Luca Cabibbo

Architettura dei Sistemi Software

Architettura esagonale

dispensa asw360

marzo 2021

There must be a cause why snowflakes have the shape of six-cornered starlets. It cannot be chance. Why always six?

Johannes Kepler

- Riferimenti

- Luca Cabibbo. **Architettura del Software**: Strutture e Qualità. Edizioni Efesto, 2021.
  - Capitolo 20, Architettura esagonale
- Cockburn, A. Hexagonal Architecture. 2005.
  - https://alistair.cockburn.us/hexagonal-architecture/
- Vernon, V. **Implementing Domain-Driven Design**. Addison-Wesley, 2013.
- Richardson, C. **Microservices Patterns**: With examples in Java. Manning, 2019.

- Obiettivi e argomenti

- Obiettivi
  - presentare l'architettura esagonale
- Argomenti
  - introduzione
  - architettura a strati e inversione delle dipendenze
  - architettura esagonale
  - discussione

* Introduzione

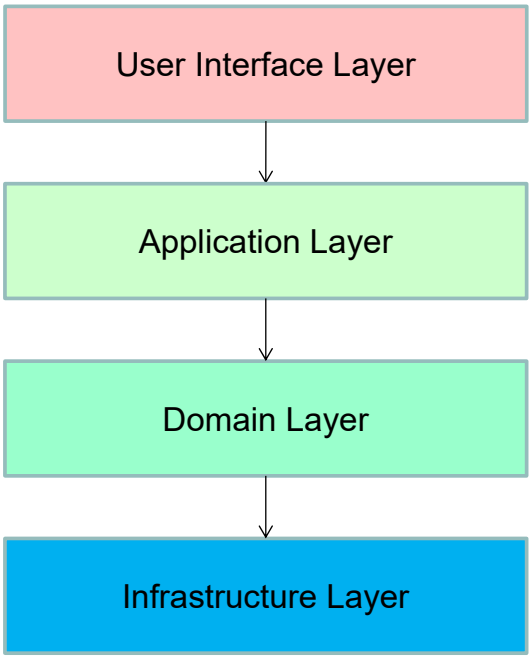
- L'architettura esagonale è un altro pattern architetturale comune
  - è diffuso soprattutto nel contesto di DDD e dell'architettura a microservizi
  - nel contesto di un sistema distribuito, consente di definire l'architettura di un singolo componente o servizio software

Responsabilità di business e infrastrutturali

- Tipi di responsabilità degli elementi software
  - *responsabilità di business* – o *logica di business*
    - una o più funzionalità applicative
  - *responsabilità infrastrutturali*
    - sono relative all'accesso ai servizi infrastrutturali
    - un *servizio infrastrutturale* è un servizio tecnico (non applicativo) – ad es., un database o un message broker
- Intuitivamente
  - le responsabilità infrastrutturali competono ai "connettori"
  - le responsabilità di business competono ai "componenti"
  - le responsabilità di business dovrebbero essere disaccoppiate da quelle infrastrutturali

* Architettura a strati e inversione delle dipendenze

- Consideriamo di nuovo l'architettura a strati "tradizionale" – ad es., la *Layered Architecture* di DDD

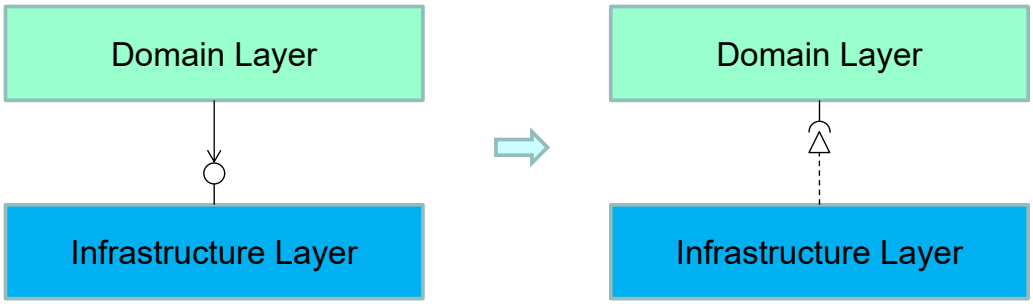


DIP e architettura a strati

- Consideriamo di nuovo anche il *principio di inversione delle dipendenze* (*DIP*)
  - i moduli di alto livello (più importanti) non dovrebbero dipendere dai moduli di basso livello (meno importanti) – piuttosto, entrambi dovrebbero dipendere da opportune astrazioni
  - le astrazioni non dovrebbero dipendere dai dettagli – piuttosto, i dettagli dovrebbero dipendere dalle astrazioni
  - l'architettura a strati tradizionale soddisfa il DIP?

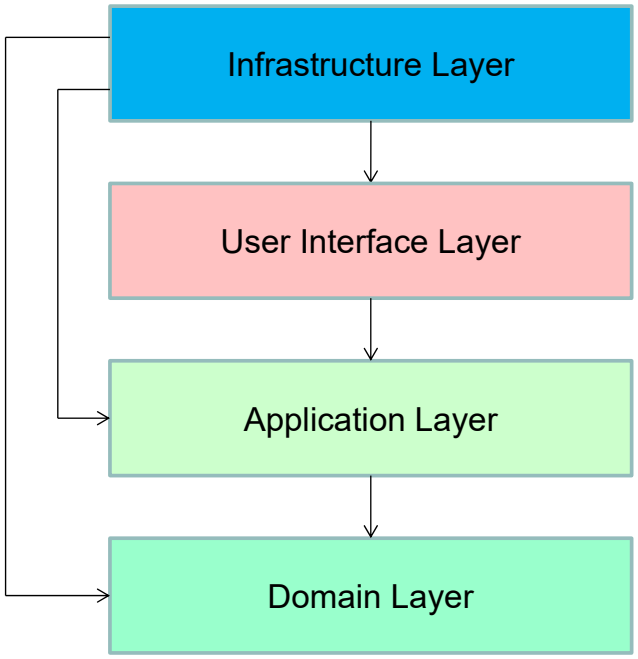
Architettura a strati e DIP

- L'architettura a strati tradizionale non soddisfa il DIP
  - è però possibile applicare il DIP, introducendo delle "opportune astrazioni"



  - l'idea è definire delle interfacce per le responsabilità infrastrutturali negli strati in cui queste responsabilità sono richieste – e implementarle nello strato dell'infrastruttura
    - ad es., per gestire l'accesso ai dati persistenti, oppure per inviare dei messaggi su un canale per messaggi

Architettura a strati e DIP



Verso l'architettura esagonale

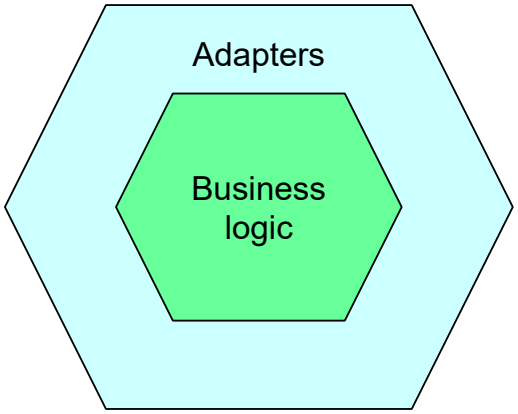
- L'architettura esagonale, intuitivamente
  - ha origine nell'architettura a strati
  - a cui però viene prima applicato il DIP
  - e poi viene rimossa l'asimmetria sopra-sotto dell'architettura a strati

* Architettura esagonale

- Il pattern architetturale **Hexagonal Architecture** (*architettura esagonale*) – chiamato anche **Ports and Adapters**
  - nel contesto dei sistemi distribuiti, aiuta ad organizzare un singolo componente distribuito – chiamato un *servizio* (*servizio applicativo*)
  - in pratica, ogni servizio ha sia responsabilità di business che responsabilità di presentazione e infrastrutturali
  - questo pattern sostiene
    - un accoppiamento debole tra le responsabilità di business e le altre responsabilità
    - un'interazione flessibile tra i servizi e con altre entità esterne

Architettura esagonale

- Ogni "esagono" rappresenta un servizio – che è suddiviso in due parti
  - *interno* (*inside*) – responsabilità di business
  - *esterno* (*outside*) – responsabilità di presentazione e infrastrutturali



Architettura esagonale

- L'interno di un servizio (*logica di business*) implementa un insieme di funzionalità e responsabilità di business
  - implementa gli strati Domain e Application del servizio applicativo
  - non implementa responsabilità di presentazione o infrastrutturali
    - ma le supporta, definendo delle interfacce utili per la presentazione e per l'accesso ai servizi infrastrutturali

Architettura esagonale

- Il servizio (la sua logica di business) ha la necessità di interagire con diversi tipi di entità esterne
  - ad es., gli utenti, i test automatizzati e la base di dati, oppure anche altri servizi
  - si vuole che la logica di business possa interagire con altre entità esterne
    - con un accoppiamento basso – evitando l'intreccio tra il codice della logica di business e quello relativo alle interazioni con le entità esterne
    - in modo flessibile

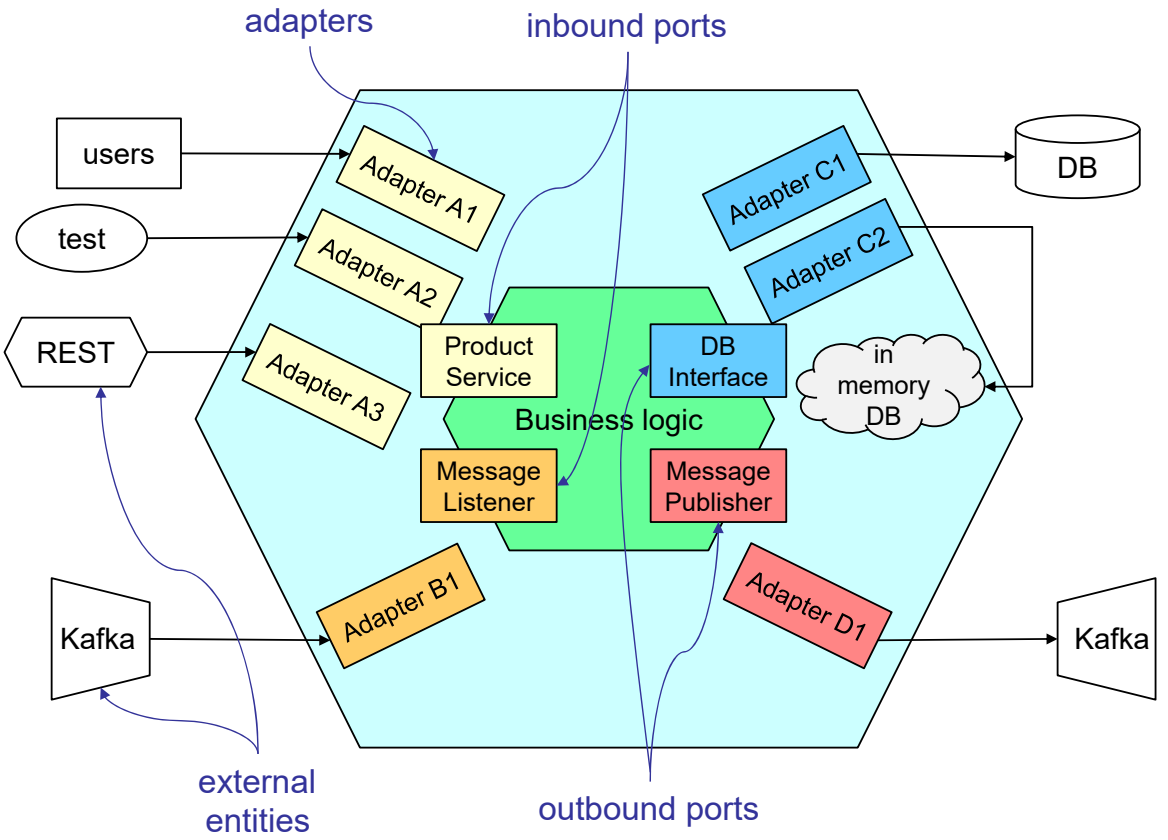
Porte e adattatori

- La logica di business interagisce con le entità esterne tramite porte e adattatori
  - ciascuna *porta* (definita nella logica di business) definisce un'API che rappresenta una modalità di interazione con la logica di business
  - due tipi di porte
    - *inbound port* – un'interfaccia fornita
    - *outbound port* – un'interfaccia richiesta
  - esempi di porte per un servizio S
    - porte per interagire con i suoi utenti (sul web) e per ricevere chiamate remote da altri servizi
    - porte per l'accesso alla base di dati e per effettuare chiamate remote ad altri servizi
    - porte per lo scambio asincrono di messaggi

Porte e adattatori

- La logica di business interagisce con le entità esterne tramite porte e adattatori
  - intuitivamente, gli adattatori implementano lo strato Presentation e lo strato Infrastructure del servizio
  - ciascun *adattatore* (definito nell'esterno del servizio) è relativo a una specifica porta (un'API interna), e adatta le interazioni con un'entità esterna a quella porta
  - due tipi di adattatori – *inbound adapter* e *outbound adapter*
  - esempi di adattatori per un servizio S
    - un controller Spring Web MVC
    - un adattatore per l'accesso a una base di dati MySQL – un altro per l'accesso a una base di dati inmemory
    - adattatori per ricevere ed effettuare chiamate REST – e altri per le chiamate gRPC
    - adattatori per inviare e ricevere messaggi con Kafka

Architettura esagonale



Porte

- Una *porta* rappresenta una modalità di interazione con il servizio (con la sua logica di business), con uno scopo specifico
  - una nozione volutamente generica e flessibile
  - ogni porta è rappresentata graficamente da un lato dell'esagono
    - ma non vuol dire che ogni servizio debba avere esattamente sei porte
  - ogni porta è in corrispondenza con un'API interna (un'interfaccia) della logica di business
    - ad es., in DDD potrebbero essere service, repository e application service

Adattatori

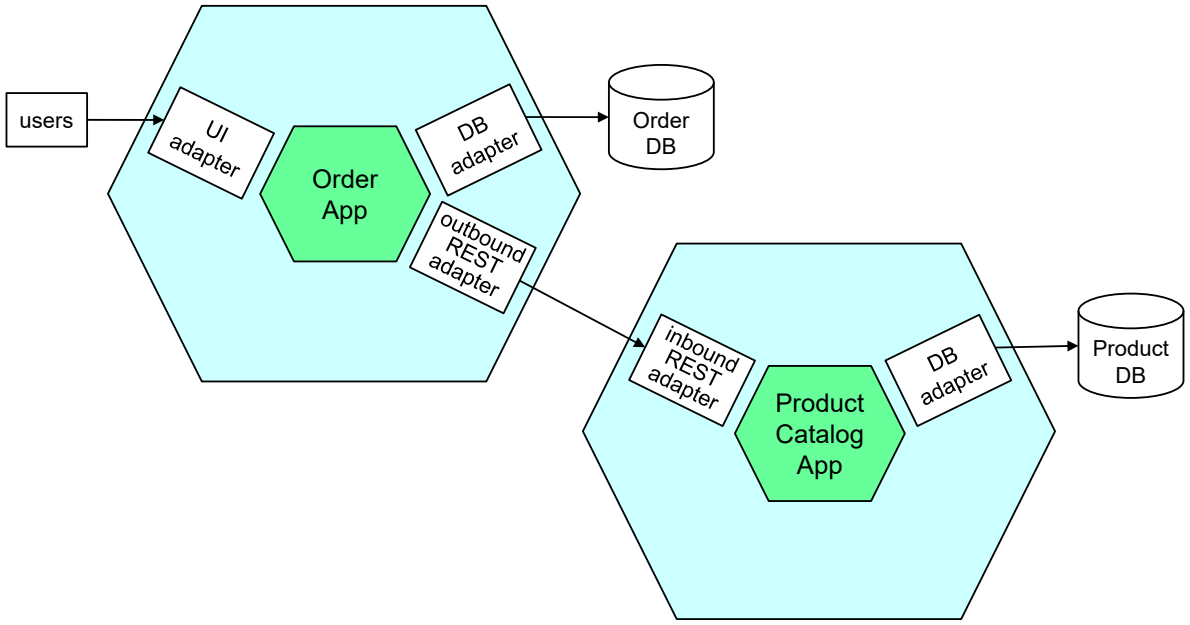
- Ciascun *adattatore* è relativo all'interazione con una specifica tipologia di entità esterna – adatta il tipo di interazione richiesto dall'entità esterna all'API interna dell'applicazione, e viceversa
  - ad esempio
    - un controller web MVC, l'implementazione di un repository JPA, un controller REST, un endpoint per messaggi Kafka
  - per ciascuna porta ci possono essere più adattatori
  - ciascun adattatore può essere utilizzato anche da più entità esterne differenti

Scenari

- Alcuni scenari principali
  - un'entità esterna interagisce con un servizio attraverso un adattatore specifico, relativo a una porta specifica
    - l'adattatore converte questo evento esterno in un'invocazione di un'operazione o in un messaggio, e lo passa alla logica di business tramite l'API di quella porta
  - la logica di business di un servizio deve interagire con l'esterno
    - la logica di business interagisce mediante un adattatore di una porta – che converte la richiesta o il messaggio in un formato appropriato per l'entità esterna
  - in entrambi i casi, la logica di business è indipendente sia dalla natura dell'entità esterna che dagli specifici adattatori utilizzati e dalle tecnologie sottostanti

Scenari

- Alcuni scenari principali
  - un sistema software distribuito composto da più servizi, che interagiscono tra di loro mediante porte e adattatori



Conseguenze

- L'architettura esagonale ha impatto positivo su diverse qualità
  - modificabilità e flessibilità
    - ☺ l'accoppiamento debole tra l'interno e l'esterno di un servizio sostiene la possibilità di implementare e di far evolvere la logica di business in modo indipendente dagli adattatori
    - ☺ l'indirezione fornita dagli adattatori consente un'interazione flessibile di un servizio con altre entità esterne e con altri servizi
    - ☺ l'architettura esagonale è compatibile con team cross-funzionali

Conseguenze

- L'architettura esagonale ha impatto positivo su diverse qualità
  - verificabilità
    - ☺ l'ampio uso di interfacce sostiene l'utilizzo di test double
    - ☺ è possibile verificare separatamente la logica di business e gli adattatori
    - ☺ è possibile usare adattatori specifici per i test (ad es., per un inmemory database)
  - interoperabilità
    - ☺ grazie alle indirezioni fornite dagli adattatori
  - prestazioni
    - ☹ possono essere penalizzate dall'uso degli adattatori

Architettura esagonale e Layers

- Abbiamo detto che l'architettura esagonale ha origine nell'architettura a strati
  - ogni servizio è basato su due "strati"
    - sostiene un isolamento delle responsabilità di business del servizio dalle sue responsabilità di presentazione e infrastrutturali
  - c'è un'importante differenza con l'architettura a strati
    - nell'architettura a strati, gli strati possono essere allocati a team di sviluppo separati (team mono-funzionali)
    - nell'architettura esagonale, sono i servizi applicativi che vengono allocati a team di sviluppo separati (team cross-funzionali, che si occupano di interi servizi)

* Discussione

- L’architettura esagonale
  - nell’ambito di un singolo servizio, sostiene un isolamento delle responsabilità di business da quelle di presentazione e infrastrutturali
  - può essere applicata nella realizzazione di sistemi distribuiti – ad es., nei microservizi
  - è compatibile con DDD – ogni esagono può essere utilizzato per implementare un Bounded Context
  - è un’architettura flessibile, compatibile con altri pattern – come l’architettura a servizi, i microservizi, gli eventi di dominio e CQRS