

Luca Cabibbo Architettura dei Sistemi Software

Pattern software

dispensa asw310
marzo 2021

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million time over, without ever doing it the same way twice.

Christopher Alexander

1

Pattern software

Luca Cabibbo ASW



- Riferimenti

- Luca Cabibbo. **Architettura del Software: Strutture e Qualità**. Edizioni Efestò, 2021.
 - Capitolo 15, **Pattern software**
- [POSA1] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. **Pattern-Oriented Software Architecture (Volume 1): A System of Patterns**. Wiley, 1996.
- [POSA4] Buschmann, F., Henney, K., and Schmidt, D.C. **Pattern-Oriented Software Architecture (Volume 4): A Pattern Language for Distributed Computing**. Wiley, 2007.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995. Citato come [GoF]

2

Pattern software

Luca Cabibbo ASW



- Obiettivi e argomenti

□ Obiettivi

- discutere il ruolo dei pattern software nel contesto dell'architettura del software
- descrivere alcuni design pattern utili nella definizione dell'architettura

□ Argomenti

- pattern software
- alcuni design pattern utili
- discussione



* Pattern software

□ Un **pattern software** è

- una soluzione provata e ampiamente applicabile a un particolare problema di progettazione
- che è descritta in una forma standard, in modo che possa essere facilmente condivisa e riusata

□ Tre principali categorie di pattern software

- pattern architeturali
- design pattern
- idiomi



- Pattern architetturali

- Un **pattern architetturale** [POSA] – o **stile architetturale** [SSA] – è un pattern per l'architettura del software
 - esprime uno schema per l'organizzazione strutturale fondamentale per i sistemi software
 - un insieme di (tipi di) elementi architetturali predefiniti, con le loro responsabilità e regole e linee guida per l'organizzazione delle relazioni tra di essi
 - esempi – Layers e l'architettura a microservizi



Pattern architetturali

- Detto in modo diverso, un **pattern architetturale** [SAP]
 - è un pacchetto (una combinazione) di decisioni di progetto
 - che è stato applicato ripetutamente in pratica
 - che ha delle proprietà note, che ne consente il riuso
 - e descrive una *classe* di architetture
 - nello studio di un pattern architetturale
 - è necessario comprendere le proprietà del pattern
 - può essere utile comprendere le singole decisioni di progetto implicate dal pattern



Sui pattern architetturali

- I pattern architetturali hanno in genere un ruolo fondamentale nella definizione dell'architettura
 - ogni sistema (ogni sua struttura fondamentale) è basato su un pattern architetturale “principale”
 - ma un sistema o una struttura può essere anche basato sull'applicazione di pattern architetturali “secondari”
 - in ogni caso, uno dei pattern architetturali è dominante, e definisce lo “stile” dell'intero sistema



- Design pattern

- Un **design pattern** [GoF]
 - è la descrizione di oggetti e classi che comunicano, personalizzati per risolvere un problema generale di progettazione in un contesto particolare
- Alcuni esempi di design pattern
 - Singleton, Adapter, Proxy, Observer, ...



Sui design pattern

- Anche i design pattern possono avere un ruolo utile nella definizione dell'architettura
 - per raffinare gli elementi di un sistema software o le relazioni tra di essi
 - sono spesso utili per descrivere le connessioni tra elementi architettonici
 - alcuni pattern architettonici prevedono l'uso di uno o più design pattern
 - sono utili anche per favorire la comunicazione con l'architetto e tra gli sviluppatori



- Idiomi



- Un **idioma**
 - è un pattern di basso livello, specifico di un linguaggio di programmazione – che descrive come implementare aspetti particolari di elementi o delle relazioni tra essi, usando una caratteristica di un certo linguaggio



- Linguaggi di pattern

- Un **pattern language** (*linguaggio di pattern*)
 - una famiglia di pattern correlati – con una discussione relativa alle loro correlazioni
 - ne esistono diversi – specifici per la progettazione di certi tipi di sistemi o per certi tipi di requisiti
 - ad es., pattern per i sistemi distribuiti [POSA4], per la sicurezza, per l'integrazione di applicazioni, per i microservizi, per il software cloud native, ...
 - infatti, i pattern non sono tra di loro indipendenti, e quindi può essere utile ragionare anche sulle loro relazioni



Linguaggi di pattern – [GoF] e [POSA]





* Alcuni design pattern utili

- Descriviamo ora, in modo sintetico, alcuni **design pattern** di uso comune – utili nella definizione dell'architettura
 - alcuni di questi design pattern sono utilizzati come elementi di pattern architeturali
 - altri possono essere utilizzati nella realizzazione delle connessioni tra elementi architeturali



- Adapter [GoF]

- **Adapter**
 - adatta l'interfaccia di un elemento di un sistema a una forma richiesta da uno dei suoi client
 - ha lo scopo di convertire l'interfaccia di una classe in un'altra interfaccia richiesta dal client – *Adapter* consente a classi diverse di operare insieme quando ciò non sarebbe altrimenti possibile a causa di interfacce incompatibili



Adapter

□ Contesto

- bisogna connettere diversi elementi eterogenei

□ Problema (e forze)

- un elemento (*client*) vuole usare i servizi offerti da un altro elemento (*adaptee*) – ma l'interfaccia dell'*adaptee* non è adatta al client
- il client potrebbe usare direttamente l'*adaptee* – ma questo accoppiamento è indesiderato
- preferibile l'uso di un intermediario (adattatore)
 - l'adattatore dovrebbe solo fornire un servizio di adattamento e “traduzione” – e non dovrebbe avere effetti negativi sulle qualità



Adapter

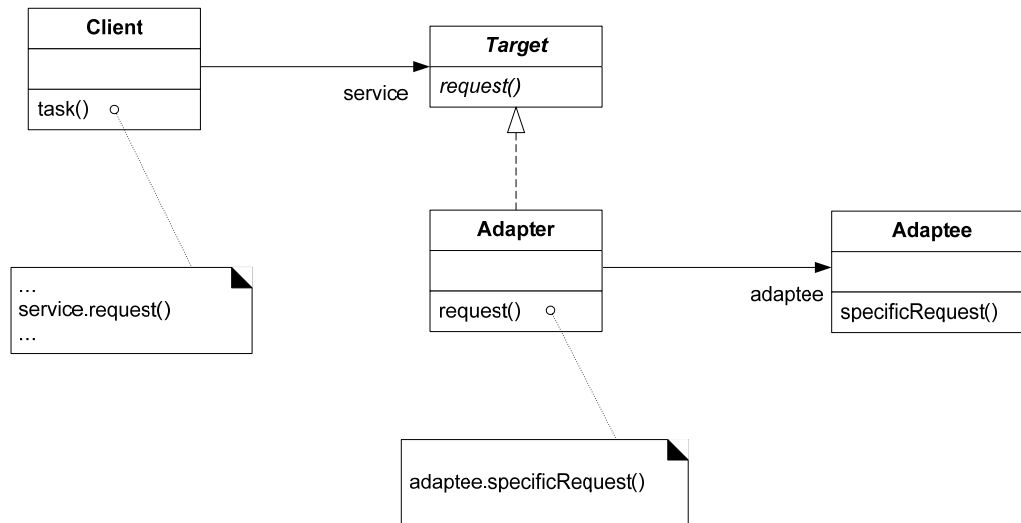
□ Soluzione

- introdurre un terzo elemento separato tra il client e l'*adaptee* – un elemento adattatore (*adapter*)
 - un intermediario nella comunicazione tra client e *adaptee*
 - l'adattatore ha solo il ruolo di interpretare le richieste del client, trasformarle in richieste all'*adaptee*, ottenere risposte dall'*adaptee*, trasformarle in risposte al client
 - l'adattatore ha di solito un'interfaccia (*target*) diversa da quella dell'*adaptee*



Adapter

□ Struttura (e dinamica) della soluzione



Adapter

□ Conseguenze

- 😊 disaccoppiamento delle implementazioni del client e dell'adaptee
- 😊 l'adaptee può essere usato da diversi tipi di client, ciascuno col suo adattatore
- 😞 l'indirizione addizionale potrebbe ridurre l'efficienza
- 😞 ci potrebbe essere un aumento nell'overhead per la manutenzione nel caso in cui cambiasse il servizio offerto dall'adaptee



- Proxy [GoF]

- Il design pattern **Proxy** fa comunicare i client di un componente che offre servizi con un rappresentante di quel componente
 - **Proxy** fornisce un surrogato o un segnaposto per un altro oggetto – per controllarne l'accesso



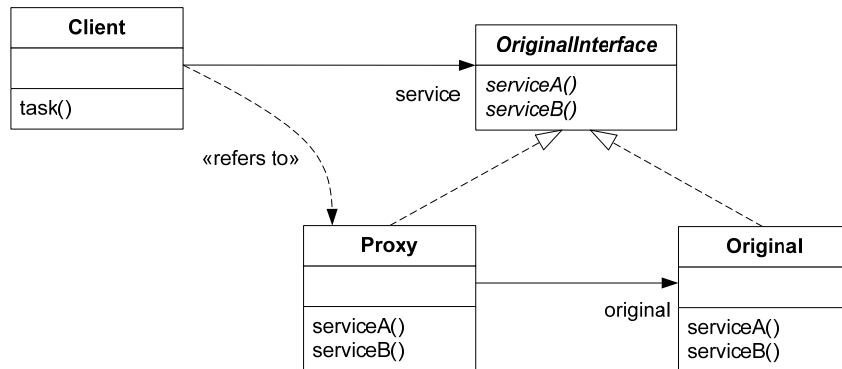
Proxy

- **Contesto**
 - l'accesso diretto tra un client e un componente è tecnicamente possibile – ma non è l'approccio migliore
- **Problema**
 - l'accesso diretto a un componente è spesso inappropriato – per motivi di accoppiamento, efficienza, sicurezza, ...
- **Soluzione**
 - il client viene fatto comunicare con un rappresentante del componente – il **proxy** – che non fa solo da intermediario, ma esegue delle ulteriori pre- e post-elaborazioni



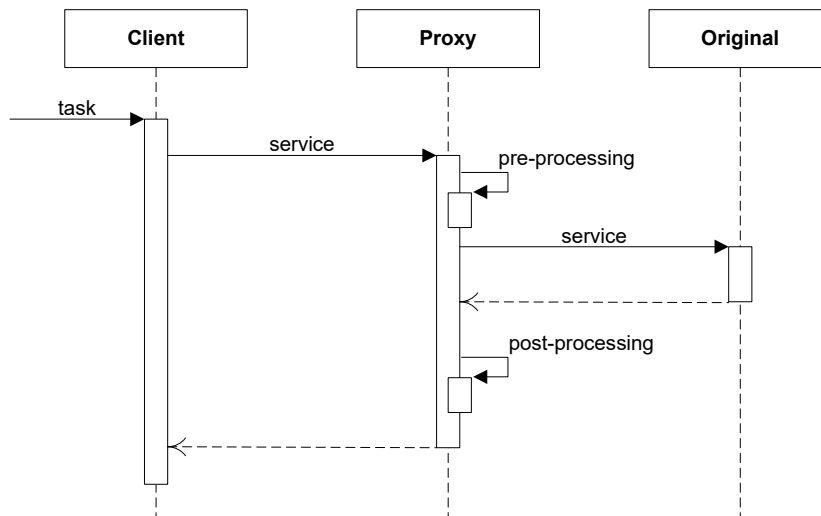
Proxy

□ Struttura della soluzione



Proxy

□ Dinamica della soluzione





Possibili applicazioni

- Alcuni casi specifici comuni di Proxy
 - *Remote Proxy*
 - *Protection Proxy*
 - *Cache Proxy*
 - *Synchronization Proxy*
 - *Virtual Proxy*
 - ...



Proxy

- Conseguenze
 - 😊 un proxy remoto può nascondere al client il fatto che il servizio sia remoto
 - 😊 un proxy virtuale può svolgere ottimizzazioni come la creazione di oggetti su richiesta
 - 😊 un proxy di protezione può svolgere operazione aggiuntive di gestione del servizio
 - 😊 ...
 - 😞 l'indirezione addizionale potrebbe ridurre l'efficienza



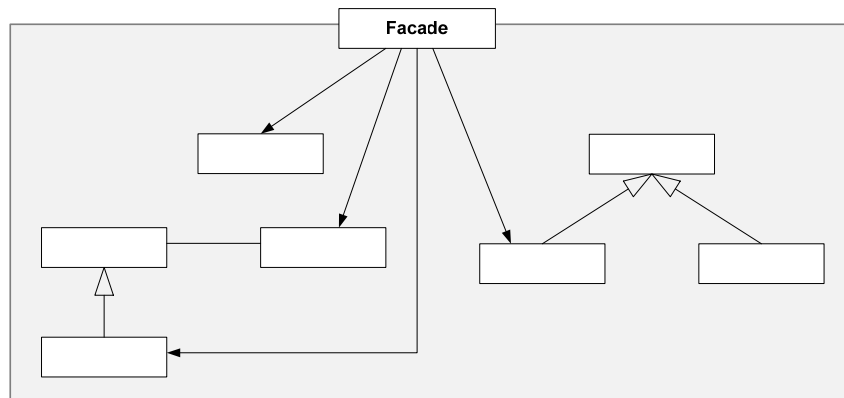
- Factory [GoF]

- [GoF] definisce alcuni design pattern *creazionali*
 - **Factory Method** definisce un'interfaccia per la creazione di un oggetto, ma lascia alle sottoclassi decidere da quale classe instanziare l'oggetto
 - **Abstract Factory** fornisce un'interfaccia per creare una famiglia di oggetti correlati o dipendenti tra di loro, senza specificare le loro classi concrete
- Questi pattern assegnano la responsabilità di creare uno o più oggetti a una factory separata
 - la factory fornisce ai suoi client un'interfaccia per gestire la creazione – i client della factory non devono conoscere i dettagli della creazione
 - una factory può anche occuparsi dell'iniezione delle dipendenze degli oggetti che crea (*application context*)



- Facade [GoF]

- Il design pattern **Facade**
 - fornisce un'interfaccia unificata per un insieme di interfacce presenti in un sottosistema
 - **Facade** definisce un'interfaccia di livello più alto, che rende il sottosistema più semplice da utilizzare





- **Facade** e tattiche per la modificabilità
 - una facade sostiene l'applicazione di **Encapsulate**
 - la facade può definire l'interfaccia di un sottosistema
 - una facade può essere un'applicazione di **Use an intermediary**
 - una facade può sostenere l'applicazione di **Abstract common services**
 - la facade può nascondere (ai client di un'astrazione) un'implementazione (complessa) dell'astrazione
 - una facade può essere un'applicazione di **Restrict dependencies**



- **Observer**
 - definisce una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo siano notificati e aggiornati automaticamente
 - chiamato anche **Publisher/Subscriber** [POSA1]



Observer

□ Contesto

- un elemento (*publisher*) crea informazioni che sono di interesse per altri elementi (*subscriber*)

□ Problema

- diversi tipi di oggetti *subscriber* sono interessati ai cambiamenti di stato o agli eventi di un oggetto *publisher*
- ciascun subscriber vuole reagire in un modo proprio quando un publisher genera un evento
- il publisher vuole mantenere un accoppiamento basso verso i suoi subscriber



Observer

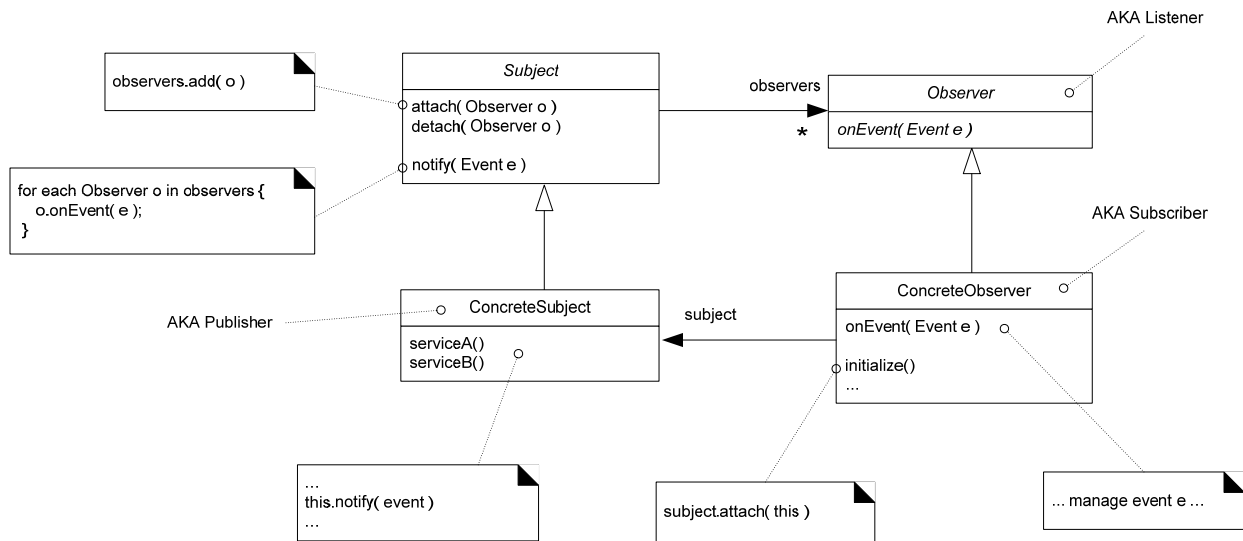
□ Soluzione

- definisci un'interfaccia *observer* oppure *listener*
- i subscriber implementano questa interfaccia
- il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi – li avvisa quando si verifica un evento
- componenti – publisher e subscriber
- connettori – un canale affidabile per la trasmissione delle notifiche



Observer

□ Struttura (e dinamica) della soluzione



Observer

□ Conseguenze

- 😊 accoppiamento debole tra il publisher e i suoi subscriber
- 😊 supporto per comunicazione broadcast
- 😊 possibilità di aggiungere/rimuovere i subscriber dinamicamente
- 😊 rimuove la necessità del polling da parte dei subscriber
- 😞 potrebbe essere difficile comprendere le relazioni di dipendenza tra i vari elementi
- 😞 effetto imprevedibile degli aggiornamenti – una modifica in un publisher può scatenare una catena di aggiornamenti e sincronizzazioni su tutti i subscriber
- 😞 implementazione complessa – soprattutto se è richiesta una consegna affidabile dei messaggi



* Discussione

- I pattern software costituiscono delle soluzioni ben provate e applicabili a problemi specifici ma ricorrenti di progettazione
 - i pattern sono descritti in una forma standard, in modo da poter essere facilmente condivisi e riutilizzati
 - tre categorie principali di pattern – pattern (o stili) architetturali, design pattern e idiomi – con un ruolo più o meno utile nell'architettura del software
 - i pattern architetturali guidano l'organizzazione della struttura fondamentale dei sistemi software
 - i design pattern sono utili nella realizzazione delle connessioni tra elementi architetturali



Discussione

- Nell'ambito del processo di progettazione di un'architettura software
 - i pattern architetturali vengono usati soprattutto per la progettazione iniziale dell'architettura – per sostenere gli scenari architetturali più rilevanti
 - per identificare un insieme (iniziale) di elementi architetturali, e un insieme (iniziale) di relazioni e interazioni tra di essi
 - i pattern architetturali possono essere usati anche per guidare la decomposizione di elementi architetturali complessi
 - i design pattern vengono invece usati soprattutto per il raffinamento dell'architettura
 - nella progettazione delle connessioni e delle interazioni tra elementi architetturali



Ruolo e benefici dei pattern software

- Alcuni ruoli svolti da pattern (e linguaggi di pattern)
 - deposito di conoscenza
 - esempi di buone pratiche, già applicate con successo
 - un linguaggio per discutere problemi di progettazione
 - un aiuto alla standardizzazione
 - una sorgente di miglioramento continuo
 - incoraggiamento alla generalità e base per l'adattamento

- Benefici principali dei pattern – soprattutto dal punto di vista dell'architettura del software
 - riduzione del rischio, sulla base di soluzioni provate e ben comprese
 - un aiuto nella gestione della complessità del software
 - incremento della produttività, della standardizzazione e della qualità