DESIGN PATTERNS

CREATIONAL PATTERNS

1. FACTORY METHOD

La classe genitore sceglie a quale classe figlia l'istanza da costruire appartiene

⇓

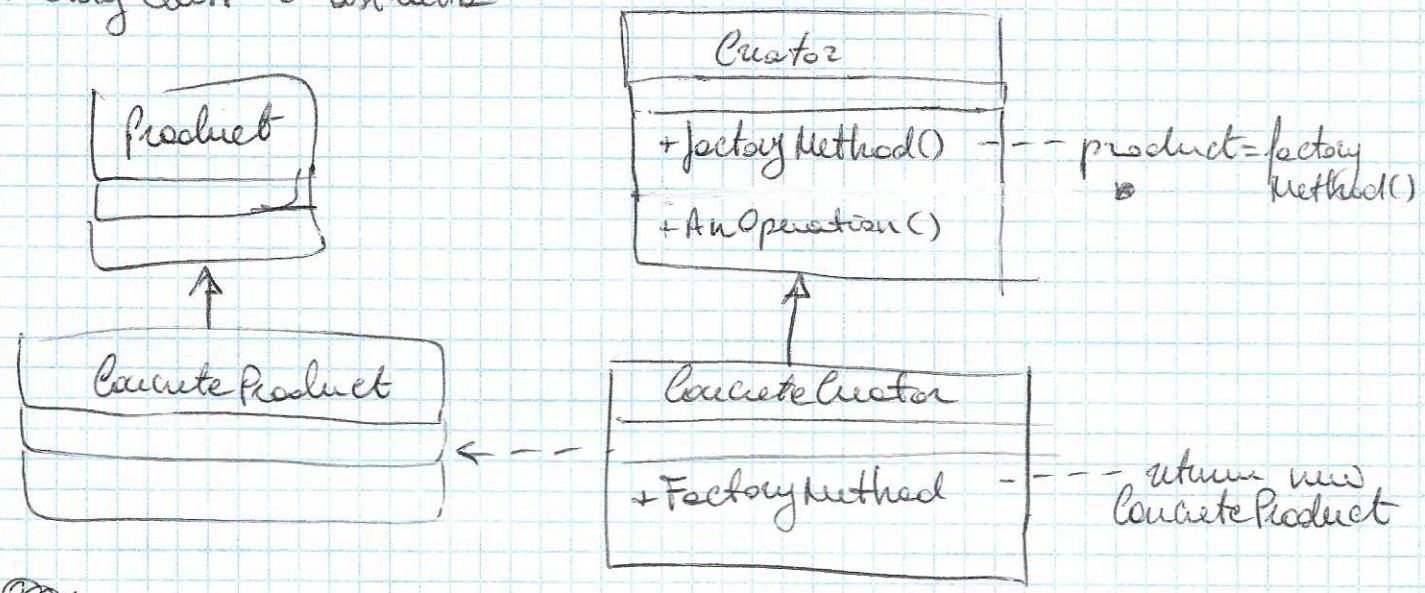
Nascondere la creazione in un metodo "factory": restituisce un oggetto di una classe senza essere costruttore di quella classe.

1.1. ABSTRACT FACTORY

Rispetta il principio open/closed, non devo aggiungere un metodo per ogni classe figlia.

⇓

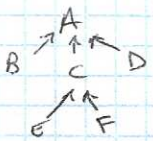
La Factory Class è astratta



STRUCTURAL PATTERNS

2. COMPOSITE

Trattare oggetti individuali e compositi allo stesso modo. Rappresentare oggetti in gerarchie parte-tutto. Ogni oggetto estende una stessa classe astratta, indipendentemente se è parte o tutto



3. ADAPTER

Permettere ad un client di collaborare con un fornitore di servizi con un'interfaccia non compatibile con la sua. Utilizzare il fornitore senza modificarlo.

Inserire una classe intermedia che espone un'interfaccia compatibile con il client

4. FAÇADE

Permettere l'accesso ad un sottosistema composto da diversi elementi agendo con la sua interfaccia in modo da disaccoppiare i client dai fornitori dei servizi

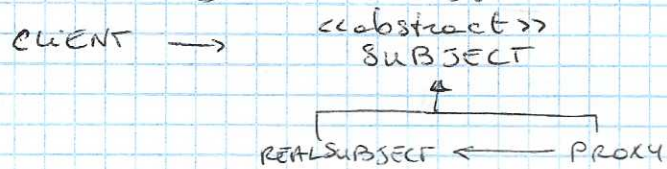
Fornire una classe che ha un'unica interfaccia e che ha accesso a tutti gli altri elementi favorendo un approccio stratificato

5. PROXY

Gestire l'accesso ad un oggetto con il fine di:

- fornire una descrizione locale di un oggetto di un altro spazio degli indirizzi
- gestire accessi ad oggetti persistenti
- gestire transazioni
- proteggere l'oggetto da accessi indesiderati.

Fornire un surrogato dell'oggetto di cui si vuole gestire l'accesso



BEHAVIORAL PATTERNS

6. OBSERVER

Notificare i cambiamenti di stato di un oggetto ad altri oggetti che devono aggiornare il loro stato

Fornire un surrogato dell'oggetto di cui si vuole gestire l'accesso

7. STRATEGY

Ci sono diversi algoritmi per risolvere un problema.

Inserire una classe astratta che funge da genitore per diverse classi concrete, una per ogni versione dell'algoritmo

8. COMMAND

Tenere traccia della sequenza di comandi lanciati da un client e dei risultati.

Incapsulare le richieste in un oggetto e le risposte in un altro

9. STATE

Un oggetto deve cambiare il comportamento in base allo stato.

Separare l'oggetto dal suo comportamento in un altro oggetto

10. VISITOR

Object structure che cambia raramente, ma spesso vengono definite

SOLID

S: Single Responsibility Principle: una classe deve avere una sola responsabilità

O: Open/Closed Principle: le entità devono essere aperte alle estensioni ma chiuse alle modifiche
↓
comportamento modificato senza alterare il codice sorgente

L: Liskov Substitution Principle: oggetti devono poter essere sostituiti con istanze dei loro sottotipi senza alterare la correttezza del programma

I: Interface Segregation Principle: molte interfacce specifiche sono meglio di una generale

D: Dependency Inversion Principle: componenti di alto livello non devono dipendere da componenti di basso livello, ma entrambi da astrazioni.
↓
I dettagli devono dipendere dalle astrazioni, non viceversa

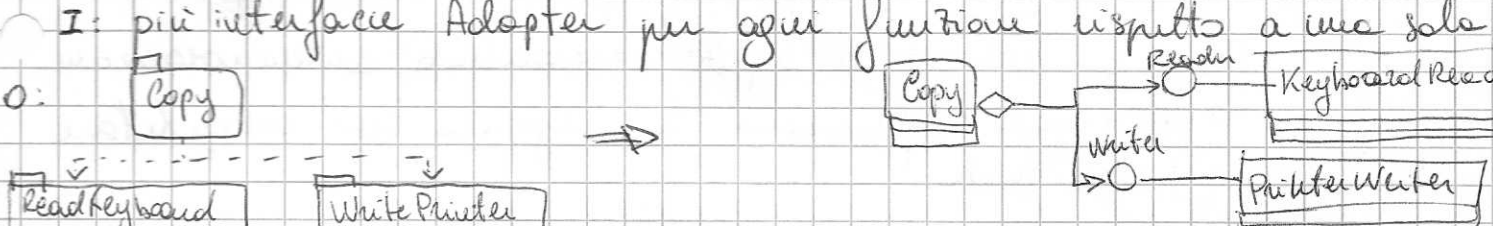
ESEMPI:

S: classe che rappresenta rettangolo e lo disegna
↳ diviso in due classi ⟷ rettangolo matematico
classe rettangolo GUI

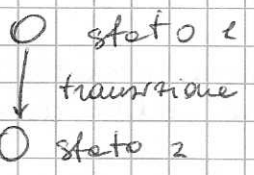
O: classe "Forma" il cui metodo drawAll disegna le forme in base al tipo → se aggiungo una nuova forma devo modificare la classe
↓
devo rendere più generale il metodo, non dipendere dai sottotipi

L: legato al corretto uso di astrazione e polimorfismo.
I metodi setHeight e setWidth di una classe Quadrato sono diversi da quelli della classe Rettangolo. Quadrato non può essere figlia di Rettangolo. Meglio entrambe figlie di Shape (astratta)

I: più interfacce Adapter per ogni funzione rispetto a una sola

O: 

AUTOMA: Rappresentazione del sistema, evoluzione da uno stato all'altro



AUTOMA a STATI FINITI $FA = \langle Q, q_0, F, E, T \rangle$ basato su $S = \langle Q, q_0, F \rangle$

E: insieme di etichette di transizioni

T: $Q \times E \times Q$ insieme di transizioni

Se $T \subseteq Q \times E \times Q$ → AUTOMA non DETERMINISTICO
↓ relazione

se $T \subseteq Q \times E \Rightarrow Q$ → AUTOMA DETERMINISTICO
↓ funzione

USABILITÀ di un'INTERFACCIA: misura con cui un prodotto può essere usato da specifici utenti per raggiungere specifici obiettivi con ~~successo~~ efficacia, efficienza, soddisfazione

- Efficacia: accuratezza e completezza. Devo essere in grado di svolgere il compito prefissato
- Efficienza: output su input - Limitare i costi e parità di compito ottenuto.
- Soddisfazione: Comfort e accettabilità dell'interfaccia.

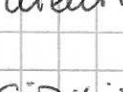
- Naturalezza: pochi cambiamenti. Utente concentrato sul compito no su come usare il sistema
- Completezza: l'interfaccia deve permettere di accedere a tutti i task, per ogni categoria di utente, in ogni fase dell'interazione. Non deve portare a process chiusi.
- Consistenza: un dialogo è consistente quando l'esperienza acquisita dall'utente con l'uso di una parte del sistema non viene contraddetta in altre parti
- Non complessità: operazioni eseguite con una certa efficienza.
  - ↳ visivo: numero di oggetti visualizzati contemporaneamente
  - ↳ funzionali: operazioni da eseguire per completare un task

- Non ridondanza: bisogna inserire il minimo indispensabile di informazioni.
- Assistenza: fornire aiuto all'utente
- Flessibilità: interfaccia efficace, efficiente, confortevole anche per utenti con caratteristiche diverse.

ACCESSIBILITÀ: interfaccia progettata per l'utilizzo da parte di persone diversamente abili.

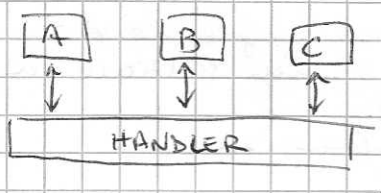
ARCHITECTURAL PATTERNS

MONOLITICA



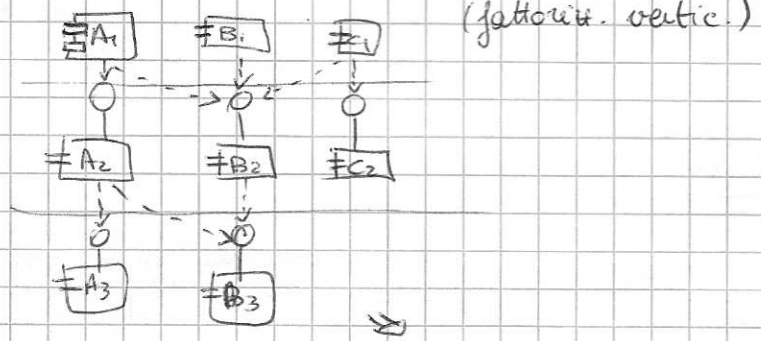
PRO: codice compatto
buone prestazioni
CONS: scarsa manutenibilità
di difficile progettazione

EVENT-DRIVEN: BROADCAST MODEL

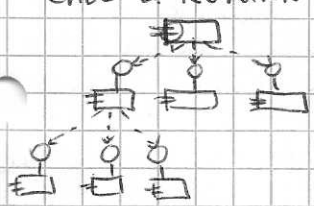


- Scalabili e modulari
- I componenti interagiscono tra loro senza conoscere l'indirizzo
- Potrebbero nascere dei conflitti
- I sottosistemi non sanno quando i loro eventi saranno gestiti

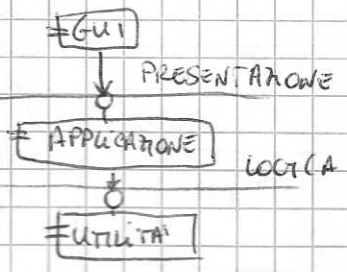
ARCHITETTURA STRATIFICATA (fattorizz. vertic.)



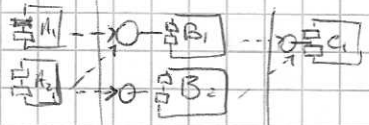
CALL & RETURN



3-LAYERS



DATA FLOW (fattorizz. orizzon.)

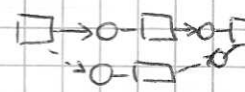


PRO: modulare e manutenibile facile da progettare per flussi di dati rilevanti
CONS: Architettura non scalabile
Non adatta per applicaz. interattive
(Sequenziali)

BATCH

L'output deve attendere la fine del flusso di input

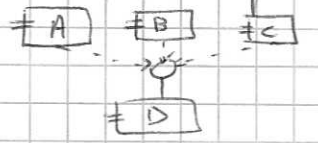
PIPES AND FILTERS



Componenti concorrenti

ARCHITETTURA CENTRATA SUI DATI

- Shared Repository

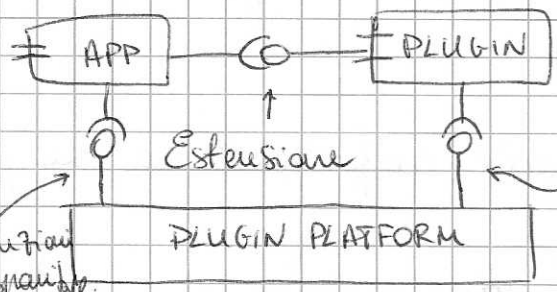


A richiede un update, il Rep. esegue e risponde
A invia una query, il Repos. risponde

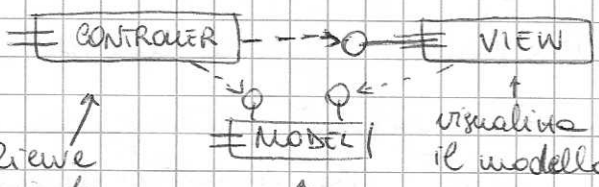
- Blackboard (active Repository)

A si registra presso il Repo, B chiede un update, il Repo esegue e notifica la modifica agli altri

ARCHITETTURA a PLUGIN



ARCHITETTURA MODEL-VIEW-CONTROLLER



Riceve richieste dall'utente e invia le richieste al modello

Comportamento e dati dell'applicazione
Risponde a View (richieste sullo stato) e a Controller (richieste di modifica dello stato)

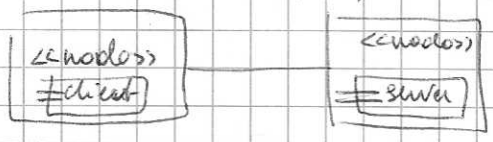
PUSH MODEL

1. La View si registra presso il model
2. L'utente invia una richiesta al controller
3. Il controller chiede un cambiamento di stato al Model
4. Il Model notifica alla view lo stato
5. Il Controller seleziona la View
6. Visualizzazione

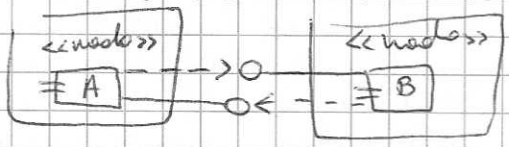
PULL MODEL

1. L'utente invia una richiesta al controller
2. Il controller chiede un cambiamento di stato al Model
3. La View chiede al Model lo stato
4. Il Model modifica lo stato
5. Il controller seleziona la vista
6. Visualizzazione

ARCHITETTURA TWO-TIERED

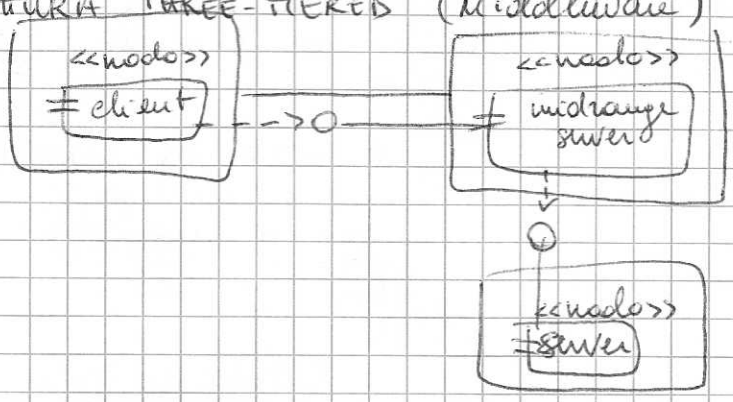


ARCHITETTURA PEER-TO-PEER

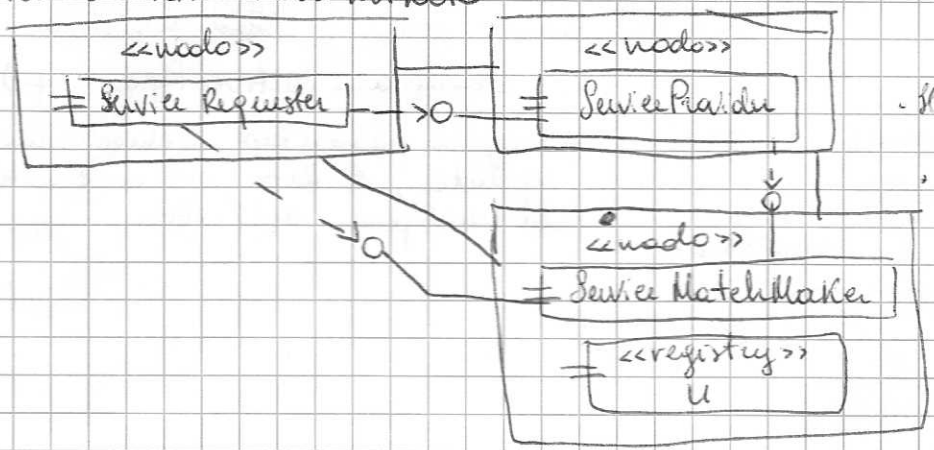


- Qui ogni nodo agisce come client e come server
- Le risorse cercate non sono presenti sul peer ma sono in un altro

ARCHITETTURA THREE-TIERED (Middleware)



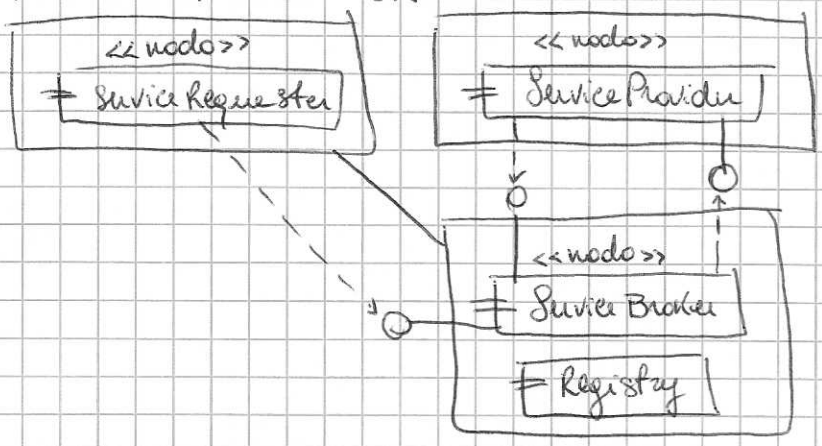
ARCHITETTURA MATCH MAKER



1 MatchMaker per n provider
1 Richiesta per 1 provider

- Il provider pubblica la sua descrizione e i suoi servizi
- Il requester richiede al matchmaker un servizio
- Il matchmaker fornisce l'indirizzo
- Il requester si collega al provider

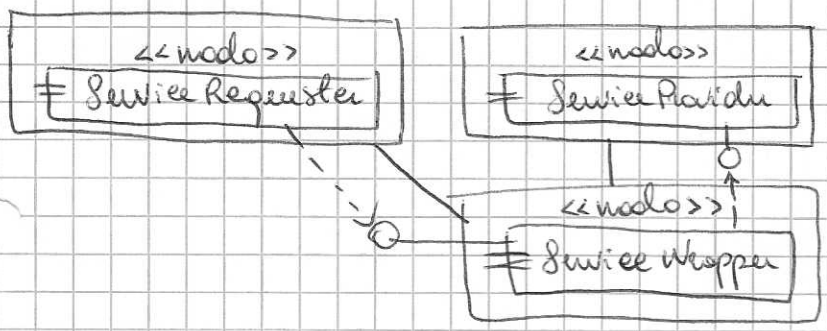
ARCHITETTURA BROKER



1 Broker per n providers
1 Richiesta per provider

- Il provider pubblica la sua descrizione e i suoi servizi
- Il requester chiede al broker un servizio
- Il broker invoca il servizio di un provider
- Il broker risponde al requester

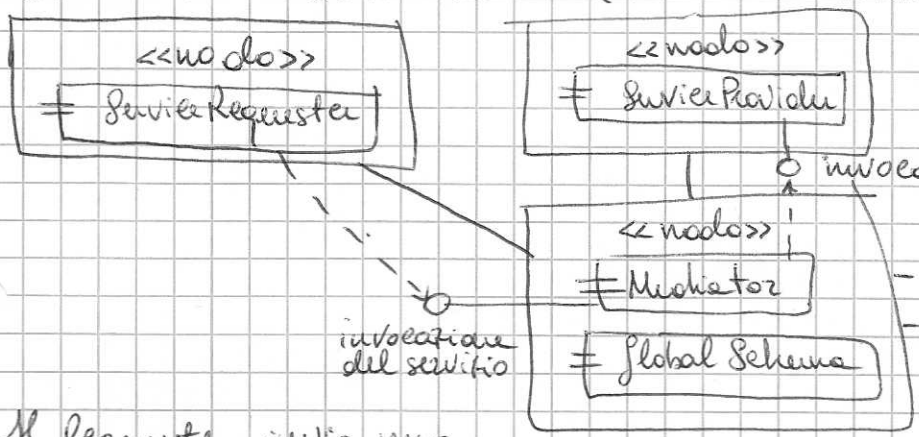
ARCHITETTURA WRAPPER



1 Wrapper per provider
1 per Richiesta per provider
- Requester invia richiesta a Wrapper
- Wrapper traduce la richiesta e la invia al provider
- Il provider risponde
- Traduzione e il Wrapper risponde al requester

ARCHITETTURE IBRIDE: composizione di più pattern, più utilizzate nella realtà

ARCHITETTURA MEDIATOR (BROKER + WRAPPER)



1 Mediator per n Provider
1 Richiesta per n Provider
- scomposizione richiesta
- composizione risposta

- Il Requester invia una richiesta al Mediator → Il Mediator scompone la richiesta in n sottorichieste secondo uno schema globale
  - Il Mediator invia le sottorichieste a n provider → i provider rispondono
  - Il Mediator compone le sottorisposte e la invia al requester