizione su Fallimento**: Se la richiesta sonda fallisce, si deduce che il guasto persiste. L'interruttore scatta immediatamente indietro allo stato **Open**, riavviando il timer di attesa (spesso con un backoff esponenziale per aumentare il tempo di riposo).

L'adozione di questa struttura permette di degradare le funzionalità in modo controllato (es. disabilitando temporaneamente feature non essenziali o restituendo dati in cache) e garantisce che ogni transizione venga tracciata nei log per il monitoraggio operativo.

## 5.4 Bulkheads (Paratie)

Il pattern **Bulkhead** trae ispirazione dalle paratie navali, che suddividono una nave in compartimenti stagni per impedire che una falla in una sezione comprometta l'intera imbarcazione. L'obiettivo ingegneristico è il **contenimento del danno**: partizionare il sistema affinché il guasto di una parte non distrugga il tutto.

### 5.4.1 Strategie di Partizionamento e Granularità

Il partizionamento può avvenire a diversi livelli di granularità:

- **Ridondanza Fisica (Cluster)**: L'utilizzo di server indipendenti garantisce che il guasto hardware di una macchina non coinvolga le altre.
- **Separazione per Funzionalità (Farm dedicate)**: In sistemi su larga scala, è possibile dedicare pool di server specifici per funzionalità *mission-critical*, separandoli da servizi ad alto traffico.
- **Virtualizzazione**: La virtualizzazione consente di condividere l'hardware fisico mantenendo l'isolamento logico tra i server. Inoltre, permette di variare dinamicamente la capacità avviando o migrando macchine virtuali.
- **Livello Processo/Thread (CPU Binding)**: Tecniche come il *CPU binding* impediscono che un singolo processo consumi tutte le risorse computazionali disponibili. Anche la separazione dei thread pool rappresenta una forma efficace di paratia.



**Università  
di Catania**

**UNIVERSITY OF CATANIA**

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

A MASTER OF SCIENCE DEGREE IN COMPUTER SCIENCE.

---

*Alfio Spoto*

Ingegneria dei sistemi distribuiti

---

APPUNTI LEZIONI ISD  
IV

---

---

Academic Year 2024 - 2025

# Contents

<b>1</b>	<b>Sistemi di Messaggistica e RabbitMQ</b>	<b>1</b>
1.1	Introduzione alla Messaggistica . . . . .	1
1.1.1	Vantaggi dell'approccio a messaggi . . . . .	1
1.1.2	Struttura del Messaggio . . . . .	1
1.2	RabbitMQ e il protocollo AMQP . . . . .	2
1.2.1	Canali vs Connessioni . . . . .	2
1.3	Scenari e Pattern Implementativi . . . . .	2
1.3.1	Esempio 1: Coda Semplice (Hello World) . . . . .	3
1.3.2	Esempio 2: Work Queues (Competing Consumers) . . . . .	4
1.3.2.1	Gestione dell’Affidabilità (Ack) e QoS . . . . .	4
1.3.3	Esempio 3: Publish/Subscribe (Fanout) . . . . .	5
1.3.4	Esempio 4: Routing (Direct Exchange) . . . . .	6
1.3.5	Esempio 5: Topics (Pattern Matching) . . . . .	7
1.3.6	Esempio 6: RPC (Remote Procedure Call) . . . . .	7
1.4	Gestione Avanzata e Affidabilità . . . . .	8
1.4.1	Operazioni sulle Code . . . . .	8
1.4.2	Pull API e TTL . . . . .	9
1.4.3	Recupero Automatico . . . . .	9
<b>2</b>	<b>Aspect Oriented Programming (AOP)</b>	<b>10</b>
2.1	Introduzione e Motivazioni . . . . .	10
2.1.1	Limiti dell’OOP: Scattering e Tangling . . . . .	10
2.1.2	La Soluzione AOP : Crosscutting Concerns . . . . .	10
2.2	Concetti Fondamentali . . . . .	11
2.2.1	Esempio Introduttivo: Il Logger . . . . .	11
2.3	Dettaglio dei Costrutti AOP . . . . .	11
2.3.1	Tipi di Pointcut: Call vs Execution . . . . .	11
2.3.2	Il Weaver . . . . .	12
2.4	Esempio Avanzato: Protection Proxy . . . . .	13
2.4.1	Collezione del Contesto . . . . .	13
2.5	Caso Studio: Editor di Figure . . . . .	13
2.5.1	Il Problema del ”Display Updating” . . . . .	14
2.5.2	Soluzione con AspectJ . . . . .	14
2.6	Composizione dei Pointcut . . . . .	15
2.7	Cattura del Contesto (Context Exposure) . . . . .	15
2.7.1	1. Target: L’Oggetto Ricevente . . . . .	15
2.7.2	2. Args: I Parametri del Metodo . . . . .	15

2.7.3 3. This: L’Oggetto ”Attivo” . . . . .	16
2.7.3.1 Differenza cruciale tra This e Target . . . . .	16
2.8 Tipologie Avanzate di Advice: Around . . . . .	16
2.9 Wildcard e Pattern Matching . . . . .	17
2.10 Accesso Riflessivo al Join Point . . . . .	18
2.11 Static Crosscutting . . . . .	19
2.11.1 Inter-type Declarations (ITD) . . . . .	19
2.11.2 Policy Enforcement (Declare Error/Warning) . . . . .	20
2.12 Caso Studio: Adapter Pattern con AOP . . . . .	20
2.12.1 Approccio Classico (Object-Oriented) . . . . .	20
2.12.2 Approccio AOP (Aspect Adapter) . . . . .	20
2.13 Pointcut per Campi . . . . .	21

# Chapter 1

## Sistemi di Messaggistica e RabbitMQ

### 1.1 Introduzione alla Messaggistica

Nei sistemi distribuiti moderni, componenti software eterogenei o intere applicazioni devono comunicare attraverso la rete. Tuttavia, la rete è un mezzo intrinsecamente inaffidabile: i componenti possono subire ritardi, interruzioni impreviste e le velocità di trasmissione sono nettamente inferiori rispetto a una comunicazione locale in memoria.

La **messaggistica** (messaging) è la tecnologia che abilita comunicazioni **asincrone** e **affidabili** tra programmi diversi, fungendo da strato di integrazione che astrae le differenze di linguaggi di programmazione, piattaforme e formati dati.

#### 1.1.1 Vantaggi dell'approccio a messaggi

L'adozione di un middleware di messaggistica offre benefici architetturali significativi:

- **Disaccoppiamento (Send and Forget):** La comunicazione è asincrona. Il mittente (Produttore) non deve attendere che il destinatario riceva o elabori il messaggio, né deve attendere che il sistema sottostante completi l'invio. Una volta affidato il messaggio al broker, il mittente può proseguire con altre attività.
- **Integrazione:** Il sistema agisce come un "traduttore universale", permettendo lo scambio di dati (serializzati) tra applicazioni scritte in linguaggi diversi (es. Java e Python) su sistemi operativi diversi.
- **Gestione del Carico (Throttling):** Nelle chiamate remote sincrone, un picco di richieste può mandare in crash il ricevente. La messaggistica permette al ricevente di consumare le richieste alla propria velocità, livellando i picchi di carico.
- **Affidabilità:** Viene utilizzato un approccio *store and forward*. Se la rete o il destinatario falliscono, il sistema di messaggistica memorizza il dato e tenta la ritrasmissione finché non ha successo.

#### 1.1.2 Struttura del Messaggio

Un messaggio è l'unità atomica di comunicazione ed è composto da due parti:

1. **Header:** Contiene i metadati, come le informazioni sul mittente e le istruzioni di instradamento (dove deve andare il messaggio).
2. **Body (Payload):** Contiene il dato effettivo trasmesso (stringa, array di byte, oggetto serializzato). Il broker tratta il payload come opaco.

## 1.2 RabbitMQ e il protocollo AMQP

**RabbitMQ** è un *message broker* open-source che implementa il protocollo standard **AMQP** (Advanced Message Queuing Protocol). Il suo ruolo è accettare, immagazzinare e inoltrare messaggi.

Per poter inviare un messaggio si interagisce con tre entità principali:

- **Scambi (Exchange):** Sono i punti dove i produttori pubblicano i messaggi.
- **Coda (Queue):** Buffer che conservano i messaggi in attesa di essere consumati. I consumatori ricevono messaggi da una coda a cui si sono iscritti.
- **Connessioni (Binding):** Regole che permettono ai messaggi di viaggiare da uno scambio a una specifica coda.

La logica di consegna segue queste regole:

- Se una coda ha consumatori iscritti, i messaggi vengono inviati immediatamente (push).
- Se non ci sono iscritti, il messaggio attende in coda finché un consumatore non si connette.
- La comunicazione è unidirezionale (*fire-and-forget*): il mittente non attende risposta immediata.

### 1.2.1 Canali vs Connessioni

Le applicazioni comunicano con RabbitMQ tramite connessioni TCP. Tuttavia, aprire una connessione TCP è un'operazione costosa per il sistema operativo. AMQP introduce il concetto di **Canale AMQP** (*Channel AMQP*): una connessione virtuale multiplexata all'interno della connessione TCP.

- I canali sono leggeri e veloci da creare.
- Ogni canale ha un ID univoco.
- La maggior parte delle operazioni (pubblicazione, sottoscrizione) avviene su un canale specifico.

## 1.3 Scenari e Pattern Implementativi

RabbitMQ supporta diversi pattern di messaggistica, dalla semplice consegna punto-punto a complessi schemi di routing. Analizziamo i 6 scenari principali.

### 1.3.1 Esempio 1: Coda Semplice (Hello World)

Nello scenario più basilare, un produttore invia un messaggio e un consumatore lo riceve. Qui si utilizza lo *scambio di default* (identificato da una stringa vuota ""), che instrada il messaggio direttamente alla coda specificata come routing key.



**Figure 1.1:** Scenario base: Il produttore invia direttamente alla coda e il consumatore riceve.

Il Produttore crea la connessione, dichiara la coda e pubblica il messaggio:

```

1 ConnectionFactory factory = new ConnectionFactory();
2 factory.setHost("localhost"); // nome della macchina del broker
3 Connection connection = factory.newConnection();
4 Channel channel = connection.createChannel();

5
6 String QUEUE_NAME = "queue-hello";
7 // queueDeclare: nome, durable, exclusive, autoDelete, args
8 channel.queueDeclare(QUEUE_NAME, false, false, false, null);
9 // basicPublish: exchange, routingKey, props, body
10 channel.basicPublish("", QUEUE_NAME, null, "hello".getBytes());

```

- queueDeclare() crea la coda con i parametri: queue, durable (sopravvive se si fa ripartire il server), exclusive (per questa connessione soltanto), autoDelete (cancelabile dal server se non usata).
- basicPublish() pubblica un messaggio, ha i parametri: exchange, routingKey, props, body.

Il Consumatore utilizza il pattern **Event-Driven**. Anziché controllare ciclicamente se ci sono messaggi (polling), registra una callback che viene invocata automaticamente dal middleware quando arriva un nuovo messaggio.

```

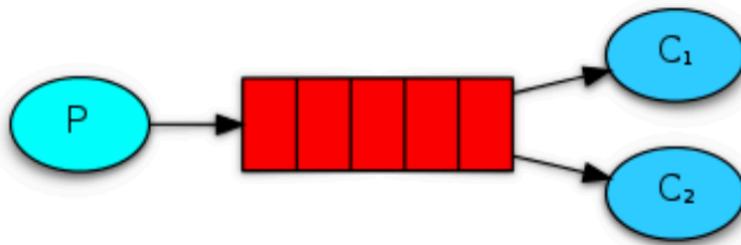
1 Consumer consumer = new DefaultConsumer(channel) {
2     @Override
3     public void handleDelivery(String consumerTag, Envelope envelope,
4                                 AMQP.BasicProperties properties, byte[] body)
5         throws IOException {
6             String message = new String(body, "UTF-8");
7             System.out.println("Ricevuto " + message + "'");
8         }
9     };
10 // basicConsume: queue, autoAck, callback
11 channel.basicConsume(QUEUE_NAME, true, consumer);

```

- `basicConsume()` registra un consumatore alla coda, ha i parametri: `queue`, `autoAck`, `callback`.
- `handleDelivery()` è il metodo di callback, previsto nell'interfaccia `Consumer` ed implementato dallo specifico consumatore.

### 1.3.2 Esempio 2: Work Queues (Competing Consumers)

Quando l'elaborazione di un messaggio è onerosa (es. resizing di immagini, rendering PDF), un singolo consumatore non basta. Si utilizza il pattern **Competing Consumers**: una singola coda è condivisa tra più consumatori per parallelizzare il lavoro.



**Figure 1.2:** Work Queues: I messaggi vengono distribuiti tra più consumatori per bilanciare il carico.

RabbitMQ distribuisce i messaggi ai consumatori in modalità **Round-Robin** (uno a testa).

```

1 Consumer consumer1 = new DefaultConsumer(channel) {
2     @Override
3     public void handleDelivery(String consumerTag, Envelope envelope,
4                                 AMQP.BasicProperties properties, byte[] body)
5     throws IOException {
6         String message = new String(body, "UTF-8");
7         System.out.println("Ricevente uno '" + message + "'");
8     }
9 };
10 channel.basicConsume(QUEUE_NAME, true, consumer1);
11
12 Consumer consumer2 = new DefaultConsumer(channel) {
13     @Override
14     public void handleDelivery(...) { // codice analogo
15         System.out.println("Ricevente due '" + message + "'");
16     }
17 };
18 channel.basicConsume(QUEUE_NAME, true, consumer2);

```

#### 1.3.2.1 Gestione dell'Affidabilità (Ack) e QoS

In uno scenario reale, è necessario gestire il caso in cui un consumatore fallisca durante l'elaborazione o sia più lento degli altri.

- **Message Acknowledgment (Ack)**: Per evitare la perdita di dati, si disabilita l'auto-ack (secondo parametro di `basicConsume` a `false`). Il consumatore invierà un `basicAck` manuale solo dopo aver completato il lavoro. Se il consumatore cade senza aver inviato l'ack, RabbitMQ rimette il messaggio in coda.
- **Fair Dispatch (QoS)**: Per evitare che un consumatore veloce rimanga senza lavoro mentre uno lento è pieno di messaggi non ancora processati, si usa `basicQos(1)`. Questo istruisce RabbitMQ a non inviare un nuovo messaggio a un worker finché questo non ha inviato l'Ack per il precedente.

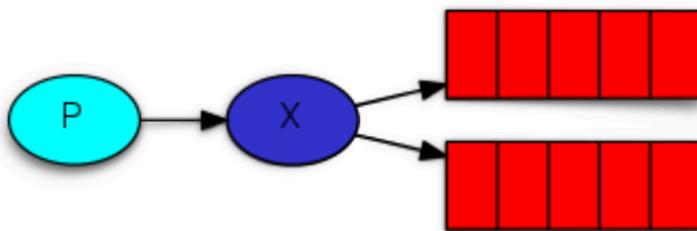
```

1 channel.basicQos(1); // Accetta solo 1 messaggio non confermato alla volta
2 channel.basicConsume(QUEUE_NAME, false, consumer); // AutoAck false
3
4 // Dentro handleDelivery, dopo il lavoro:
5 channel.basicAck(envelope.getDeliveryTag(), false);

```

### 1.3.3 Esempio 3: Publish/Subscribe (Fanout)

In questo pattern, l'obiettivo è il *broadcasting*: inviare lo stesso messaggio a tutti i consumatori interessati (es. un sistema di logging). Si utilizza uno scambio di tipo **Fanout**, che ignora le routing key e duplica il messaggio verso tutte le code connesse.



**Figure 1.3:** Publish/Subscribe: Lo scambio Fanout duplica il messaggio verso tutte le code connesse.

Spesso i consumatori utilizzano code temporanee (non durevoli, esclusive, auto-delete) il cui nome è generato dal server, poiché sono interessati solo al flusso in tempo reale.

```

1 // Produttore
2 String EXCHANGE_NAME = "logs";
3 channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
4 channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));

```

Il produttore non invia i messaggi direttamente alla coda, invece li invia ad uno scambio (exchange). Uno scambio da un lato riceve messaggi dai produttori e dall'altro lato li manda alle code. Nei precedenti esempi si è usato uno scambio di default, identificato tramite la stringa vuota "", e i messaggi sono stati inviati alla coda indicata dal secondo parametro, se esiste :

```
1 channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
```

Lo scambio sa cosa fare con un messaggio ricevuto tramite regole definite per il tipo di scambio. Ci sono i tipi di scambio: direct, topic, headers, e fanout. Il tipo fanout manda i messaggi che riceve a tutte le code che conosce.

```

1 // Consumatore
2 String queueName = channel.queueDeclare().getQueue(); // Coda temporanea
3 channel.queueBind(queueName, EXCHANGE_NAME, ""); // Binding
4 channel.basicConsume(queueName, true, consumer);

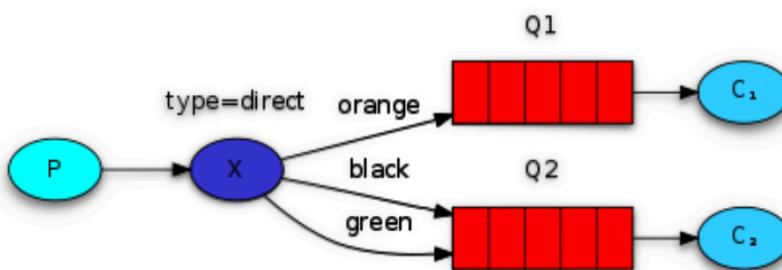
```

Il consumatore prende messaggi da una coda, quindi crea una coda vuota, non-durable, exclusive, auto-deleted,

- Il nome della coda sarà creato dal server (in modo random), non serve far conoscere il nome al produttore
- Per far sì che i messaggi dallo scambio arrivino alla coda bisogna creare una connessione (binding) fra la coda e lo scambio.
- Si registra il consumatore alla coda per fargli arrivare messaggi

### 1.3.4 Esempio 4: Routing (Direct Exchange)

Questo pattern raffina il precedente, permettendo ai consumatori di iscriversi solo a un sottoinsieme di messaggi (filtro) tramite una **Routing Key**.



**Figure 1.4:** Routing Diretto: Lo scambio inoltra il messaggio solo se la Routing Key coincide con la Binding Key.

Si utilizza uno scambio di tipo **Direct**. Un messaggio viene instradato a una coda se la Routing Key di pubblicazione coincide esattamente con la Binding Key della coda.

```

1 // Produttore: Invia con chiave "cmd"
2 channel.exchangeDeclare(EXCHANGE_NAME, BuiltInExchangeType.DIRECT);
3 channel.basicPublish(EXCHANGE_NAME, "cmd", null, msg.getBytes("UTF-8"));

```

- Per la creazione dello scambio, indicare il tipo direct (secondo parametro).
- Il produttore indica con basicPublish() una routing key (secondo parametro), in questo caso cmd è il valore della routing key.

```

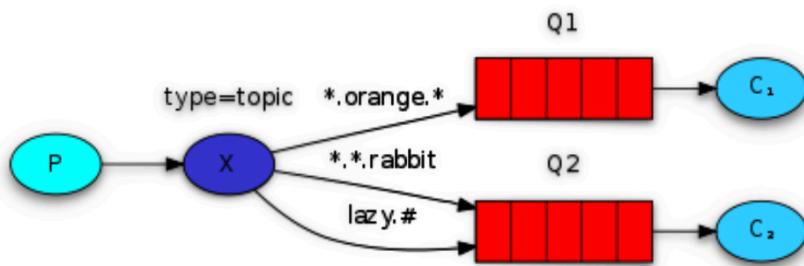
1 // Consumatore: Riceve solo messaggi "cmd"
2 String queueName = channel.queueDeclare().getQueue();
3 channel.queueBind(queueName, EXCHANGE_NAME, "cmd");
4 channel.basicConsume(queueName, true, consumer);

```

- Il consumatore crea una coda e la connette indicando la stessa routing key (terzo parametro di queueBind()).
- Quindi, come fatto in precedenza, si usa basicConsume(). Il consumatore riceverà dal canale e sulla coda indicata solo i messaggi filtrati secondo la routing key fornita.

### 1.3.5 Esempio 5: Topics (Pattern Matching)

Per sistemi complessi, il routing esatto è limitante. Il **Topic Exchange** permette il routing basato su pattern e wildcard. Le chiavi sono parole separate da punti (es. quick.orange.rabbit).



**Figure 1.5:** Topic Exchange: Routing flessibile basato su pattern e wildcard.

Caratteri jolly supportati:

- \* (asterisco): sostituisce esattamente una parola.
- # (cancelletto): sostituisce zero o più parole.

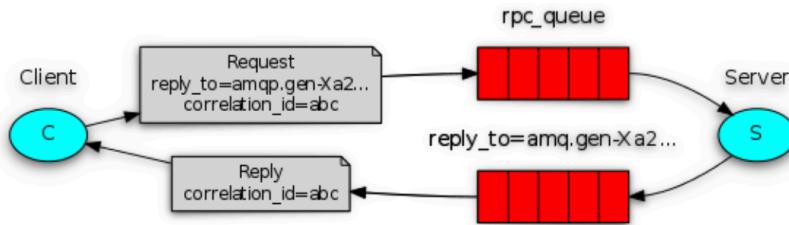
```

1 // Produttore
2 channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);
3 channel.basicPublish(EXCHANGE_NAME, "quick.orange.rabbit", null, body);
4
5 // Consumatore (riceve tutti gli animali orange)
6 channel.queueBind(queueName, EXCHANGE_NAME, "*.*.orange.*");

```

### 1.3.6 Esempio 6: RPC (Remote Procedure Call)

Sebbene la messaggistica sia asincrona, è possibile simulare chiamate sincrone (Richiesta/Risposta). Il client invia una richiesta e specifica una **Coda di Callback** su cui attendere la risposta.



**Figure 1.6:** RPC su RabbitMQ: Uso di due code e Correlation ID.

Si utilizza un **Correlation ID** per associare la risposta asincrona alla richiesta originale.

```

1 // Lato client
2 String replyQueueName = channel.queueDeclare().getQueue();
3 String corrId = UUID.randomUUID().toString();
4
5 BasicProperties props = new BasicProperties.Builder()
6     .correlationId(corrId)
7     .replyTo(replyQueueName)
8     .build();
9
10 channel.basicPublish("", reqQueueName, props, message.getBytes("UTF-8"));
11
12 // Il client ascolta sulla replyQueueName per la risposta
13 channel.basicConsume(replyQueueName, true, consumer);

```

Il server, ricevuta la richiesta, elabora il dato e invia la risposta alla coda specificata in `replyTo`, copiando il `correlationId`.

```

1 // Lato server
2 BasicProperties replyProps = new BasicProperties.Builder()
3     .correlationId(properties.getCorrelationId())
4     .build();
5
6 channel.basicPublish("", properties.getReplyTo(), replyProps, responseBody);
7 channel.basicAck(envelope.getDeliveryTag(), false);

```

## 1.4 Gestione Avanzata e Affidabilità

RabbitMQ offre strumenti per la gestione del ciclo di vita e la resilienza del sistema.

### 1.4.1 Operazioni sulle Code

È possibile gestire le code programmaticamente:

- `queuePurge("name")`: Svuota una coda.
- `queueDelete("name")`: Cancella una coda.
- `queueDelete("name", true, false)`: Cancella solo se inutilizzata (senza consumatori).

### 1.4.2 Pull API e TTL

Oltre al modello *push* (event-driven), esiste un modello *pull* sincrono tramite `basicGet`, che preleva un singolo messaggio se disponibile. Inoltre, è possibile impostare un TTL (Time To Live) sui messaggi:

```
1 channel.basicPublish(exchange, key,  
2     new BasicProperties.Builder().expiration("60000").build(), // 60s  
3     body);
```

### 1.4.3 Recupero Automatico

I client Java di RabbitMQ supportano il recupero automatico della connessione. Se si verifica un'eccezione di I/O o un timeout di rete, la libreria tenta di ristabilire la connessione e ripristinare automaticamente la topologia (code, scambi e binding creati). Ecco la prima parte del capitolo sulla Programmazione Orientata agli Aspetti (AOP), basata sui contenuti delle slide "Part 1" che hai fornito.

Ho strutturato il testo per seguire il flusso logico delle slide: dai limiti della OOP (scattering/tangling) ai concetti fondamentali (Join Point, Advice, Pointcut), fino all'esempio avanzato del Protection Proxy.

“‘latex

# Chapter 2

## Aspect Oriented Programming (AOP)

### 2.1 Introduzione e Motivazioni

Nella programmazione orientata agli oggetti (OOP), i sistemi vengono decomposti in moduli (classi) basati sulle funzionalità primarie. Tuttavia, esistono requisiti di sistema che non si allineano perfettamente con questa decomposizione verticale.

#### 2.1.1 Limiti dell'OOP: Scattering e Tangling

Consideriamo l'implementazione di funzionalità trasversali come il *logging*, la sicurezza o la gestione delle transazioni. Utilizzando solo tecniche OOP, ci scontriamo con due fenomeni problematici:

- **Code Scattering (Dispersione):** Lo stesso codice viene duplicato in molte classi diverse. Ad esempio, istruzioni come `log.writeLog(...)` appaiono ripetutamente in vari metodi di varie classi.
- **Code Tangling (Intreccio):** Una singola classe contiene codice relativo a molteplici *concern* (interessi) diversi. Una classe **Product**, che dovrebbe occuparsi solo delle informazioni sul prodotto, finisce per contenere codice di logging, violando il principio di singola responsabilità (SRP).

Questo problema è definito come **Tirannia della Decomposizione Predominante**: una volta stabilita una gerarchia di classi principale, tutto ciò che "taglia trasversalmente" questa gerarchia diventa difficile da modularizzare.

#### 2.1.2 La Soluzione AOP : Crosscutting Concerns

L'**Aspect Oriented Programming (AOP)** permette di separare questi *concern* trasversali (Crosscutting Concerns) definendo una nuova unità modulare: l'**Aspetto (Aspect)**. Un aspetto specifica quali azioni compiere (Advice) quando vengono raggiunti determinati punti del programma (Join Point), permettendo di rimuovere il codice "sporco" dalle classi di business.

## 2.2 Concetti Fondamentali

I pilastri dell'AOP sono quattro:

1. **Join Point**: Un punto ben definito nell'esecuzione del programma (es. chiamata di un metodo, accesso a un campo, gestione di un'eccezione).
2. **Pointcut**: Un costrutto che seleziona un insieme di Join Point e ne colleziona il contesto di esecuzione.
3. **Advice**: Il codice che viene eseguito quando un Join Point viene raggiunto.
4. **Aspect**: Il modulo che incapsula pointcut e advice.

### 2.2.1 Esempio Introduttivo: Il Logger

Immaginiamo di voler registrare ogni variazione di prezzo nella classe `Product` senza modificare la classe stessa.

```

1 public aspect LoggerAsp {
2     // Istanza dedicata al logging
3     private Logger l = new Logger();
4
5     // Pointcut: Intercepta le chiamate a putPrice su Product
6     pointcut loggable(): call(public void Product.putPrice(*));
7
8     // Advice: Esegue PRIMA del metodo intercettato
9     before(): loggable() {
10         l.writeLog("Price changed");
11     }
12 }
```

**Code Listing 2.1:** Definizione dell'Aspetto Logger

Grazie a questo aspetto, la classe `Product` rimane pulita ("oblivious"): non sa nemmeno che esiste un logger che la sta osservando.

## 2.3 Dettaglio dei Costrutti AOP

### 2.3.1 Tipi di Pointcut: Call vs Execution

In AspectJ, è fondamentale distinguere tra il lato chiamante e il lato chiamato. Sebbene l'effetto finale sembri spesso lo stesso (l'advice viene eseguito), la differenza sta nel **luogo fisico** in cui il codice dell'aspetto viene inserito.

Consideriamo uno scenario in cui una classe `Client` invoca un metodo di `Service`.

- **call(Firma)**: Intercepta l'istruzione di chiamata nel codice del **chiamante** (`Client`).
  - **Dove agisce**: Il codice dell'aspetto viene inserito nella classe `Client`.
  - **Quando usarlo**: Necessario se si vuole intercettare chiamate verso librerie esterne o classi che non possiamo modificare/riconciliare.

- **execution(Firma)**: Intercetta l'esecuzione del corpo del metodo nel codice del **chiamato** (Service).
  - **Dove agisce**: Il codice dell'aspetto viene inserito all'interno della classe Service, proprio all'inizio del metodo.
  - **Quando usarlo**: È il metodo standard quando si possiede il codice sorgente delle classi target. Garantisce che l'advice scatti sempre, indipendentemente da chi chiama il metodo.

Visualizziamo la differenza nel codice:

```

1 public class Client {
2     public void doWork() {
3         Service s = new Service();
4
5         // --- QUI agisce il pointcut CALL ---
6         // L'aspetto viene inserito PRIMA che il flusso salti al metodo
7         s.calculate();
8     }
9 }
10
11 public class Service {
12     public void calculate() {
13         // --- QUI agisce il pointcut EXECUTION ---
14         // L'aspetto viene inserito DENTRO il metodo, come prima istruzione
15         System.out.println("Calculating...");
16     }
17 }
```

**Code Listing 2.2:** Differenza tra Call ed Execution

### 2.3.2 Il Weaver

Il **Weaver** è il componente fondamentale dell'architettura AOP (in Java corrisponde al compilatore `ajc` di AspectJ). Possiamo immaginarlo come un "linker" intelligente che fonde il codice normale con il codice degli aspetti.

Il suo compito è trasformare i concetti astratti (advice, pointcut) in bytecode Java standard, poiché la Java Virtual Machine (JVM) non conosce nativamente gli "aspetti".

- **Input**: Prende in ingresso le classi Java standard ('.java') e gli aspetti ('.aj').
- **Processo (Weaving)**: "Tesse" il codice degli advice direttamente dentro le classi target nei punti specificati dai pointcut (modificando il bytecode).
- **Output**: Genera file .class standard che contengono sia la logica di business che la logica trasversale, pronti per essere eseguiti su una JVM standard senza bisogno di plugin speciali.

L'operazione di weaving può avvenire in tre momenti:

1. **Compile-time**: (Più comune) Il codice viene unito durante la compilazione.
2. **Post-compile**: Si modificano file .class o .jar già esistenti (Binary Weaving).

3. **Load-time:** Il codice viene modificato al volo mentre la JVM carica le classi in memoria (richiede un agent specifico).

## 2.4 Esempio Avanzato: Protection Proxy

Un caso d'uso tipico dell'AOP è la sicurezza (Access Control). Vogliamo implementare un meccanismo di protezione per la classe Book senza che il Client o la classe Book debbano gestire esplicitamente i permessi. L'aspetto agisce come un **Proxy**.

### 2.4.1 Collezione del Contesto

Spesso l'advice ha bisogno di accedere agli oggetti coinvolti (es. l'oggetto su cui è chiamato il metodo). Si usano parametri nel pointcut per "catturare" il contesto.

```

1 public aspect Protection { // Sta facendo da Proxy per la classe Book
2     private AuthrzRules r = Client.getRules();
3     pointcut readOp(Book b) : call(String Book.getText()) && target(b);
4     pointcut writeOp(Book b) : call(void Book.append(String)) && target(b);
5
6     String around(Book b) : readOp(b) {
7         System.out.println("Aspect: before read");
8         if (r.canRead(b)) proceed(b);
9         System.out.println("Aspect: Read access has not been granted");
10        return null;
11    }
12
13    void around(Book b) : writeOp(b) {
14        System.out.println("Aspect: before write");
15        if (r.canWrite(b)) proceed(b);
16        else System.out.println("Aspect: Write access has not been granted");
17    }
18 }
```

**Code Listing 2.3:** Aspetto di Protezione (Proxy)

In questo esempio:

- **target(b):** Seleziona i join point dove l'oggetto target è istanza di Book e lo lega alla variabile b.
- **around:** Sostituisce l'esecuzione originale.
- **proceed(b):** È il comando speciale che permette di continuare l'esecuzione del metodo originale intercettato. Senza questa chiamata, il metodo originale non verrebbe mai eseguito.

## 2.5 Caso Studio: Editor di Figure

Per comprendere la potenza dei pointcut complessi, analizziamo un caso studio classico: un editor grafico composto da elementi geometrici (**Point**, **Line**) che implementano l'interfaccia **Figure**.

```

1 public interface Figure {
2     public void moveBy(int dx, int dy);
3 }
4
5 public class Point implements Figure {
6     private int x = 0, y = 0;
7     // Getters e Setters...
8     public void setX(int x) { this.x = x; }
9     public void setY(int y) { this.y = y; }
10    public void moveBy(int dx, int dy) { /* modifica x e y */ }
11 }
12
13 public class Line implements Figure {
14     private Point p1, p2;
15     // Getters e Setters...
16     public void setP1(Point p1) { this.p1 = p1; }
17     public void setP2(Point p2) { this.p2 = p2; }
18     public void moveBy(int dx, int dy) { /* muove p1 e p2 */ }
19 }
```

Code Listing 2.4: Interfaccia e Classi Base

### 2.5.1 Il Problema del "Display Updating"

Ogni volta che un elemento viene spostato o modificato (tramite `moveBy`, `setX`, `setP1`, ecc.), il sistema deve aggiornare lo schermo chiamando `Display.update()`. Senza AOP, siamo costretti a inserire la chiamata `Display.update()` in ogni metodo che modifica lo stato, causando **code scattering** e accoppiando le classi di dominio con la logica di presentazione.

### 2.5.2 Soluzione con AspectJ

Possiamo centralizzare questa logica in un unico aspetto che intercetta tutte le modifiche.

```

1 public aspect DisplayUpdating {
2     // Pointcut composto: unisce tutti i metodi che modificano lo stato
3     pointcut move():
4         call(void Figure.moveBy(int, int)) ||
5         call(void Line.setP1(Point)) ||
6         call(void Line.setP2(Point)) ||
7         call(void Point.setX(int)) ||
8         call(void Point.setY(int));
9
10    // Advice: aggiorna il display dopo ogni modifica
11    after(): move() {
12        Display.update();
13    }
14 }
```

Code Listing 2.5: Aspetto DisplayUpdating

## 2.6 Composizione dei Pointcut

I pointcut possono essere combinati utilizzando gli operatori logici per creare regole di selezione complesse:

- || (OR): Seleziona se almeno una delle condizioni è vera.
- && (AND): Seleziona solo se entrambe le condizioni sono vere.
- ! (NOT): Nega la selezione.

```

1 // Cattura setX O setY
2 pointcut setCoordinate(): call(void Point.setX(int)) || call(void Point.setY(
    int));
3
4 // Cattura setX ma SOLO se chiamata dall'interno della classe Editor
5 pointcut setInEditor(): call(void Point.setX(int)) && within(Editor);

```

**Code Listing 2.6:** Esempi di Composizione

## 2.7 Cattura del Contesto (Context Exposure)

In AspectJ, i pointcut non servono solo a "selezionare" un punto del codice, ma anche a estrarre i valori disponibili in quel contesto (come parametri, oggetti chiamanti, ecc.) per passarli all'advice. Questa tecnica si chiama *Context Exposure*.

Per farlo, si dichiarano delle variabili nel pointcut e si legano ai valori di runtime usando tre designatori specifici: **target**, **args** e **this**.

### 2.7.1 1. Target: L'Oggetto Ricevente

Il designatore **target(Tipo variabile)** cattura l'oggetto **su cui** viene invocato il metodo (il "destinatario" del messaggio).

```

1 // Cattura qualsiasi chiamata a moveBy su un oggetto Figure
2 pointcut move(Figure f):
3     call(void Figure.moveBy(int, int)) && target(f);
4
5 after(Figure f): move(f) {
6     // 'f' è l'oggetto che sta per essere spostato (es. un oggetto Line o
7     // Point)
8     System.out.println("Sto muovendo la figura: " + f);
9     Display.update(f);
}

```

### 2.7.2 2. Args: I Parametri del Metodo

Il designatore **args(variabile, ...)** cattura gli **argomenti** passati al metodo intercettato. Questo evita di dover usare metodi complessi come **getArgs()** dell'oggetto **thisJoinPoint**.

```

1 // Cattura i valori interi passati a moveBy
2 pointcut moveArgs(int x, int y):
3     call(void Figure.moveBy(int, int)) && args(x, y);
4
5 after(int x, int y): moveArgs(x, y) {
6     System.out.println("Spostamento richiesto di X=" + x + ", Y=" + y);
7 }
```

### 2.7.3 3. This: L'OGGETTO "ATTIVO"

Il designatore `this(Tipo variabile)` cattura l'oggetto che è **attualmente in esecuzione** (il riferimento `this` di Java nel punto in cui scatta l'aspetto).

La sua semantica cambia a seconda del tipo di pointcut usato (`call` vs `execution`), ed è qui che nasce la confusione.

#### 2.7.3.1 Differenza cruciale tra This e Target

Per capire la differenza, analizziamo uno scenario con due classi: un `Client` che chiama un `Service`.

```

1 public class Client {
2     public void doWork() {
3         Service s = new Service();
4         s.process(); // <-- JOIN POINT (Punto di intercettazione)
5     }
6 }
```

Se intercettiamo l'istruzione `s.process()` con un pointcut `call`, ci troviamo "fisicamente" ancora dentro la classe `Client`, nel momento in cui sta per chiamare `Service`.

In questo preciso istante:

- **`target(s)`**: Si riferisce all'oggetto `Service` (il chiamato, colui che riceve l'ordine).
- **`this(c)`**: Si riferisce all'oggetto `Client` (il chiamante, colui che sta eseguendo il codice che effettua la chiamata).

**Quando usare `this` con `call`?** È utilissimo quando vuoi intercettare una chiamata *solo se parte da una specifica classe*. Esempio: "Intercetta le chiamate a `save()` solo se provengono da un `Controller`, ma ignorale se provengono da un `Service` interno".

```

1 // Intercetta save() solo se il chiamante (this) è un Controller
2 pointcut saveFromController():
3     call(void Repository.save()) && this(Controller);
```

**Code Listing 2.7:** Filtrare in base al chiamante

## 2.8 Tipologie Avanzate di Advice: Around

Oltre a `before` e `after`, esiste l'advice `around`. È il più potente perché avvolge completamente il join point, permettendo di:

1. Eseguire codice prima e dopo.
2. Decidere *se* eseguire il metodo originale (o bloccarlo).
3. Modificare i parametri in ingresso o il valore di ritorno.

Per eseguire il metodo originale, si deve invocare `proceed()`.

```

1 public aspect Check {
2     pointcut checkValore(int v):
3         args(v) && (call(void Point.setX(int)) || call(void Point.setY(int)));
4
5     void around(int v): checkValore(v) {
6         if (v < 0) {
7             v = 0; // Forza il valore a 0 se negativo
8         }
9         proceed(v); // Esegue il metodo originale con il (nuovo) valore
10    }
11 }
```

Se il metodo intercettato restituisce un valore, l'advice `around` deve restituirlo (o restituirne uno diverso).

```

1 double around(): call(double Product.getPrice()) {
2     double originalPrice = proceed();
3
4     double discountedPrice = originalPrice * 0.8;
5
6     // Il client riceverà il prezzo scontato, senza saperlo!
7     return discountedPrice;
8 }
```

## 2.9 Wildcard e Pattern Matching

Per rendere i pointcut generici e applicabili a molti metodi senza elencarli uno per uno, AspectJ supporta le **Wildcard**:

- \* (Asterisco):
  - Come tipo di ritorno: `* get*()` (qualsiasi tipo).
  - Come parte del nome: `set*()` (metodi che iniziano per set).
  - Come parte del package: `java.*.Date`.
- .. (Due punti):
  - Negli argomenti: `method(..)` (zero o più argomenti di qualsiasi tipo).
  - Nei package: `java..*` (il package java e tutti i suoi sottopackage).
- + (Più):
  - Nei tipi: `Figure+` (l'interfaccia Figure e tutte le classi che la implementano).

```

1 // Qualunque metodo public di Figure (o sottotipi)
2 call(public * Figure+.*(..))
3
4 // Qualunque metodo di Line che inizia con "set"
5 call(* Line.set*(..))
6
7 // Qualunque metodo println di PrintStream
8 call(void java.io.PrintStream.println(*))

```

Code Listing 2.8: Esempi di Wildcard

## 2.10 Accesso Riflessivo al Join Point

Nei casi in cui l'advice necessiti di informazioni generiche sul contesto di esecuzione (ad esempio per realizzare un sistema di tracing o logging universale), AspectJ fornisce una variabile speciale implicita chiamata `thisJoinPoint`. Questa variabile è analoga al `this` di Java, ma invece di riferirsi all'oggetto corrente, si riferisce al **Join Point corrente**.

L'oggetto `thisJoinPoint` offre metodi per accedere a:

- **Informazioni Statiche:** La firma del metodo, il tipo di ritorno, la posizione nel codice sorgente.
- **Informazioni Dinamiche:** Gli argomenti passati, il riferimento all'oggetto target e all'oggetto `this`.

```

1 public aspect TraceV1 {
2     // Cattura l'esecuzione di QUALSIASI metodo nel package 'part'
3     // Esclude l'aspetto stesso per evitare ricorsione infinita
4     pointcut trace():
5         execution(* *.*(..)) && !within(TraceV1) && within(part..*);
6
7     before(): trace() {
8         System.out.println("Entering: " + thisJoinPoint.getSignature());
9
10        // Accesso agli argomenti dinamici
11        Object[] args = thisJoinPoint.getArgs();
12        for (int i = 0; i < args.length; i++) {
13            System.out.println("Arg[" + i + "]: " + args[i]);
14        }
15
16        // Accesso alla locazione sorgente
17        System.out.println("At: " + thisJoinPoint.getSourceLocation());
18    }
19 }

```

I metodi principali dell'interfaccia `JoinPoint` sono:

- `getSignature()`: Restituisce la firma del membro intercettato (nome, tipo, parametri).
- `getArgs()`: Restituisce un array di `Object` con i valori dei parametri attuali.

- `getTarget()`: Restituisce l'oggetto target (destinatario).
- `getThis()`: Restituisce l'oggetto in esecuzione (chiamante o eseguente a seconda del contesto).
- `getKind()`: Restituisce una stringa che identifica il tipo di join point (es. "method-execution", "method-call").

## 2.11 Static Crosscutting

Finora abbiamo analizzato il *Dynamic Crosscutting*, ovvero l'intercettazione del flusso di esecuzione. AspectJ supporta anche lo **Static Crosscutting**, che permette di modificare la struttura statica delle classi (il loro "scheletro") a tempo di compilazione (Weave-time).

Le due funzionalità principali sono:

1. **Inter-type Declarations (ITD)**: Aggiunta di metodi, campi o interfacce a classi esistenti.
2. **Declaration of Errors/Warnings**: Imposizione di regole architettonali verificate dal compilatore.

### 2.11.1 Inter-type Declarations (ITD)

Un aspetto può "iniettare" nuovi membri in una classe target. La sintassi è `TipoClasse.NomeMembro`.

Questo è estremamente potente per implementare pattern in modo non invasivo. Ad esempio, possiamo aggiungere un campo `writeTime` e i relativi metodi di accesso alla classe `Product` senza toccare il file `Product.java`.

```

1 public aspect ChangeProduct {
2     // Aggiunge un campo privato alla classe Product
3     private LocalTime Product.writeTime;
4
5     // Aggiunge un metodo pubblico alla classe Product
6     public void Product.setWriteTime() {
7         this.writeTime = LocalTime.now();
8     }
9
10    // Aggiunge un altro metodo
11    public LocalTime Product.getTime() {
12        return this.writeTime;
13    }
14
15    // Usa i nuovi membri in un advice
16    before(Product p): execution(public void Product.putPrice(*)) && this(p) {
17        p.setWriteTime(); // Ora Product ha questo metodo!
18    }
19 }
```

È anche possibile modificare la gerarchia di ereditarietà:

```

1 // Dichiara che Product estende BasicProduct
2 declare parents: Product extends BasicProduct;
3
4 // Dichiara che Product implementa Serializable
5 declare parents: Product implements Serializable;

```

## 2.11.2 Policy Enforcement (Declare Error/Warning)

Lo Static Crosscutting permette di trasformare regole di design (spesso solo documentate) in vincoli reali controllati dal compilatore. Se il codice viola la regola, la compilazione fallisce con un messaggio personalizzato.

**Esempio 1: Migrazione API** Vogliamo segnalare come deprecata una vecchia chiamata di log.

```

1 declare warning:
2   call(void Logger.log(..)):
3     "Attenzione: Sostituire la chiamata a log() con log2() per efficienza.";

```

**Esempio 2: Vincoli Architetturali (Factory Pattern)** Vogliamo assicurare che le istanze di Product siano create solo tramite la Factory e mai direttamente con `new`.

```

1 public aspect DetectFactoryUse {
2   declare error:
3     call(Product.new(..)) && !within(Creator+):
4       "Errore: Violazione del pattern Factory. Solo i Creator possono
5        istanziare Product.";
}

```

## 2.12 Caso Studio: Adapter Pattern con AOP

Il Pattern **Adapter** risolve incompatibilità di interfaccia tra un **Client** (che si aspetta un metodo `request()`) e un **Adaptee** (che possiede solo `specificRequest()`).

### 2.12.1 Approccio Classico (Object-Oriented)

Si crea una classe wrapper **Adapter** che implementa l'interfaccia attesa e delega le chiamate all'oggetto **Adaptee** interno. **Svantaggi:**

- Il Client deve essere modificato per istanziare l'**Adapter** invece dell'**Adaptee**.
- Proliferazione di classi wrapper.
- Non si ha la garanzia che tutti usino l'**Adapter** (qualcuno potrebbe bypassarlo).

### 2.12.2 Approccio AOP (Aspect Adapter)

Utilizzando lo Static Crosscutting, possiamo "adattare" la classe **Adaptee** direttamente, iniettando il metodo mancante `request()`.

```

1 public aspect AdapterAspect {
2
3     // 1. Introduzione (ITD): Inietta il metodo request() direttamente in
4     // Adaptee
5     public void Adaptee.request() {
6         this.specificRequest(); // Delega interna
7     }
8
9     // 2. Opzionale: Dichiara che Adaptee implementa l'interfaccia Target
10    declare parents: Adaptee implements Target;
11
12    // 3. Policy Enforcement: Impedisce l'uso diretto del vecchio metodo
13    declare warning:
14        call(void Adaptee.specificRequest()) && !within(AdapterAspect):
15            "Non usare specificRequest(), usa il nuovo metodo standard request()";
}

```

Il compilatore AspectJ prende il metodo `request()` e lo inserisce fisicamente nel bytecode della classe `Adaptee`. Una volta compilato, è come se il programmatore avesse scritto quel metodo direttamente dentro la classe Java.

#### Vantaggi:

- **Trasparenza:** Il Client non deve cambiare codice; può continuare a istanziare `Adaptee`, che ora possiede "magicamente" il metodo corretto.
- **Ereditarietà:** Il metodo introdotto è disponibile automaticamente a tutte le sottoclassi di `Adaptee`.
- **Sicurezza:** Il compilatore ci aiuta a eliminare le chiamate al vecchio metodo deprecato.

## 2.13 Pointcut per Campi

Oltre ai metodi, AspectJ permette di intercettare l'accesso ai campi (variabili di istanza) tramite i designatori `get` e `set`.

- `get(Firma)`: Intercetta la **lettura** di un campo.
- `set(Firma)`: Intercetta la **scrittura** (assegnazione) di un campo.

```

1 // Intercetta la scrittura di qualsiasi campo nella classe Rettangolo
2 pointcut fieldWrite(): set(* Rettangolo.*);
3
4 before(): fieldWrite() {
5     System.out.println("Attenzione: stato interno modificato!");
6 }

```

**Code Listing 2.9:** Intercettazione accesso ai campi