Luca Cabibbo

Architettura dei Sistemi Software

Soluzione provata ed esemplare per un problema di progettazione ricorrente

Pattern software

dispensa asw310

ottobre 2024

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

- Riferimenti

- Luca Cabibbo. **Architettura del Software**: Strutture e Qualità. Edizioni Efesto, 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]

- 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**
  - una soluzione provata e ampiamente applicabile a un particolare problema di progettazione
  - descritta in una forma standard, per poter essere condivisa e riusata facilmente
- Tre principali categorie di pattern software
  - pattern architetturali
  - design pattern

  } Ci interessano
  - idiomi Soluzioni di problemi specifici per linguaggi di programmazione specifici, grana piccola

Variano nella granularità del problema e della soluzione, ma non solo

- Pattern architetturali

- Un **pattern architetturale** [POSA] – o **stile architetturale** [SSA] – è un pattern per l'architettura del software
  - affronta un problema architetturalmente significativo – anche la soluzione è architetturale
  - 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

Sono una coppia problema-soluzione: affronta un problema architetturalmente significativo (cioè che riguarda la QUALITÀ) la cui soluzione riguarda l'architettura (cioè le STRUTTURE) del sistema software

La soluzione è definita da un insieme di elementi architetturali e dalle regole e interazioni tra essi

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 spesso un ruolo fondamentale nella definizione dell'architettura
  - ogni sistema (ogni sua struttura fondamentale) è basato su un pattern architetturale "principale"
    - 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

Se sto definendo un'architettura quanti pattern mi servono? Normalmente uno principale, ma in un pattern possono essere presenti pattern diversi. Es il pattern principale è quello a micro servizi, ma ci sono altre strutture che possono essere basati su pattern diversi (es pattern esagonale per il codice)

Per la stessa architettura quindi possono essere applicati più pattern, di solito uno per ogni vista ma può non essere necessariamente così

- 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
  - per descrivere le connessioni tra elementi architetturali
  - alcuni pattern architetturali prevedono l'uso di uno o più design pattern
  - come tutti i pattern, sono utili per favorire la comunicazione con l'architetto e tra gli sviluppatori

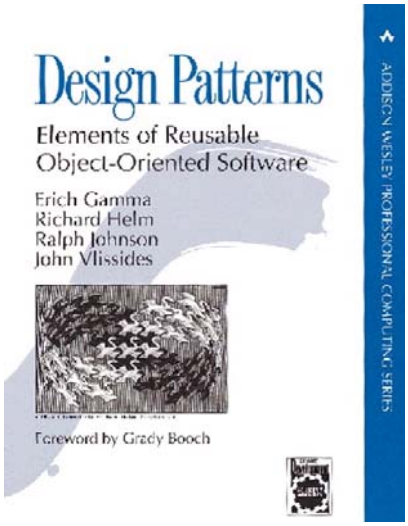
- Idiomi

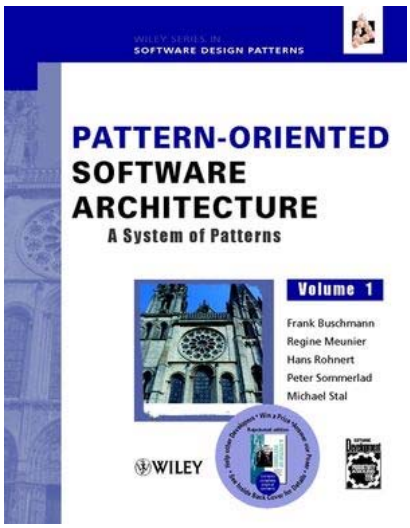
- Un **idioma** Es come implemento singleton in Java? Come in c++? A livello di architettura non ci interessano
  - è un pattern di basso livello, specifico di un linguaggio di programmazione – che descrive come implementare aspetti particolari di elementi o delle relazioni tra essi, usando una caratteristica 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, …

Linguaggi di pattern – [GoF] e [POSA]





* Alcuni design pattern utili

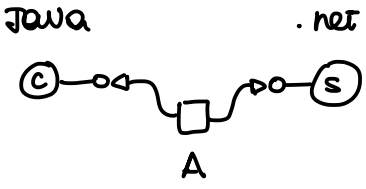
Design pattern utili nel contesto delle architetture. Non sono ancora pattern architetturali

- 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 architetturali
  - altri possono essere utili per realizzare connessioni tra elementi architetturali

- 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

Ci sono due elementi che devono comunicare ma le loro interfacce sono incompatibili. Cioè client e server, client fa richieste al server in modo diverso da quello che il server vorrebbe. Adapter suggerisce di interporre un intermediario tra questi due elementi, formato da un'interfaccia e un'implementazione. La prima dice quali sono i modi di comunicazione di questi due elementi, la seconda mi risolve la comunicazione. L'adapter introduce quindi un'interfaccia gradita al client, SULLA BASE dell'interfaccia gradita dal server. Quindi il client fa richieste "come vuole lui" e l'adapter le adatta allo standard richiesto dal server.



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
  - per evitare questo accoppiamento è preferibile l'uso di un intermediario (l'adattatore)
    - l'intermediario dovrebbe solo fornire un servizio di adattamento e "traduzione" – e non dovrebbe avere effetti negativi sulle qualità

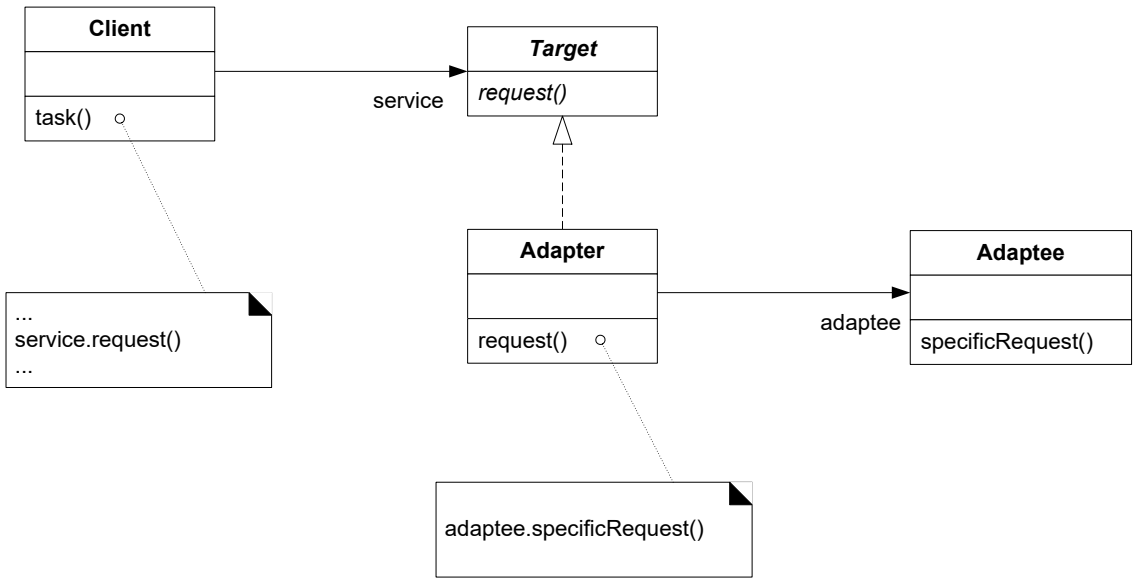
Adapter

- 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

Molte delle tattiche per la modificabilità sono implementare in questo design pattern:
Si introduce un'interfaccia (INCAPSULATE)
Sto spezzando le responsabilità rispetto a … (…)



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'indirezione 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

Strutturalmente adapter e proxy sono simili. Una differenza è che quando si una proxy l'interfaccia dei due oggetti è simile e compatibile, ma questa non è la scelta migliore

Proxy

- Contesto
  - l'accesso diretto tra un client e un componente è tecnicamente possibile – ma non è l'approccio migliore

I componenti non devono avere responsabilità di controllare aspetti qualitativi che quindi vengono lasciati alla responsabilità dei connettori

- Problema
  - l'accesso diretto a un componente è spesso inappropriato – per motivi di accoppiamento, efficienza, sicurezza, ...

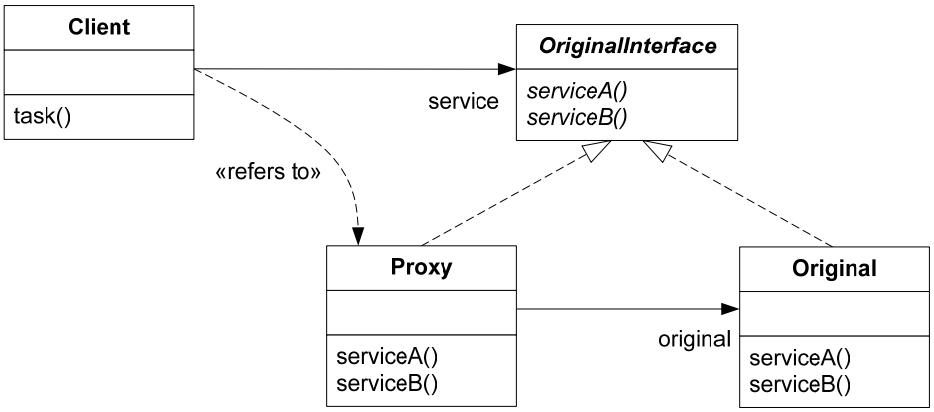
Sono problemi di qualità

- 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

Al proxy viene assegnata la gestione di questa qualità. Non fa da adattatore, ma da intermediario

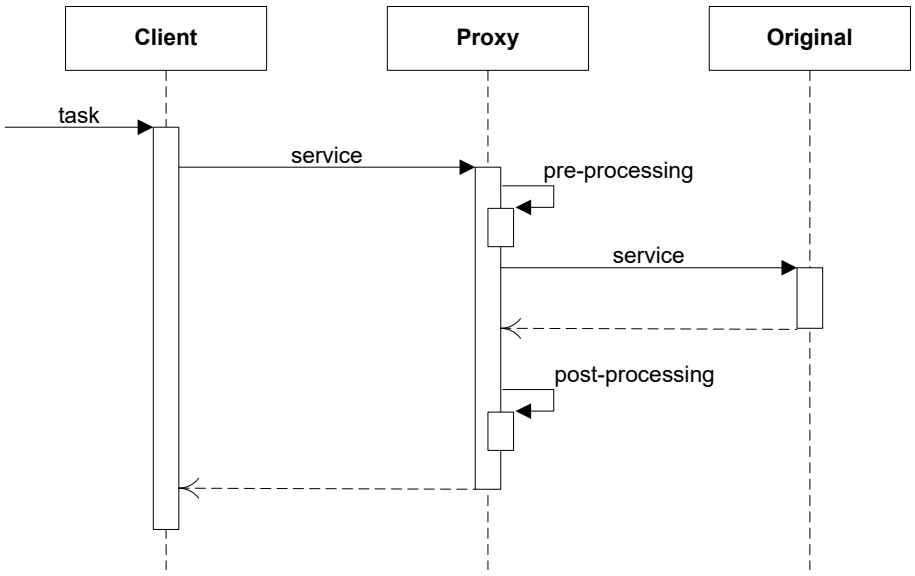
Proxy

- Struttura della soluzione



Proxy

- Dinamica della soluzione



Possibili applicazioni

inizio registrazione

- Alcuni casi specifici comuni di Proxy
  - *Remote Proxy*
  - *Protection Proxy* Fa controllo dell'accesso
  - *Cache Proxy*
  - *Synchronization Proxy* Fa controlli sulla sincronizzazione dei due elementi
  - *Virtual Proxy*
  - ...

Proxy

Intermediario che si occupa della gestione di qualità (semplici).
Se voglio gestire più qualità uso più 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]

In realtà non è un design pattern, è più un mix di altri due design pattern

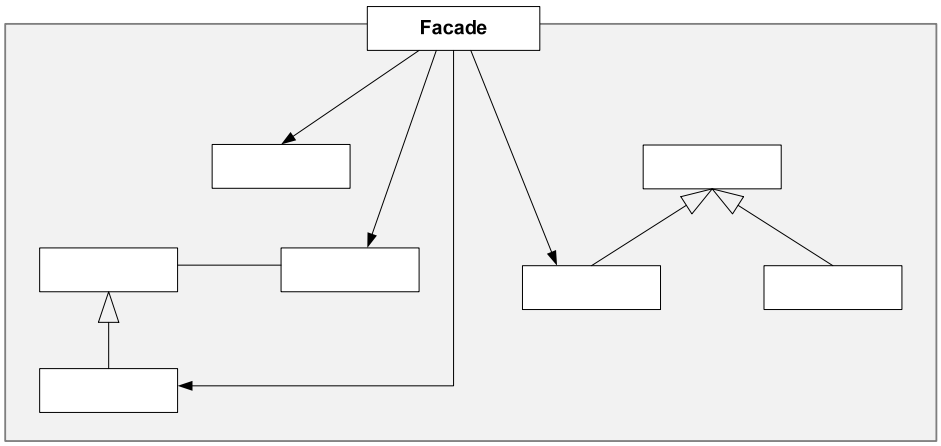
- [GoF] definisce alcuni design pattern *creazionali*
  - **Factory Method** definisce un'interfaccia per la creazione di un oggetto, ma lascia alle sottoclassi decidere da quale classe instanziare l'oggetto
  - **Abstract Factory** fornisce un'interfaccia per creare una famiglia di oggetti correlati 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]

Questa rappresentazione raffigura un componente formato da più classi e interfacce. Senza una facade, un altro componente per relazionarti con questo dovrebbe interagire (e quindi mettersi in una relazione di dipendenza) con molte interfacce dello stesso componente, e quindi avere con esso un alto livello di accoppiamento.

Facade rappresenta una interfaccia unificata per cui chi interagisce con un componente in un sistema lo fa solo con la facade e quindi abbassa molto il livello di accoppiamento.

- 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

- *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 [GoF]

- **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 è considerato il design pattern, publisher/subscriber è considerato il pattern architetturali, quindi cambia la grana.

Il contesto è questo: applicazione con tanti oggetti. Quando ci sono cambiamenti significativi su alcuni oggetti, questi cambiamenti vanno comunicati ad altri (molteplici) oggetti in modo che questi ultimi facciano qualcosa in risposta ai cambiamenti (ogni oggetto potrebbe dover fare qualcosa di diverso). Quindi ci sono oggetti (publisher) che creano eventi (evento: "é successo qualcosa di importante") che vengono recepiti da altri oggetti (subscriber) che sono interessati ad essere notificati dai cambiamenti avvenuti sulla prima categoria di oggetti. I subscriber che ricevono un evento possono fare cose diverse (o niente) e la relazione pub/sub può cambiare dinamicamente nel tempo (possono arrivare nuovi sub, altri possono andarsene). Si vuole un accoppiamento basso dal pub verso il sub.

Observer

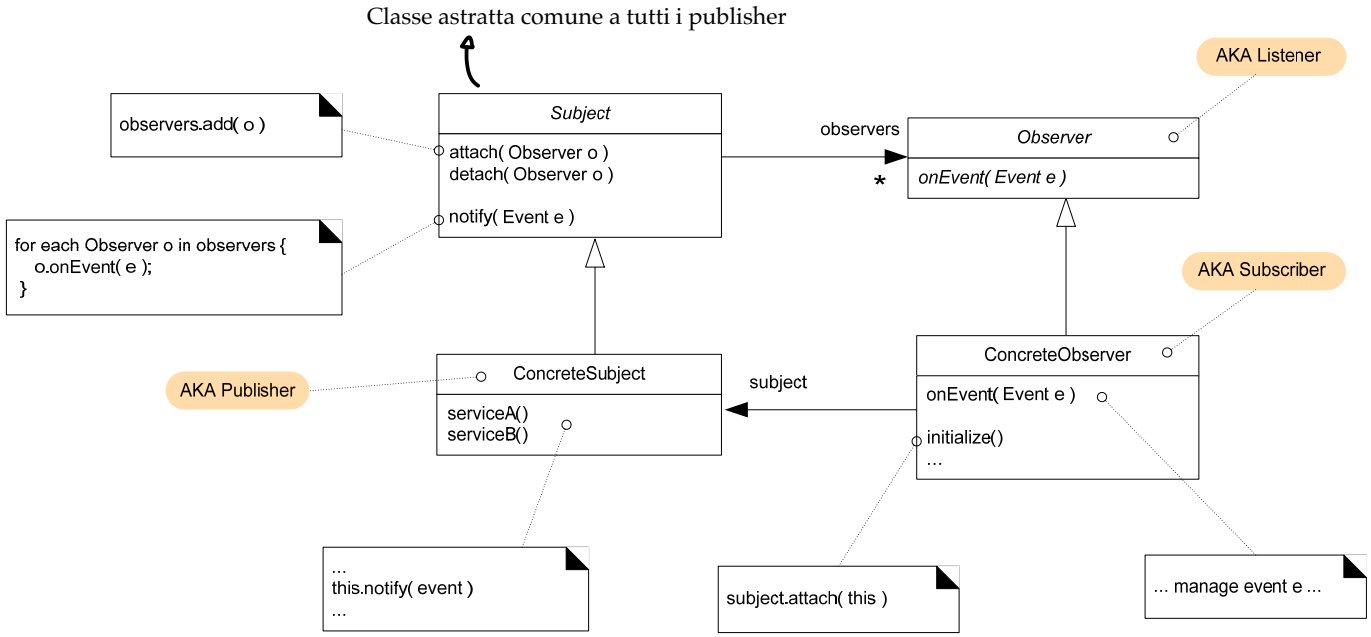
- Contesto
  - un elemento (*publisher*) crea informazioni che sono di interesse per altri elementi (*subscriber*)
- Problema
  - diversi tipi di oggetti *subscriber* sono interessati ai cambiamenti di stato o agli eventi di un oggetto *publisher*
  - ciascun subscriber vuole reagire in un modo proprio quando un publisher genera un evento
  - il publisher vuole mantenere un accoppiamento basso verso i suoi subscriber

Observer

- Soluzione
  - definisci un'interfaccia *observer* (o *listener*) Che si interpone tra pub e sub
  - i subscriber implementano questa interfaccia
  - il publisher, mediante questa interfaccia
    - registra dinamicamente i subscriber che sono interessati ai suoi eventi
    - avvisa i subscriber quando si verifica un evento
  - componenti – publisher e subscriber
  - connettori – un meccanismo 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
  - sono descritti in una forma standard, in modo da poter essere facilmente condivisi e riusati
  - 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