

Ingegneria del Software

Gabriel Pesce

2024/2025

Indice

1 Riassunto del corso	4
2 Introduzione	5
2.1 Diagrammi delle componenti e distribuzione	5
2.2 Diagramma delle componenti	5
2.2.1 Esempio di diagramma delle componenti	6
2.3 Diagramma di distribuzione	6
2.3.1 Elementi principali del diagramma di distribuzione	6
2.3.2 Esempio di uso di «manifest»	7
2.4 CBSE, SOSE e Microservizi	7
2.5 CBSE - Component Based Software Engineering	7
2.5.1 Componenti essenziali del CBSE	8
2.5.2 Principi di progettazione	8
2.5.3 Elementi basici di un modello di componente	8
2.5.4 Processi del CBSE	9
2.5.5 Processi di supporto del CBSE	9
2.5.6 CBSE per il riuso	9
2.5.7 CBSE con riuso	10
2.5.8 Problemi ancora aperti con CBSE	10
2.6 SOSE - Service Oriented Software Engineering	11
2.6.1 Standard di SOSE	11
2.6.2 Illustrazione dell'approccio SOSE	11
2.6.3 Servizi per il riuso	12
2.6.4 Servizi con riuso	12
2.6.5 CBSE vs SOSE	12
2.7 MS - Microservices Software Engineering	13
2.8 Cloud Computing	14
2.8.1 La gerarchia del Cloud	14
3 Progettazione dell'architettura Software e Design Principle	15
3.1 Analisi Architettonicale	15
3.1.1 Collezione e organizzazione dei requisiti non funzionali	15
3.1.2 Gerarchia di obiettivi per le decisioni architettoniche	15
3.1.3 Principi di base della progettazione architettonica	16
3.1.4 L'architettura logica	16
3.1.5 I pattern	16
3.1.6 Problemi nelle architetture software	17
3.1.7 Il Pattern architettonico "Layers"	17
3.1.8 Esempio di architettura a layers	18
3.1.9 Considerazioni sull'Application Layer	18
3.1.10 Esempio di diagramma di sequenza	19
3.1.11 Collaborazione tra pattern Layers e design pattern	20
3.1.12 Funzionamento del Facade	20
3.1.13 Funzionamento del Controller	21
3.1.14 Funzionamento di Observer	22
3.1.15 Model-View Separation	23
3.1.16 Architettura a strati rilassata	23
3.1.17 Layer sovrapposti	23
3.1.18 Layers e partizioni	24
3.1.19 Architettura 3-tier	24
3.2 Design Principles e Design Patterns	25
3.2.1 Principi del Design Orientato agli Oggetti	25
3.2.2 Open Closed Principle - OCP	25
3.2.3 Liskov Substitution Principle - LSP	25
3.2.4 Dependency Inversion Principle - DIP	26
3.2.5 Interface Segregation Principle - ISP	26
3.2.6 Package Cohesion Principles	26
3.2.7 Release Reuse Equivalency Principle - REP	26

3.2.8	Common Closure Principle - CCP	26
3.2.9	Common Reuse Principle - CRP	26
3.2.10	Package Coupling Principle	27
3.2.11	Acyclic Dependencies Principle - ADP	27
3.2.12	Stable Dependencies Principle - SDP	27
3.2.13	Stable Abstraction Principle - SAP	27
3.3	Pattern of Enterprise Applications Architectures	29
3.3.1	Design Patterns e Architetturali	29
3.3.2	Tipologie di Pattern Architetturali	30
3.3.3	Frameworks	30
3.4	Web Presentation Patterns	31
4	Metriche, Understand, Refactoring e Antipattern strutturali	32
4.1	Metriche Object Oriented	32
4.1.1	Line of code per class (LOCC)	32
4.1.2	Weighted method per class (WMC)	33
4.1.3	Depth of the inheritance tree (DIT)	33
4.1.4	Number of children (NOC)	33
4.1.5	Coupling between object classes (CBO)	34
4.1.6	Response for a class (RFC)	34
4.1.7	Lack of cohesion in methods (LCOM)	34
5	Esercitazioni	36
5.1	Design Pattern State	36
5.1.1	Esempio di Pattern State	37
5.1.2	State vs Strategy	37
5.2	Design Pattern Command	38
5.2.1	Esempio di Design Pattern Command	39
5.3	Design Pattern Template	40
5.3.1	Struttura del pattern Template	40
5.3.2	Esempio di Design Pattern Template	41
5.4	Design Pattern Repository	42
5.5	Failover	43
5.5.1	Accesso alla cache locale	44
5.6	Design Pattern Proxy	45
5.6.1	Ultime considerazioni sul Proxy	46
5.7	Design Pattern Convert Exceptions	47
5.8	Design Pattern Centralized Error Logging	48
5.9	Design Pattern Error Dialog	48
5.10	Design Pattern Abstract Factory	49
5.11	Design Pattern Do It Myself	50
5.12	Progettazione di Framework di persistenza dei dati	51
5.12.1	Proprietà dei framework	51
5.12.2	Servizio di persistenza	52
5.12.3	Mapping	53
5.12.4	Design Pattern Representing Objects as Tables	53
5.12.5	Identificazione univoca di oggetti	53
5.12.6	Design Pattern Object Identifier	53
5.12.7	Accesso a un Servizio di Persistenza	54
5.12.8	Design Pattern Facade	54
5.12.9	Facade vs Adapter	54
5.12.10	Materializzazione e Dematerializzazione	55
5.13	Active Record - Direct Mapping	56
5.14	Database Mapper - Indirect Mapping	57
5.15	Materializzazione in cache	58
5.15.1	Design Pattern Cache Management	58
5.16	Transazioni e Stati transazionali	60
5.16.1	Modellazione delle operazioni di una transazione	62
5.16.2	Lazy materialization con l'uso di Proxy	63
5.16.3	Design Pattern Virtual Proxy	63
5.16.4	Eager vs Lazy Materialization	64

1 Riassunto del corso

Il corso di Ingegneria del Software approfondisce tematiche viste nel corso di Analisi e Progettazione del Software, ovviamente introducendo nuovi concetti mai analizzati nel corso del secondo anno.

Il corso è molto interessante, soprattutto perchè, rispetto ad APS, viene trattato molto il lato pratico, quindi non ci si sofferma a definire le classi in Java ma anche i loro comportamenti e relazioni con le altre classi. Il mio consiglio è quello di non vedere il corso come un APS 2.0, solo la parte introduttiva rispolvera le vecchie nozioni, per il resto sono argomenti mai trattati sino ad'ora.

I professori del corso sono:

1. Per la teoria: **Francesca Arcelli Fontana**
2. Per le esercitazioni: **Oliviero Riganelli**

In questo file che metto a disposizione riassumerò tutta la teoria del corso + le esercitazioni, consiglio sempre di accompagnare il file con le slide che mettono a disposizione i prof, (**soprattutto la parte di esercitazione**), dove ho scritto la teoria, ma non ho inserito alcun codice se non i diagrammi fondamentali, può accadere che mi sia perso qualcosa per strada sicuramente...

Riguardo la modalità di esame, è simile al preappello di APS, quindi un progetto (ovviamente più esteso) + l'orale, i prof vi daranno indicazioni a riguardo.

Detto questo, in bocca al lupo se sceglierete questo corso e auguro a tutti

una vita meravigliosa

2 Introduzione

2.1 Diagrammi delle componenti e distribuzione

Sono diagrammi usati per poter raggruppare il codice in moduli e evitare le dipendenze tra loro (low coupling). Sono simili ai diagrammi delle classi, solo che invece di contenere le classi contengono rispettivamente i componenti e nodi.

I **nodi** sono degli elementi fisici che possiedono memoria e capacità di elaborazione, vengono usati per modellare la tipologia dell'hardware, la distribuzione di componenti, sistemi client-server o un sistema embedded. Si possono raggruppare in package e all'interno di questi si possono specificare relazioni di dipendenza, generalizzazione, associazione e aggregazione.

I **componenti** sono dei moduli di codice che partecipano all'esecuzione di un sistema, questi vengono eseguiti dai nodi (dato che esistono già a run-time). Come le classi, hanno un nome e possono realizzare più interfacce. Le componenti dispongono di operazioni raggiungibili solo attraverso le loro interfacce.

Ricorda...

- Le **classi** sono **astrazioni logiche**.
- I **componenti** sono **oggetti fisici**, quindi l'implementazione delle classi.

2.2 Diagramma delle componenti

Come accennato prima, un componente è un modulo (o una parte modulare) di un sistema che incapsula i suoi contenuti rendendoli invisibili all'esterno (e quindi inaccessibili). I componenti possono essere sostituiti o aggiornati senza influenzare l'intero sistema, garantendo flessibilità e manutenibilità.

I diagrammi delle componenti definiscono il comportamento delle interfacce fornite e richieste.

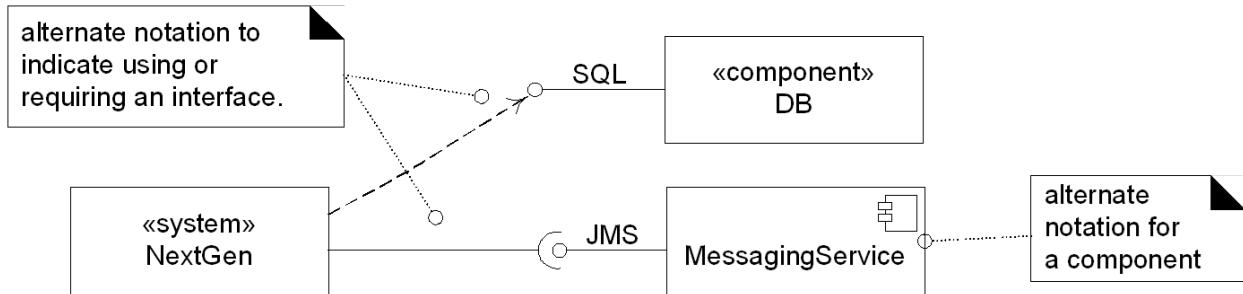
L'obiettivo di questo diagramma è quello di enfatizzare l'importanza delle interfacce e di come i componenti siano vantaggiosi per i motivi spiegati sopra (indipendenza, modularità, riusabilità, rimpiazzabili senza influenzare il sistema...).

I componenti possono definire sistemi software di dimensione e complessità arbitraria. I diagrammi UML delle componenti permettono di modellare i componenti software e le interfacce annesse.

Si parla di **dipendenza (dependency)** tra due elementi quando alla modifica di uno dei due a sua volta potrebbe modificare anche l'elemento collegato.

Spesso si fa riferimento a questo diagramma col nome di **Wiring Diagram**.

2.2.1 Esempio di diagramma delle componenti



L'immagine sopra mostra come può essere dichiarato un componente in un diagramma UML, ovvero tramite notazione «component» oppure inserendo un simbolo sul lato destro. Altra cosa da ricordare è il collegamento tra il sistema e il componente, in base al tipo di freccia, indica se il componente sta **usando** (NextGen usa il DB) o **richiedendo** (NextGen richiede il MessagingService) l'**interfaccia**.

2.3 Diagramma di distribuzione

I diagrammi di distribuzione (o deployment) mostrano l'assegnazione degli artefatti software concreti (es. file eseguibili) ai nodi computazionali (es. servizi di elaborazione).

Mostrano la distribuzione degli elementi software all'interno dell'architettura fisica e la comunicazione tra questi.

L'obiettivo di questo diagramma è quello di comunicare l'architettura fisica o di distribuzione.

2.3.1 Elementi principali del diagramma di distribuzione

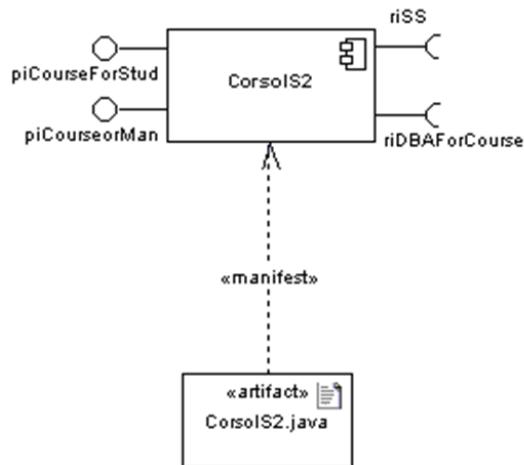
Nodo dispositivo	Risorsa fisica con servizi di memoria e processazione per eseguire il software.
Nodo dell'ambiente di esecuzione	Risorsa software che viene eseguita all'interno di un nodo esterno (es. computer) e fornisce un servizio per ospitare e eseguire un altro software eseguibile.
Percorsi di comunicazione	Connessione tra i nodi.

Esempi di diagrammi di distribuzione posso essere degli artefatti, ovvero elementi fisici, come ad esempio **file eseguibili** (file JAR, .exe, script...) oppure **data files** (HTML, XML...).

Un artefatto **“manifesta”** uno o più elementi di un modello. Con il termine manifestare si intende che l'artefatto rappresenta l'oggetto fisico di uno o più elementi del modello. Questi elementi solitamente sono dei componenti.

2.3.2 Esempio di uso di «manifest»

Per rappresentare una manifestazione si usa una linea tratteggiata con freccia aperta etichettata con la keyword «manifest»



2.4 CBSE, SOSE e Microservizi

In linea generale sono tre approcci all'ingegneria del software, ciascuno con diverse caratteristiche.

2.5 CBSE - Component Based Software Engineering

Le parole chiave che determinano questo approccio sono la riusabilità e l'uso di componenti software.

Questo approccio nasce dal momento che molto spesso lo sviluppo orientato agli oggetti fallisca per mancanza di riuso dei componenti, ciò è causato da vari motivi:

1. Le classi sono troppo dettagliate e specifiche per poter essere riutilizzate in altri contesti.
2. I componenti sono più astratti delle classi stesse.
3. I componenti vengono visti come dei provider di servizi indipendenti (stand alone).
4. I componenti possono esistere come delle entità indipendenti.

L'approccio CBSE fa sì che i componenti forniscano una funzionalità senza dover considerare dove il componente viene eseguito o il suo linguaggio di programmazione, questo perché è un'entità eseguibile e indipendente che può essere usato per uno o più oggetti eseguibili. L'interfaccia del componente è pubblica e tutte le interazioni avvengono tramite questa.

Come per i diagrammi, questo approccio dispone di due tipi di interfacce:

Interfacce Fornite	Definiscono i servizi forniti dal componente verso altri componenti. Essenzialmente, è l'API che definisce i metodi che possono essere chiamati dall'utente.
Interfacce Richieste	Definiscono i servizi richiesti dal componente per eseguire quanto specificato. Ciò non compromette l'indipendenza o la distribuzione del componente perché l'interfaccia richiesta non definisce come questi servizi vengono forniti.

2.5.1 Componenti essenziali del CBSE

1. Componenti indipendenti specificati dalle loro interfacce.
2. Standard specifici dei componenti per facilitare la loro integrazione.
Stabiliscono come i componenti comunicano e operano tra loro.
3. Middleware che fornisce supporto per la portabilità dl componente.
4. Un processo di sviluppo mirato al riuso.

2.5.2 Principi di progettazione

1. I componenti sono indipendenti, non interferiscono tra loro.
2. L'implementazione dei componenti è nascosta.
3. La comunicazione avviene tra interfacce ben definite.
4. Le piattaforme dei componenti sono condivise in modo da ridurre i costi di sviluppo.

I componenti sviluppati tramite diversi approcci NON lavorano tra loro.

2.5.3 Elementi basici di un modello di componente

Interfacce	Il modello dei componenti specifica come le interfacce devono essere definite e i suoi elementi (operazioni, nomi, parametri...).
Uso	Per essere distribuiti e accessi in remoto, ciascun componente deve avere un nome univoco.
Distribuzione	Il modello dei componenti include una specifica di come i componenti devono essere impacchettati per la loro distribuzione come entità indipendenti e eseguibili.

2.5.4 Processi del CBSE

I processi del CBSE sono, come dice il nome, dei processi software che supportano l'utilizzo di tale approccio. Permettono la riusabilità delle attività di processo coinvolte nelle attività di sviluppo e un riuso dei componenti.

Bisogna fare una distinzione in merito alla modalità di sviluppo e il riuso:

Sviluppo PER il riuso	Si basa sullo sviluppo di componenti che verranno poi utilizzati in altre applicazioni.
Sviluppo CON riuso	Si basa sullo sviluppo di nuove applicazioni usando dei componenti già esistenti.

2.5.5 Processi di supporto del CBSE

Acquisizione del componente	Processo che acquisisce i componenti per il loro riuso o trasforma lo sviluppo in un componente riutilizzabile.
Gestione del componente	Verte sulla gestione dei componenti riusabili assicurandosi siano propriamente catalogati, immagazzinati e resi disponibili per il riuso.
Certificazione del componente	Processo che controlla se un componente soddisfa le sue specifiche.

2.5.6 CBSE per il riuso

Si concentra sullo sviluppo dei componenti. Lo scopo è quello di generalizzare quei componenti che vengono creati per specifiche applicazioni con lo scopo di renderli riutilizzabili.

Un componente ha più probabilità di essere riutilizzato se è associato a un'astrazione di dominio stabile, ovvero a un oggetto di business ben definito e costante nel sistema.

Per fare un esempio di dominio stabile può essere un ospedale, dove i componenti hanno ciascuno degli scopi fondamentali (dottori, pazienti, trattamenti ecc...)

2.5.7 CBSE con riuso

Al contrario della prima tipologia, questa cerca e integra componenti già esistenti. Quando si riutilizzano i componenti in un progetto software, è fondamentale considerare i compromessi tra i requisiti ideali di ciò che si desidererebbe avere e i servizi reali forniti dai componenti disponibili.

Questo approccio richiede:

1. Sviluppo dei requisiti preliminari (essenziali).
2. Ricerca dei componenti e la loro modifica in base alle funzionalità disponibili.
3. Cercare nuovamente per trovare componenti migliori che soddisfano i requisiti.
4. Unire (o comporre) i componenti per creare il sistema.

2.5.8 Problemi ancora aperti con CBSE

1. Quanto è affidabile un componente senza un codice sorgente disponibile?
2. Chi certifica la qualità dei componenti?
3. Come possono essere predette le proprietà emergenti della composizione dei componenti?
4. Come si fanno le analisi tra le caratteristiche di un componente con un altro?

2.6 SOSE - Service Oriented Software Engineering

Le parole chiave che determinano questo approccio sono la riusabilità e l'uso di servizi software (per il CBSE era l'uso dei componenti software).

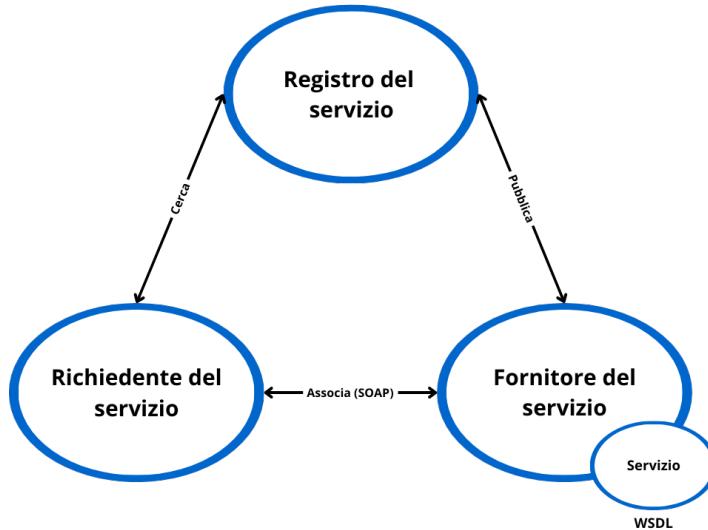
Questo approccio nasce con lo scopo di fornire lo stesso servizio a più applicazioni o utenti, questo perché i servizi sono indipendenti tra loro.

Per servizio si intende un'entità software con basso accoppiamento (loosely-coupled) che incapsula delle funzionalità che possono essere distribuite e accessse.

2.6.1 Standard di SOSE

SOAP	Protocollo di accesso agli oggetti che definisce un'organizzazione per lo scambio di dati strutturati tra i web services.
WSDL	Definisce come possono essere rappresentate le interfacce dei web services.
UDDI	Standard di ricerca che definisce come possono essere organizzate le informazioni di descrizione dei servizi, utilizzate dai richiedenti dei servizi per trovarli.

2.6.2 Illustrazione dell'approccio SOSE



Come per CBSE, anche SOSE distingue, nel suo caso, i servizi PER il riuso e i servizi CON riuso:

2.6.3 Servizi per il riuso

Come dice il titolo, questi servizi vengono sviluppati con lo scopo di essere riutilizzati all'interno di applicazioni orientati ai servizi. Questi servizi devono essere progettati come delle 'astrazioni riutilizzabili' che possono essere usati in sistemi diversi, in modo da poter garantire un servizio robusto e affidabile. Alla fine, il servizio deve essere ben documentato in modo da poter essere scoperto e capito da potenziali utenti.

2.6.4 Servizi con riuso

I servizi in questo caso vengono visti come dei componenti riutilizzabili. In questo modo, i servizi possono essere forniti localmente o esternamente verso altri fornitori. Altro vantaggio è che i servizi sono indipendenti dal linguaggio in cui vengono scritti, e per concludere l'investimento fatto in sistemi informatici obsoleti può essere preservato, conservandone quindi il valore attraverso aggiornamenti o modernizzazioni.

2.6.5 CBSE vs SOSE

La differenza sostanziale tra i due è che CBSE fa uso dei componenti, SOSE dei servizi.

1. I servizi sono indipendenti, i componenti no.
2. I servizi non possiedono alcuna interfaccia 'richiesta'.
3. La comunicazione tra i servizi avviene tramite messaggi in formato XML.

2.7 MS - Microservices Software Engineering

Si tratta di uno stile architetturale che struttura l'applicazione come una collezione di piccoli e contenuti componenti con basso accoppiamento. Questi componenti vengono anche chiamati servizi e implementano delle specifiche capacità di business.

Tra le caratteristiche dei microservizi troviamo:

1. Comunicazione attraverso protocolli leggeri (lightweight).
2. Sviluppati da team dedicati.
3. La distribuzione è indipendente.

L'obiettivo principale dei microservizi sarebbe quello di garantire scalabilità, affidabilità, eterogeneità tecnologica e aggiornamenti continui del codice, purtroppo, nella realtà ci troviamo di fronte a una complessa manutenibilità e una fase di testing difficile.

	SOSE	MS
Obiettivo	Riusabilità dei servizi	Garantire basso accoppiamento
Di cosa fa uso	Servizi: include tante funzionalità di business e spesso è implementato come un completo sottosistema	Micro-servizi: creati per servire solo una specifica funzionalità di business
Cosa succede alla modifica	Richiede la modifica del sistema monolitico, una modifica significativa	Richiede la creazione di un nuovo servizio
Comunicazione	Tramite ESB (Enterprise Service Bus)	Meno elaborata e semplice sistema di messaggi
Protocolli	Multipli protocolli per messaggi (SOAP)	Protocolli leggeri (HTTP, REST)
Distribuzione	Uso di una piattaforma comune per la distribuzione di tutti i servizi	Uso di una piattaforma cloud per la distribuzione
Container	Uso poco popolare (Docker o Kubernetes)	Uso molto popolare
Archiviazione dei dati	Condivisi tra i diversi servizi	Ogni micro-servizio può avere un proprio archivio indipendente

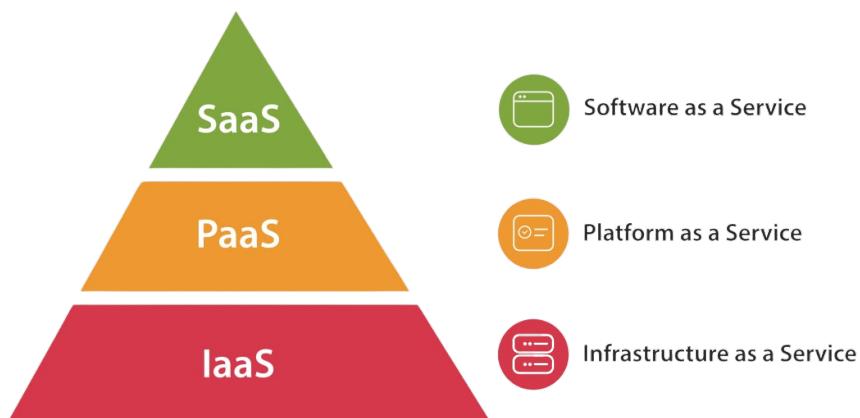
2.8 Cloud Computing

Nasce per mano del NIST (National Institute of Standard and Technology), si tratta di un modello capace di garantire un accesso via rete a una vasta area di risorse (reti, server, applicazioni, servizi ecc...) che possono essere rilasciate con la minima gestione o interazione con il fornitore del servizio.

Tra le caratteristiche principali abbiamo:

1. Capacità per un utente di registrarsi e ricevere i servizi senza dover affrontare lunghe attese.
2. Capacità di accedere ai servizi mediante piattaforme standard (desktop, laptop ecc...)
3. Le risorse sono messe in comune tra tutti gli utenti o clienti.
4. Scalabilità rapida per far fronte a un grande quantitativo di utenti.
5. La fatturazione del servizio è monitorato in base a quanto si sta usando quel servizio.

2.8.1 La gerarchia del Cloud



SaaS	Applicazioni rivolte agli utenti finali, inviati tramite il web
PaaS	Set di strumenti e servizi per creare codici e distribuire le applicazioni in modo rapido e efficiente
IaaS	Hardware e software che da vita a tutti i servizi (server, reti, sistemi operativi ecc...)

3 Progettazione dell'architettura Software e Design Principle

Il capitolo ha lo scopo di introdurre i design pattern per progettare l'architettura logica del sistema

3.1 Analisi Architetturale

L'analisi architetturale ha l'obiettivo di identificare e risolvere i requisiti non funzionali del sistema (es. la sicurezza) nel contesto dei requisiti funzionali. Ricordiamo il documento che definisce i requisiti non funzionali, ovvero le [specifiche supplementari](#).

I passi da seguire durante l'analisi sono:

1. **Investigazione:** identificazione di requisiti funzionali e non funzionali che hanno maggiore impatto sul sistema.
2. **Progettazione:** risoluzione dei requisiti identificati.

Per i requisiti che hanno impatto architettonico significativo si analizzano le alternative e si creano delle soluzioni per gestire i compromessi e le priorità.

3.1.1 Collezione e organizzazione dei requisiti non funzionali

Viene stilata la seguente tabella per i requisiti non funzionali:

Tabella dei fattori	
Nome fattore	Nome del fattore (requisito)
Misure e scenari di qualità	Come il fattore sarà misurato e valutato tramite metriche di qualità
Variabilità	Quanto il sistema è flessibile a soddisfare il requisito con possibilità di evoluzione in futuro
Impatto del fattore	Quanto influisce il fattore sull'architettura del sistema
Priorità per il successo	Quanto è importante il fattore per il successo del sistema
Difficoltà o rischi	Possibili difficoltà associate al soddisfacimento di questo requisito

3.1.2 Gerarchia di obiettivi per le decisioni architetturali

Esistono 3 tipi di obiettivi e priorità:

1. Vincoli inflessibili: sicurezza, conformità alle leggi...
2. Obiettivi di business: tipi di clienti finali interessati al prodotto software
3. Altri obiettivi: estensioni con altre funzionalità

3.1.3 Principi di base della progettazione architetturale

Sono dei principi che offrono maggiore efficienza all'architettura software:

Principi
Accoppiamento basso
Coesione alta
Variazione protetta
Separazione degli interessi e localizzazione dell'impatto
Utilizzo di pattern e stili architetturali

3.1.4 L'architettura logica

Questo tipo di architettura vede il sistema come un insieme di package logici. Descrive il sistema nei termini della sua organizzazione in layers, packages, frameworks, classi, interfacce e sottosistemi.

I **package** raggruppano un insieme di responsabilità coese (strettamente correlate tra loro), questa caratteristica viene definita anche come **modularizzazione**.

Questa caratteristica favorisce una separazione degli interessi (separation of concerns).

Ciascun package si occupa di quello per cui è stato progettato, senza preoccuparsi del resto del sistema.

3.1.5 I pattern

I pattern si dividono in diverse tipologie:

Pattern architetturali	Progettazione a larga scala (es. i pattern layers)
Design patterns	Progettazione a scala media-piccola che hanno connessioni fra gli elementi a larga scala (es. i pattern GoF)
Idiomi	Soluzioni progettuali di basso livello legate al linguaggio o alle implementazioni usate (es. comparazione di due stringhe, Singleton...)
Strategie	Direttive e consigli pratici su come risolvere un certo problema in modo efficace.

Per chiarire meglio la differenza tra i pattern architetturali e i design pattern ci si può riferire alle seguenti domande:

- Quali sono le parti fondamentali del sistema? → Pattern architetturali
- Come sono connesse tra loro? → Design patterns

3.1.6 Problemi nelle architetture software

Si possono verificare diverse problematiche durante lo sviluppo dell'architettura software:

1. Alto accoppiamento dei componenti, portano alla modifica di più elementi in caso di cambiamento.
2. La logica applicativa condivisa con le GUI.
3. I servizi tecnici direttamente collegati alla logica applicativa, sfavorendone il riuso.
4. L'evoluzione del sistema risulta difficoltosa dato l'alto accoppiamento e le numerose funzionalità distinte e non correlate tra loro.
5. Ovviamente possono esserci tante altre problematiche...

3.1.7 Il Pattern architettonico "Layers"

Questo pattern permette di risolvere i problemi elencati in precedenza tramite l'uso dei layers.

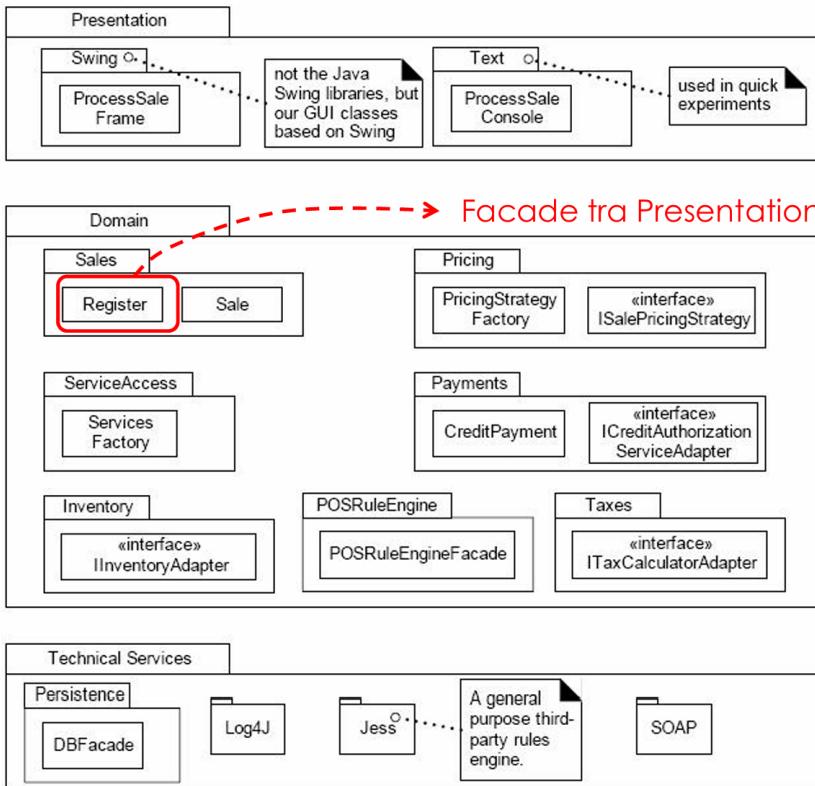
Un **Layer** è un elemento di grandi dimensioni, spesso composto da molti package e sottosistemi.

Il pattern permette di organizzare l'architettura logica in un sistema a più livelli (layers) che comprendono funzionalità distinte ma correlate in modo da permettere una netta separazione delle responsabilità e un aumento della coesione del sistema stesso.

I livelli più bassi comprendono i servizi più generali e di basso livello, quelli in alto saranno più specifici per la singola applicazione.

Il numero di layer e il loro scopo non è fisso, cambia a seconda delle applicazioni. Ciascun layer si appoggia alle funzionalità dei layer sottostanti, ma possono comunicare anche con i layer di livelli più bassi. Ovviamente, come il sistema stesso, anche l'architettura a layer viene sviluppata in modo iterativo, l'obiettivo è quello di avere l'architettura fondamentale stabilita.

3.1.8 Esempio di architettura a layers

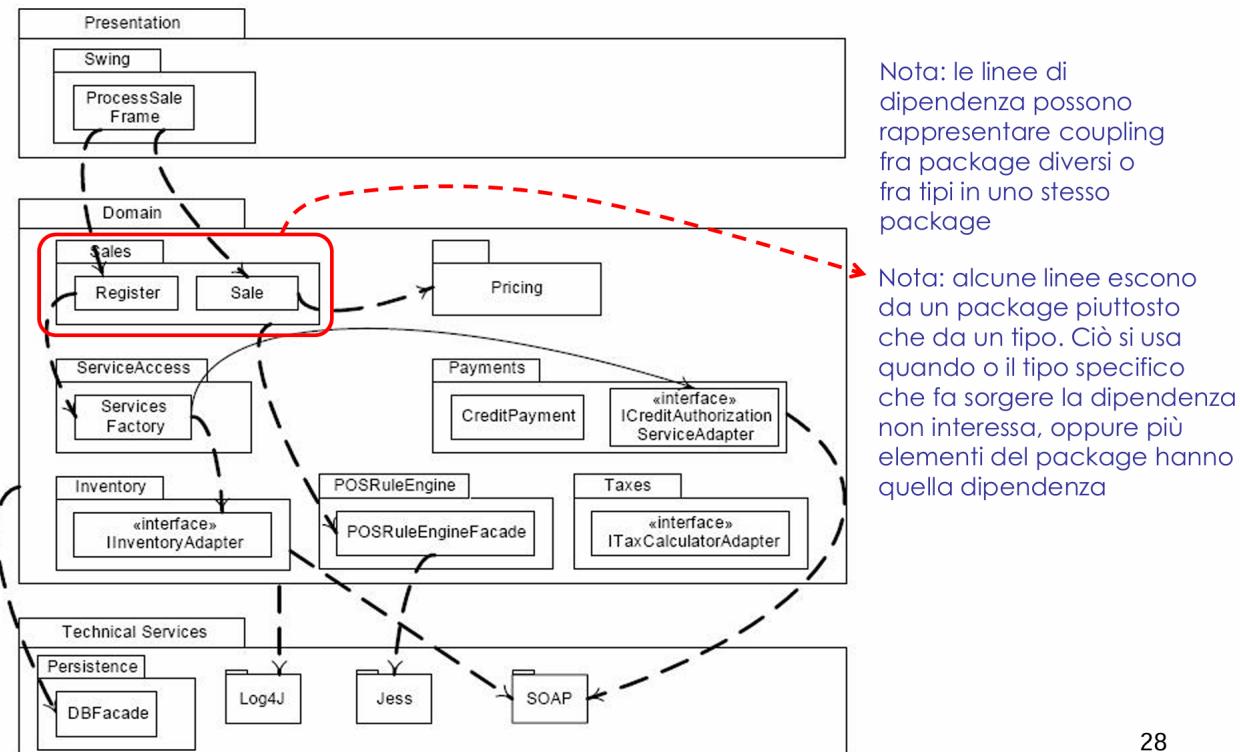


23

3.1.9 Considerazioni sull'Application Layer

- Per conoscere lo stato del client si fa uso dei package Presentation e Domain. I due package collaborano tra loro per controllare il flusso di lavoro (l'ordine delle finestre, le pagine web...)
- L'Application Layer viene usato quando:
 1. Sono disponibili più interfacce utente. In questo caso vengono usati degli Adapter per collezionare i dati delle varie interfacce e le Facade per nascondere l'accesso al Domain.
 2. Il sistema è distribuito e il Domain si trova su un nodo diverso rispetto al Presentation.
 3. Il Domain non può mantenere lo stato delle sessioni.
 4. Esiste un workflow ben definito da seguire nella presentazione delle varie schermate.

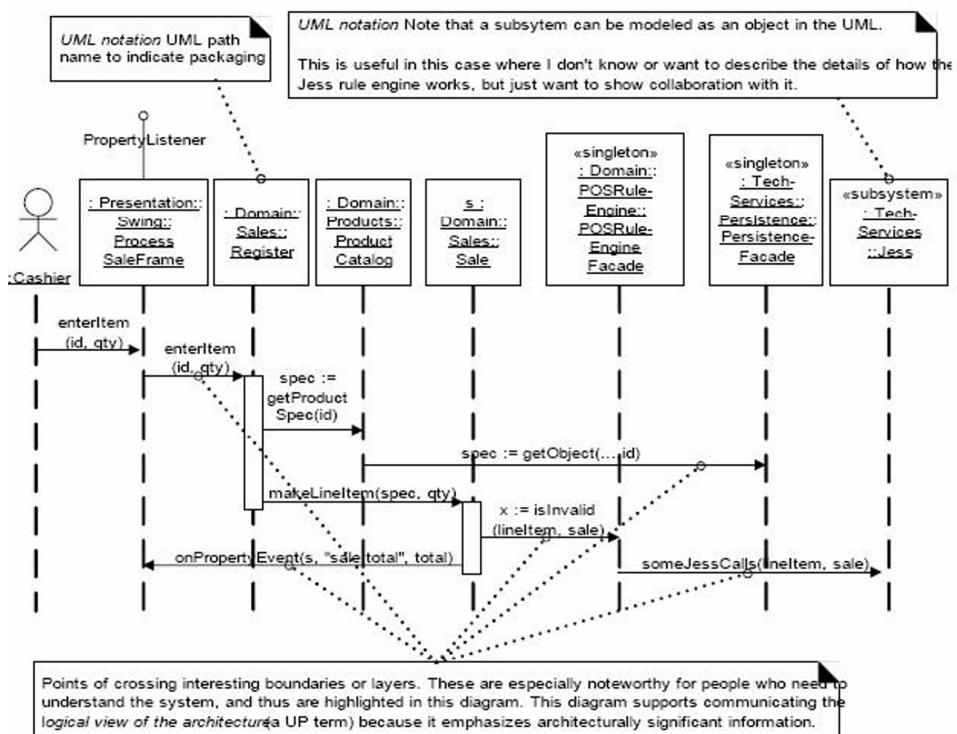
Ovviamente esisteranno accoppiamenti fra i vari livelli, ma si possono raffigurare i più significativi attraverso un apposito diagramma.



28

Per capire meglio come gli oggetti fra i vari layer comunicano fra di loro si possono usare i **Diagrammi di Sequenza** in modo da rappresentare gli scenari architetturali più significativi.

3.1.10 Esempio di diagramma di sequenza



Nel diagramma si possono fare delle considerazioni:

1. In UML si può illustrare il nome del package a cui una certa classe appartiene con la seguente dicitura: $<PackageName>::<ClassName>$. Ciò permette di evidenziare le connessioni fra package e layer all'interno di un diagramma dinamico.
2. Lo sterotipo «*Subsystem*» indica un'entità con il suo comportamento e le sue interfacce. Può essere modellato da un package o da un oggetto.
3. Il diagramma di sequenza mostra solo le relazioni rilevanti dal punto di vista architetturale.

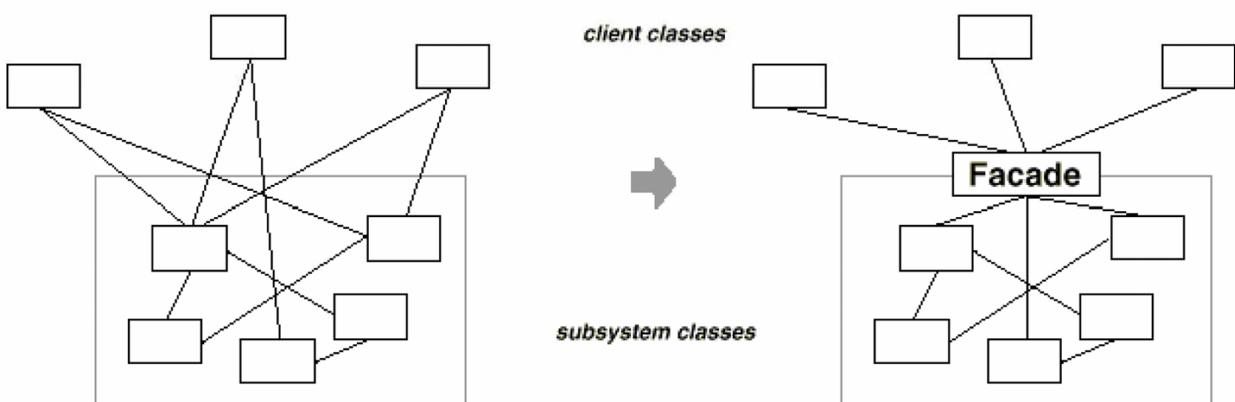
Bisogna precisare che alcuni package rappresentano veri e propri sottosistemi indipendenti (stereotipi), con il proprio comportamento e le loro interfacce. Altri package **NON** sono sottosistemi.

3.1.11 Collaborazione tra pattern Layers e design pattern

Come detto prima, il pattern architetturale serve per definire le parti più grandi del sistema. Per poter stabilire le connessioni tra gli stati e i package verranno usati i design pattern. Spesso sono usati il **Facade**, **Observer**, e **Controller**, quindi i pattern GRASP.

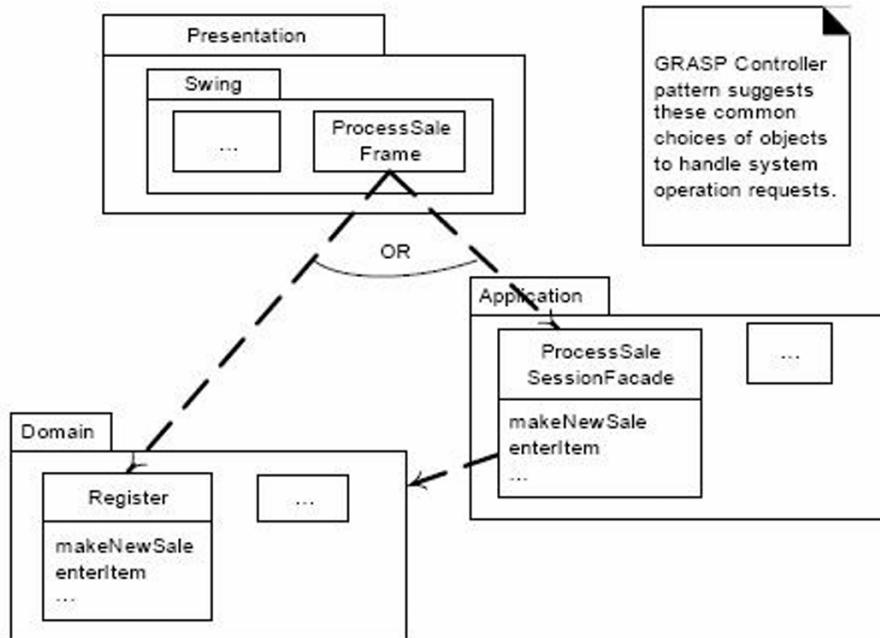
3.1.12 Funzionamento del Facade

Viene utilizzato per accedere alle funzionalità offerte da un sottosistema. Facade permette di nascondere il sottosistema dietro un'interfaccia unificata. Facade non espone le operazioni di basso livello, ma quelle di alto livello utilizzabili dai client.



3.1.13 Funzionamento del Controller

Viene utilizzato per descrivere le scelte comuni tipiche degli handler lato client per le richieste di operazioni di sistema provenienti dal Presentation Layer (UI).



Si tratta quindi del primo oggetto oltre lo stato UI che coordina le operazioni di sistema, in grado di ricevere e gestire un messaggio di un'operazione di sistema. Quando si parla di operazioni di sistema si intendono gli eventi di input principali del sistema.

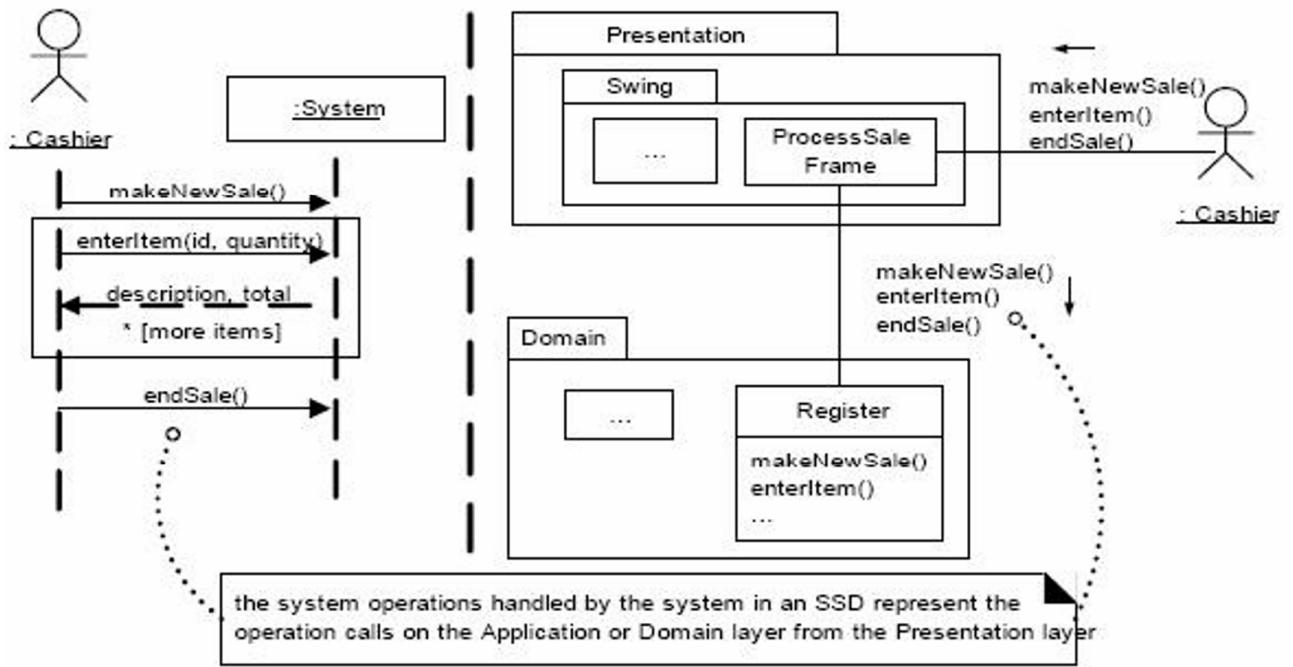
Tra i vantaggi del Controller abbiamo:

1. Logica applicativa non gestita nello strato di interfaccia.
2. Riuso della logica.
3. Si possono usare interfacce diverse.

Quando si usano i Controller, accade di compiere l'errore di "gonfiarlo":

- Un solo Controller riceve numerosi eventi di sistema.
- Il Controller svolge parte del lavoro prima di delegarlo.
- Il Controller ha numerosi attributi e conserva le informazioni sul sistema e sul dominio.

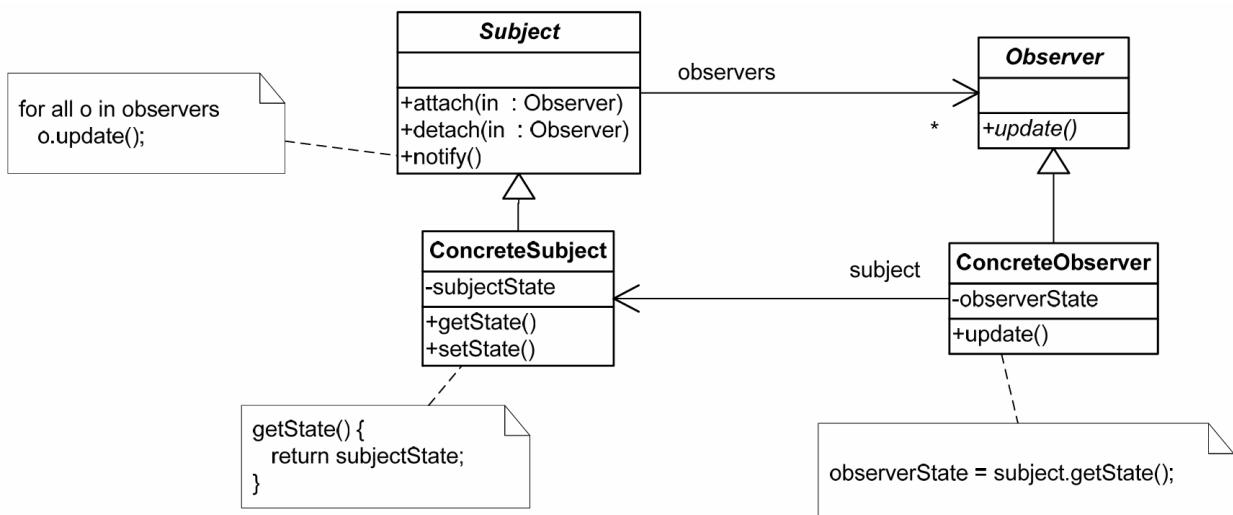
Un modo per vedere le operazioni di sistema è tramite i **Diagrammi di Integrazione**.



3.1.14 Funzionamento di Observer

Ipotizzando che il layer Application o Domain debbano comunicare con il Presentation, l'Observer è utile al caso.

Questo design pattern permette di notificare e aggiornare automaticamente tutti gli oggetti dipendenti da un'oggetto 'padre' che prende il nome di Subject.



3.1.15 Model-View Separation

Si tratta di un approccio che ha l'obiettivo di separare il layer Domain (Model) dal layer Presentation (View). Gli oggetti del Domain non devono avere conoscenza degli oggetti del Presentation.

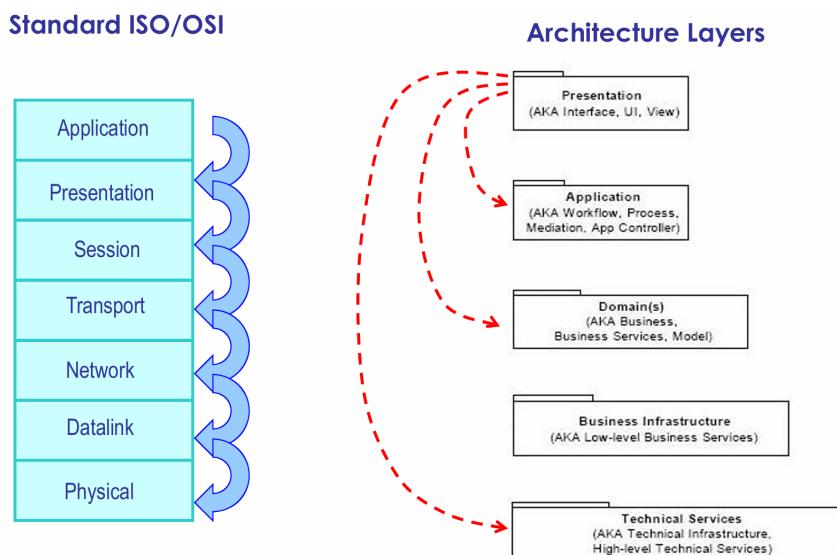
Esistono diverse motivazioni per usare questo approccio:

1. Si ha un modello coeso.
2. Sviluppo separato del modello e l'interfaccia grafica.
3. Permette di connettere facilmente nuove viste.
4. Permette l'esecuzione del modello indipendentemente dalla GUI.

Model-View Separation adotta una comunicazione verso l'alto.

3.1.16 Architettura a strati rilassata

Come accennato in precedenza, i livelli di un'architettura a layer non comunicano solo con il livello sottostante. Si parla quindi di struttura rilassata.



3.1.17 Layer sovrapposti

Alcuni oggetti appartengono univocamente a un solo layer.

Può capitare che alcuni oggetti, invece, non siano facilmente classificabili, come ad esempio i layer Domain e Business Infrastructure oppure Technical Services e Foundation che possono essere uniti sotto nome di Infrastructure Layer.

3.1.18 Layers e partizioni

Generalmente:

- Layers: linee verticali.
- Partizioni: divisioni orizzontali.

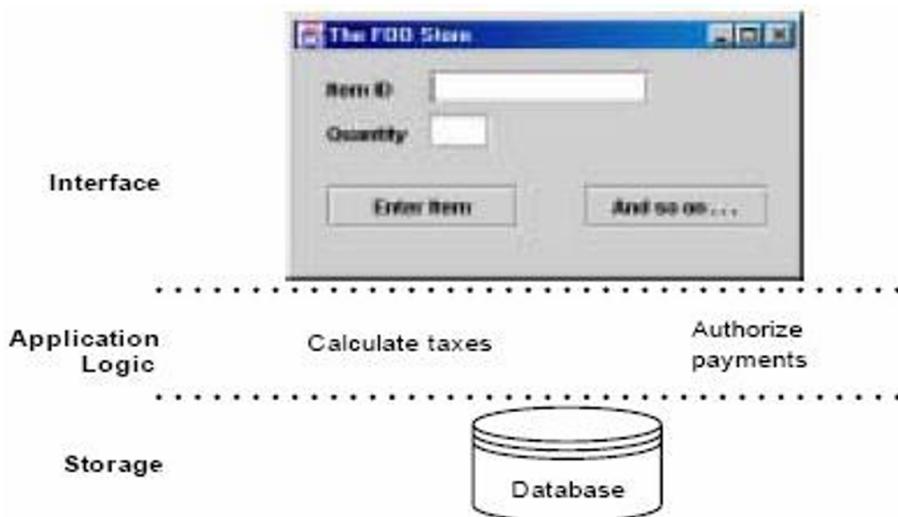
Risulta utile distinguerli per diversi motivi:

1. Separation of concerns tra i servizi di alto e basso livello.
2. Alcuni layers possono essere sostituiti nelle nuove implementazioni.
3. I layer più bassi implementano funzionalità riutilizzabili.
4. Viene favorito lo sviluppo in team dal momento che ciascuno sviluppatore è specializzato per un layer o servizio definito.

3.1.19 Architettura 3-tier

Tipo di architettura che divide il sistema in 3 componenti:

Interface	Interfaccia testuale, GUI e l'interazione diretta con l'utente
Application Logic	Insieme dei task che governano i processi
Storage	Meccanismo di memorizzazione persistente (es. Database)



3.2 Design Principles e Design Patterns

In questa sezione si vanno ad analizzare i problemi che portano a un design inefficiente (Rotting Design).

I sintomi sono i seguenti:

Rigidità	Alta difficoltà nell'effettuare cambiamenti nel software, anche se si tratta di cambiamenti semplici. Ogni cambiamento porta a sua volta la necessità di cambiare ulteriori moduli dipendenti dal primo (effetto a cascata).
Fragilità	Tendenza del software di "rompersi" in seguito a dei cambiamenti nel software. Spesso questi errori si verificano in aree che non hanno alcun collegamento con l'area modificata.
Immobilità	Non possibilità di riutilizzare il software da altri progetti oppure parti dello stesso progetto.
Viscosità	Si distingue in viscosità del design e viscosità dell'ambiente (environment). Il primo caso avviene se risulta facile commettere errori nel modificare il design, a sua volta difficile nel fare le cose in modo corretto. Il secondo caso avviene quando l'ambiente di sviluppo risulta lento e inefficiente.

Il fattore principale del perchè avvengono queste problematiche è dato dal fatto che al momento della creazione del design iniziale non vengono anticipati eventuali problemi. Il costante cambiamento delle parti del software porta a pensare che il design risulti inefficiente, quindi bisogna trovare un modo per renderlo più aperto al cambiamento.

3.2.1 Principi del Design Orientato agli Oggetti

3.2.2 Open Closed Principle - OCP

Un modulo dovrebbe essere aperto alle estensioni ma chiuso per le modifiche

L'obiettivo di questo principio è garantire la creazione di moduli estendibili ma non senza modificare il codice sorgente.

Si basa sulla tecnica dell'astrazione (in Java è abstract).

3.2.3 Liskov Substitution Principle - LSP

Le sottoclassi dovrebbero essere sostituibili per le loro classi base

L'obiettivo di questo principio è far sì che le classi derivate siano sostituibili per le loro classi padre. In questo modo, l'utente può continuare a usare le funzioni nonostante sia la classe derivata a essere passata (e non quella base).

Sfortunatamente, qualora si violasse questo principio, risulterebbe difficile rilevare gli errori.

3.2.4 Dependency Inversion Principle - DIP

Si basa sull'astrazione, non sulla concrezione

Questo principio si basa sull'uso delle interfacce o funzioni e classi astratte. Un'architettura orientata agli oggetti mostra una struttura dipendente dove la maggior parte delle dependency (relazioni) puntano verso le astrazioni. Nessuna dependency dovrebbe fare riferimento alle classi concrete. Il motivo è che gli oggetti concreti cambiano continuamente, quelli astratti meno frequentemente.

3.2.5 Interface Segregation Principle - ISP

Molte interfacce specifiche per gli utenti sono meglio di un'interfaccia generale

Se in una classe ci sono tanti clients, invece di caricare una classe con tutti i metodi richiesti dai clients, si creano delle interfacce specifiche per ciascuno. Nel caso più client necessitano dello stesso metodo, questo deve essere aggiunto in entrambi le interfacce.

3.2.6 Package Cohesion Principles

Sono utili per stabilire a quali package appartengono le classi.

3.2.7 Release Reuse Equivalency Principle - REP

La granulazione del riuso è la granulazione del rilascio

Spesso i client si rifiutano di utilizzare elementi a meno che gli autori non tengano traccia della versioni e mantengano le vecchie versioni per un po' di tempo. Quindi, un criterio per raggruppare le classi in un package è il riuso. Dal momento che i package sono delle unità di rilascio, a loro volta sono unità di riuso.

3.2.8 Common Closure Principle - CCP

Le classi che cambiano insieme sono inscindibili

Questo principio minimizza il numero di package che vengono cambiati per ogni rilascio del prodotto. Per far sì che questo principio funzioni, si raggruppano tutte quelle classi che pensiamo cambino insieme.

3.2.9 Common Reuse Principle - CRP

Classi che non vengono riusate insieme non dovrebbero essere raggruppate

Questo principio essenzialmente raggruppa quelle classi che non vengono usate insieme.

3.2.10 Package Coupling Principle

Questi principi governano le relazioni tra i package.

3.2.11 Acyclic Dependencies Principle - ADP

Le dipendenze tra i packages non devono creare dei cicli

Questo principio ha l'obiettivo di evitare la creazione di cicli, e quindi dipendenze cicliche, tra i package. L'eliminazione di un ciclo avviene in due modi, il primo consiste nel creare un nuovo package, il secondo è quello di usare i principi DIP e ISP.

3.2.12 Stable Dependencies Principle - SDP

Un modulo dovrebbe dipendere solo da moduli più stabili di lui

Quando si parla di *stabilità* si intende il quantitativo di lavoro richiesto per effettuare una modifica. Esistono tanti fattori che rendono un software difficile da modificare (complessità, dimensione, chiarezza ecc...), ma fattori più importante è il fatto che tanti package software dipendano da uno solo.

Esistono delle **Metriche di Stabilità** per semplificare le modifiche.

Ca - Afferent Coupling	Numero di classi fuori dal package che dipende dalle classi dentro al package.
Ce - Efferent Coupling	Numero di classi fuori dal package dipese dalle classi dentro al package.
I - Instability	$I = \frac{Ce}{Ca+Ce}$ Se $I = 0$, il package è stabile, altrimenti $I = 1$ ed è instabile.

3.2.13 Stable Abstraction Principle - SAP

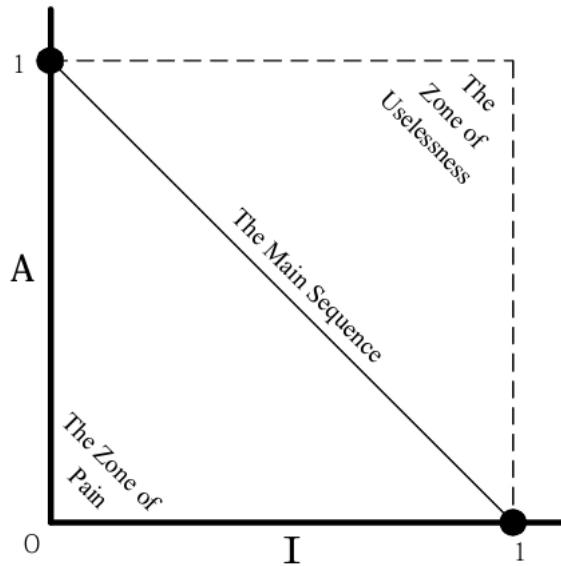
Package stabili dovrebbero essere astratti

Soltamente, una struttura di package prevede che quelli più in alto siano instabili e flessibili (facili da cambiare), mentre quelli più in basso sono difficili da cambiare per la loro stabilità.

Un modo per rendere più semplice la modifica avviene tramite **Metriche di Astrazione**:

Nc	Numero di classi nel package.
Na	Numero di classi astratte nel package
A	Astrazioni, $A = \frac{Na}{Nc}$ Se $A = 0$, il package non ha classi astratte, altrimenti $A = 1$ e contiene classi astratte.

Adesso possiamo confrontare le due metriche I e A . La metrica I dovrebbe incrementare mentre A decrementa, in questo modo, i package concreti dovrebbero essere instabili mentre quelli astratti dovrebbero essere stabili.



Nel grafico, in alto a sinistra troviamo i package astratti e stabili, in basso a destra i package concreti e instabili.

Altre metriche utili sono le **Metriche di Distanza**, utili per stabilire quanto è lontano un package dalla *main sequence*, ovvero la distanza massima tra i package concreti e quell astratti.

D - Distance	$D = \frac{ A+I-1 }{\sqrt{2}}$
D' - Normalized Distance	$D' = A + I - 1 $ Se $D' = 0$, il package è nella main sequence, se $D' = 1$, è il più lontano possibile.

3.3 Pattern of Enterprise Applications Architectures

All'interno di grandi aziende, si hanno una grandissima quantità di dati da gestire, organizzati in sistemi moderni come i DBMS, e nel tempo questi dati potranno dover essere migrati attraverso nuove applicazioni, si parla di dati persistenti su archi di tempo molto ampi.

L'accesso a questi dati è concorrente, arrivando fino a milioni di accessi contemporanei, le interfacce sono molto articolate, distinte tra loro e presentate in modo differente per tipi di utenti e obiettivi differenti.

Le applicazioni **enterprise** possono essere integrate con ulteriori applicazioni. Queste fanno uso di architetture stratificate (per la maggior parte di essi).

Business Logic: regole di interpretazione e elaborazione dei dati in base all'utilità dell'applicazione. Queste regole tendono a cambiare in base a diversi criteri:

- Gestione personalizzata di clienti
- Adeguamento a nuove logiche di mercato
- Cambiamento nelle strategie di marketing
- Altro

Riguardo la **performance**, molte decisioni architetturali aggravano su essa, portando a sacrificarne un po' pur di migliorare la comprensibilità.

Tra i fattori che costituiscono la performance abbiamo la **scalabilità**, **l'efficienza**, il **Tempo di Risposta** e la **Latenza** (Quanto puoi fare in un certo lasso di tempo).

3.3.1 Design Patterns e Architetturali

Per fare una distinzione tra i due:

I **Design Pattern** aiutano a trovare delle soluzioni a problemi in modo tale che si possano usare queste soluzioni tantissime volte. Purtroppo adattarli ai casi reali richiede uno sforzo progettuale significativo.

I **Pattern Architetturali** offrono soluzioni a problemi architetturali, aiutando a documentare le decisioni prese, facilitando la comunicazione tra gli stakeholder mediante un vocabolario comune.

3.3.2 Tipologie di Pattern Architetturali

1. Domain Logic Patterns

2. Data Source Architectural Patterns

3. Object Relational Behavioral Patterns

4. Object Relational Structural Patterns

5. Web Presentation Patterns

6. Concurrency Patterns

7. Security Patterns

 7.1 Federated Identity Pattern

 7.2 Gatekeeper Pattern

 7.3 Valet Key Pattern

8. Performance Patterns

Per descrivere un pattern architetturale ci focalizzeremo su 4 punti:

1. **Descrizione**

2. **Diagramma UML**

3. **Esempio di applicazione**

4. **Vantaggi e Svantaggi**

3.3.3 Frameworks

I framework sono delle librerie di codice astratte e estendibili che vengono adattate per scopi specifici da parte degli sviluppatori. Per i software moderni sono oramai essenziali, migliorano la produttività ma con degli effetti collaterali indesiderati, come debolezze architetturali o vulnerabilità nella sicurezza.

Tra i framework più conosciuti abbiamo

- Maven
- Django
- Spring
- Molti altri

Tra i produttori di framework troviamo anche **Amazon Web Services (AWS)** e **Microsoft's Azure**.

3.4 Web Presentation Patterns

Generalmente, sono quei pattern volti alla definizione dell’interfaccia utente. Tra le tecnologie e i framework che fa uso questo tipo di pattern abbiamo:

- Java Servlet
- Server pages (JSP, PHP, ASP)
- Apache Xalan
- Altro

I Web Presentation Patterns si dividono in due categorie:

1. Model-view-controller

- (a) Controller Design
 - i. Page Controller
 - ii. Front Controller
 - iii. Intercepting Filter
- (b) View Design
 - i. Template view
 - ii. Transformation view
 - iii. Two-step view

2. Application Controller

4 Metriche, Understand, Refactoring e Antipattern strutturali

4.1 Metriche Object Oriented

Nei sistemi Object Oriented, per gestire casi come Polimorfismo, Ereditarietà, Information hiding e Astrazione, vengono usate delle metriche apposite in base al criterio che vogliamo verificare.

Le metriche sono:

1. Line of code per class
2. Weighted method per class
3. Depth of the inheritance tree
4. Number of children
5. Coupling between object classes
6. Response for a class
7. Lack of cohesion in methods

4.1.1 Line of code per class (LOCC)

Calcola quante linee di codice ci sono in un metodo.

Sia:

1. c la classe che si sta valutando
2. $LOC(m)$ il numero di linee nel metodo m
3. $M_{Im}(c)$ insieme dei metodi implementati nella classe c

La formula per calcolare le linee di codice nella classe è

$$LOCC(c) = \sum_{m \in M_{Im}(c)} LOC(m) \quad (1)$$

4.1.2 Weighted method per class (WMC)

Calcola il livello di complessità dei metodi in una classe. Sia:

1. c la classe che si sta valutando
2. $VG(m)$ la complessità ciclomatica del metodo m
3. $M_{Im}(c)$ insieme dei metodi implementati nella classe c

La formula per calcolare la complessità di una classe (o complessità di McCabe) è

$$WMC(c) = \sum_{m \in M_{Im}(c)} VG(m) \quad (2)$$

È una metrica importante, in quanto stabilisce l'indice di complessità di una classe, quindi quanto sia comprensibile e facile da modificare.

Maggiore è il numero dei metodi, maggiore è la complessità della classe

\Rightarrow maggiore è la probabilità che la classe è legata all'applicazione

\Rightarrow minore possibilità di riuso.

4.1.3 Depth of the inheritance tree (DIT)

Calcola la distanza massima di un nodo dalla radice dell'albero. In altre parole verifica quanto una classe venga ereditata da altre classi.

- Maggiore è la profondità della classe, maggiore è il numero di metodi che essa può ereditare, rendendo più complesso predire il suo comportamento.
- Alberi di ereditarietà con maggiore profondità aumentano la complessità del progetto, dato che più classi e metodi sono coinvolti.
- Maggiore è la profondità di una classe in una gerarchia, maggiore è il riuso potenziale dei metodi ereditati.

4.1.4 Number of children (NOC)

Calcola il numero di sottoclassi di una superclasse.

Al crescere del numero di sottoclassi si verificano i seguenti casi:

- Cresce il livello di riuso.
- Aumenta la quantità di test case necessari a testare ogni superclasse.
- La superclasse tende a diventare impertinente, perché ciascuna sottoclasse può aggiungere dei comportamenti specifici ai metodi, distaccandosi dall'idea generale della superclasse.
- Le sottoclassi potrebbero non essere membri appropriati della madre, perché può accadere che la sottoclasse erediti dei metodi che non ritiene sensato usare.

4.1.5 Coupling between object classes (CBO)

Calcola il numero di classi a cui una classe è accoppiata.

Un eccessivo accoppiamento è sicuramente negativo in termini di modularità e riuso, infatti si consiglia sempre di mantenere un basso accoppiamento (low coupling) in modo da facilitare test e modifiche.

4.1.6 Response for a class (RFC)

Calcola l'insieme dei metodi che possono essere eseguiti in risposta ad un messaggio ricevuto da un oggetto della classe.

Un elevato numero di metodi può aumentare la complessità progettuale della classe e lo sforzo dei test.

4.1.7 Lack of cohesion in methods (LCOM)

Calcola la non-coesione tra gli elementi di una classe. Viene espressa tramite numero di metodi che accedono agli stessi attributi di una classe.

Sia:

1. C una classe con n metodi M_1, M_2, \dots, M_n
2. I_j l'insieme delle istanze di variabili referenziate da M_i
3. $P = \{(I_j, I_i) \mid I_i \cap I_j = \emptyset\}$
4. $Q = \{(I_j, I_i) \mid I_i \cap I_j \neq \emptyset\}$

Presi tutti gli insiemi I_1, \dots, I_n , se sono vuoti allora $P = \emptyset$.

se $|P| > |Q|$ allora $LCOM = |P| - |Q|$, altrimenti $LCOM = 0$

Quindi $LCOM = \text{num. intersezioni vuote} - \text{num. intersezioni non vuote}$

Minore è il valore di LCOM, maggiore è la coesione (metodi strettamente legati tra loro e la classe ha uno scopo chiaro).

Maggiore è il valore di LCOM, minore è la coesione (i metodi non condividono molti attributi comuni, quindi la classe è più difficile da gestire e comprendere).

- È sempre consigliato avere una classe altamente coesa, in modo da favorirne la sua encapsulazione.
- Una bassa coesione può comportare una divisione di una classe in due o più sottoclassi.
- Una bassa coesione comporta anche complessità e probabilità di compiere errori durante il processo di sviluppo.

Ultima considerazione riguardo a questa metrica, esiste un altro metodo proposto da Henderson-Sellers che, tramite una formula (che non starò a scrivere), ci indica se la coesione è perfetta o meno:

- Se **LCOM = 0**, la coesione è perfetta (tutti gli attributi sono acceduti da tutti i metodi della classe).
- Se **LCOM = 1**, c'è una completa mancanza di coesione (ogni attributo è acceduto da un solo metodo della classe).

5 Esercitazioni

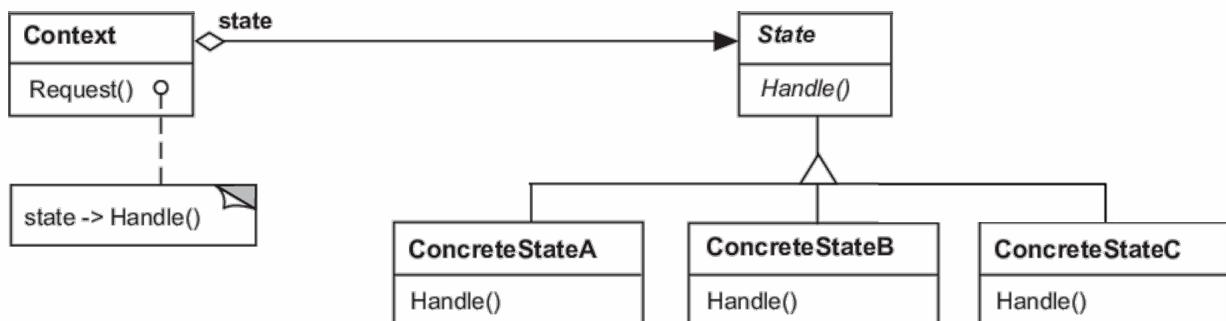
In questo capitolo, vengono analizzati dei design pattern nuovi e non trattati durante il corso di Analisi e Progettazione. Verrà fatto anche qualche riferimento a pattern già visti.

Attenzione, il file fornito chiarisce solo le nozioni teoriche sui pattern trattati, per approfondirli in maniera pratica si consigliano le slide fornite dai professori (e ovviamente testare su codice il loro funzionamento).

5.1 Design Pattern State

Permette a un oggetto di modificare il suo comportamento quando il suo stato interno cambia.

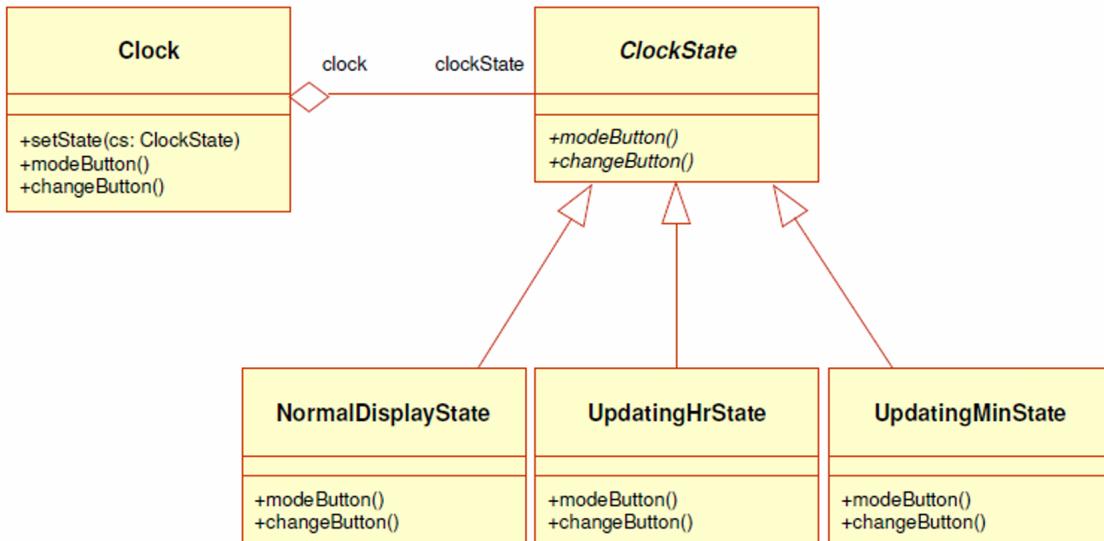
State viene utilizzato quando si vuole modificare lo stato interno di un oggetto in modo da potergli far eseguire determinati metodi. Gli stati degli oggetti vengono rappresentati da classi differenti che estendono una classe astratta che rappresenta lo stato dell'oggetto.



1. **Context** è l'interfaccia del client. Ha un'istanza di **ConcreteState** che indica in che stato si trova.
2. **State** è la classe astratta che rappresenta lo stato attuale di **Context**.
3. **ConcreteState** sono le classi che implementano il comportamento associato allo stato del **Context**.

Vantaggi	Svantaggi
Eliminazione di tanti "if"	Incremento del numero di oggetti
Facile lettura del codice	Richiede la scrittura di molto codice
Nuovi stati e transizioni possono essere aggiunti facilmente definendo nuove sottoclassi	

5.1.1 Esempio di Pattern State



In questo esempio, **Clock** contiene 3 metodi di cui **modeButton()** e **changeButton()** cambiano il loro comportamento in base allo stato in cui si trova. **setState(cs: ClockState)** permette di cambiare lo stato di **Clock**, in questo modo, anche i due metodi all'interno cambieranno il loro comportamento.

5.1.2 State vs Strategy

Sono entrambi pattern comportamentali, ma hanno obiettivi e applicazioni differenti.

State cambia il comportamento di un oggetto in base al suo stato interno.
Es. Un lettore audio che può avere stati come "Play", "Pause" e "Stop", dove ogni stato gestisce le azioni dell'utente in modo diverso.

Strategy consente di cambiare il modo in cui un'operazione viene eseguita, senza modificare lo stato interno dell'oggetto.

Es. Un sistema di pagamento che può usare diverse strategie come "Carta di credito", "PayPal" o "Bonifico", a seconda della scelta dell'utente.

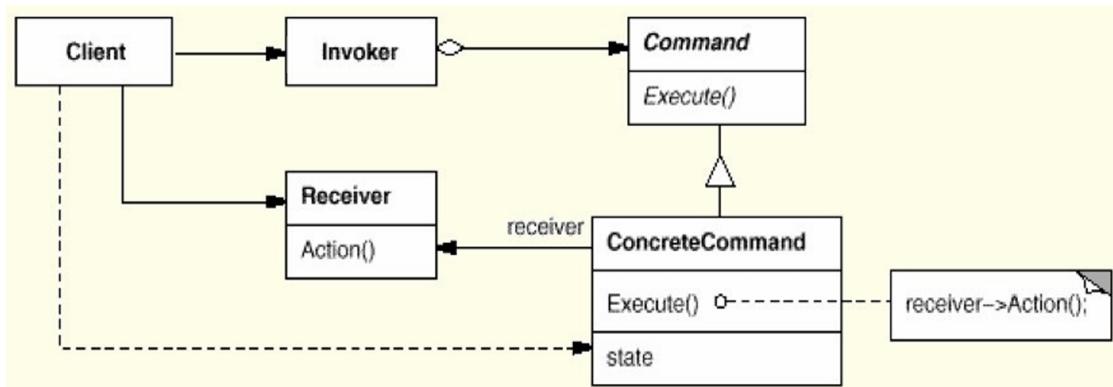
5.2 Design Pattern Command

Permette di trasformare una richiesta in un oggetto indipendente.

In questo modo si possono:

1. scollegare il mittente di una richiesta dal destinatario.
2. rendere le operazioni riutilizzabili.
3. registrare e annullare operazioni.
4. creare sequenze di comandi.

Command è ideale per implementare operazioni complesse e modulari che possono essere facilmente eseguite, annullate, memorizzate e organizzate in sequenza.



1. **Client** è l'interfaccia che crea un'istanza concreta **ConcreteCommand** e ne imposta il **Receiver**.
2. **Command** è l'interfaccia per l'esecuzione dell'operazione.
3. **ConcreteCommand** è la classe che incapsula l'operazione da eseguire in un oggetto specifico.
4. **Invoker** è la classe che richiede tramite il **Command** l'esecuzione di una o più operazioni.
5. **Receiver** è l'oggetto che esegue l'azione.

Vantaggi

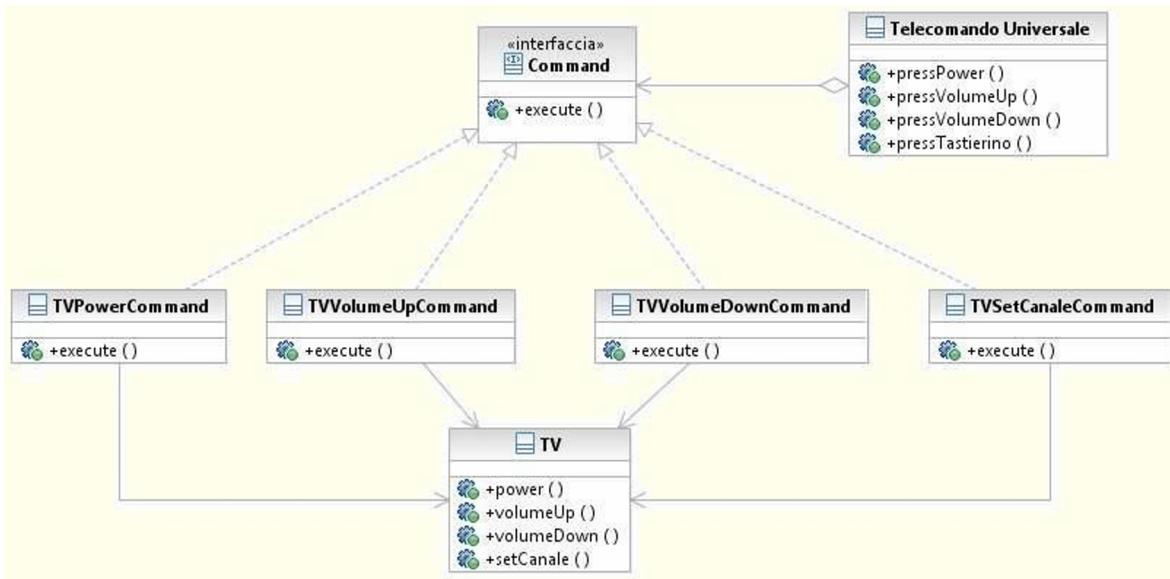
Separa l'oggetto che invoca l'operazione da quello che sa come eseguirlo

I *Command* sono classi di oggetti e possono essere manipolati come qualsiasi altro oggetto.

Si possono creare dei *Command* composti.

È semplice aggiungere nuovi *Command*.

5.2.1 Esempio di Design Pattern Command



In questo esempio, **TV** è il **Receiver** che esegue l'azione stabilita dal **Telecomando Universale**, ovvero l'**Invoker**.

I **ConcreteCommand** sono le 4 classi che ereditano il metodo **execute()** che invocano le corrispondenti operazioni del **Receiver**.

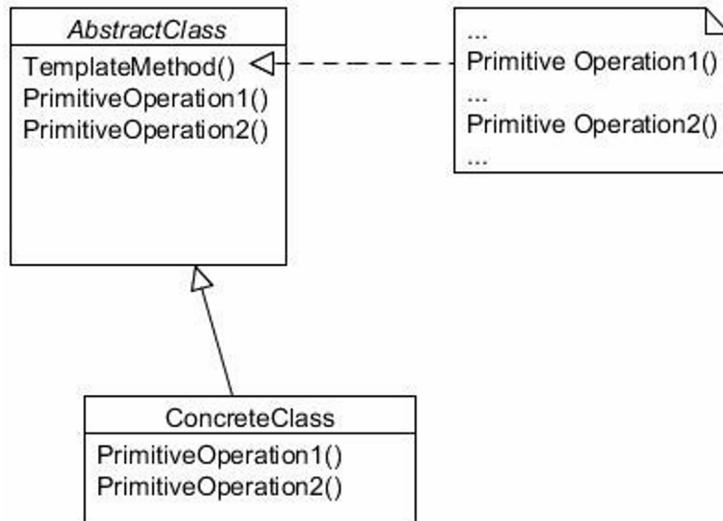
Infine l'interfaccia **Command** che definisce l'interfaccia alle classi concrete.

5.3 Design Pattern Template

Definisce lo scheletro in un'operazione, lasciando alle sottoclassi di modificare l'algoritmo senza cambiare la sua struttura.

Usiamo *Template* quando si vuole implementare le invarianti di un algoritmo una sola volta, saranno le sottoclassi a implementare le varie varianti.

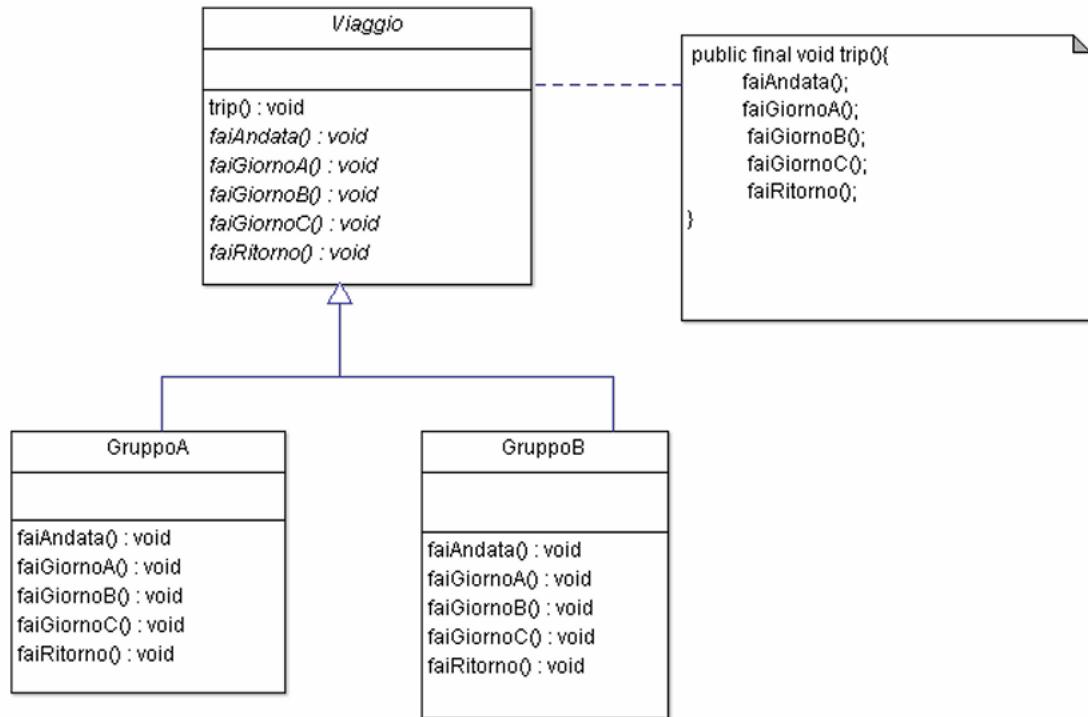
5.3.1 Struttura del pattern Template



1. **AbstractClass** è la classe che definisce le operazioni primitive di tipo astratto. Queste vengono poi implementate dalle sottoclassi **ConcreteClass**. Definisce un metodo **TemplateMethod()** che rappresenta lo scheletro di un algoritmo e richiama le operazioni primitive.
2. **ConcreteClass** sono le classi che implementano le operazioni primitive per poterne variare il comportamento.

I metodi template sono fondamentali per il riutilizzo del codice, soprattutto nelle librerie di classi, questo perchè sono i mezzi per estrapolare i comportamenti comuni in classi di libreria. Bisogna sempre specificare quali operazione si vogliono sovrascrivere e quali metodi trasferiti alle sottoclassi.

5.3.2 Esempio di Design Pattern Template

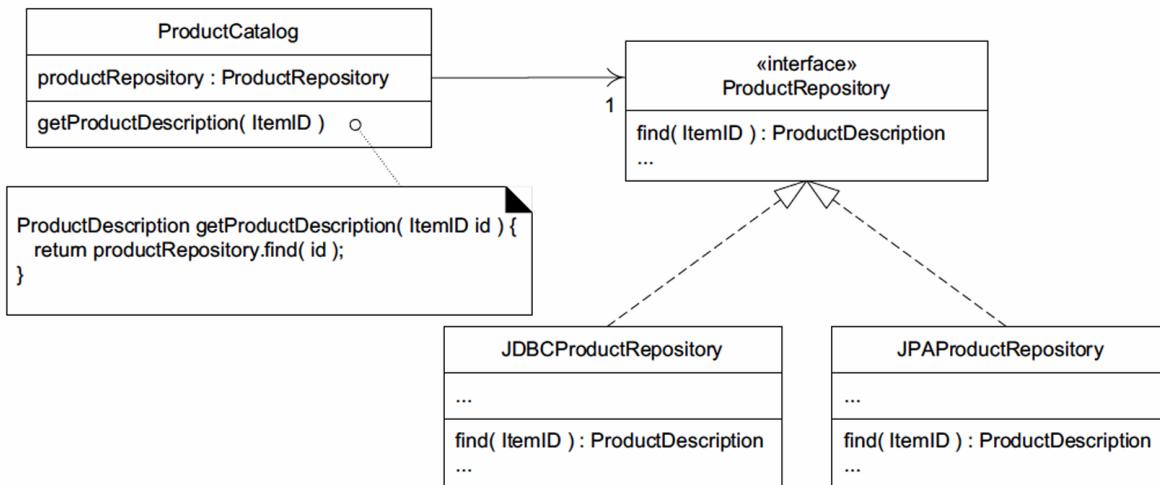


In questo esempio, **Viaggio** è l'**AbstractClass** che contiene il metodo Template **trip()**. Le **ConcreteClass** **GruppoA** e **GruppoB** ereditano i metodi dalla superclasse, decidendo però come deve essere implementato ciascun metodo.

5.4 Design Pattern Repository

Questo design pattern viene usato se, ad esempio, si vuole accedere a una base di dati persistenti.

Il pattern fornisce un'illusione di una collezione in memoria per ogni tipo di oggetto persistente che richiede l'accesso globale. L'accesso avviene tramite un'interfaccia che definisce delle operazioni per aggiungere, rimuovere o ricercare oggetti. Queste operazioni encapsulano l'accesso effettivo alla rappresentazione dei dati.



In questo esempio abbiamo 3 componenti:

1. **ProductRepository** è l'interfaccia che funge da repository dei prodotti, viene usata per accedere alle informazioni di ciascun prodotto. tramite il metodo **find()**.
2. **JDBCProductRepository** e **JPAProductRepository** sono delle repository contenenti le informazioni.
3. **ProductCatalog** è una classe che tramite l'*ItemID* può ottenere la descrizione della repository corrispondente.

5.5 Failover

Prima di parlare di failover, bisogna definire 3 termini apparentemente uguali:

Fault (Guasto/Difetto): L'origine o la causa di un comportamento sbagliato

Error (Errore): La manifestazione di un difetto nel sistema

Failure (Fallimento): La mancata erogazione di un servizio causata da un errore

Per **Failover** si intende un meccanismo usato per garantire la continuità di un servizio in caso di un eventuale guasto o interruzione del sistema. Questo meccanismo agisce in maniera trasparente all'utente, simulando le operazioni del sistema principale.

L'obiettivo di un sistema è quello di garantire affidabilità, permettendo il ripristino in seguito a guasti esterni. La soluzione consiste nel:

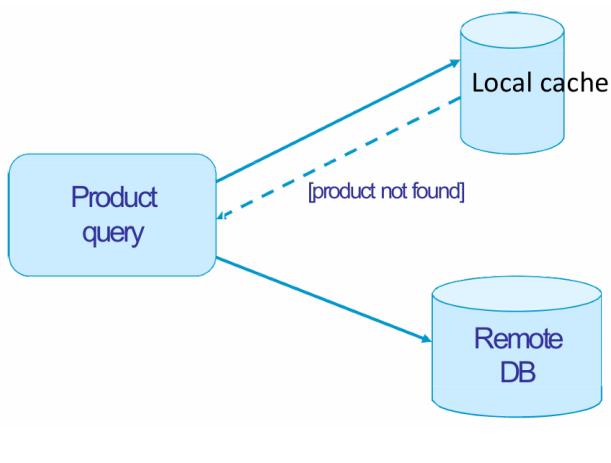
1. **Rendere trasparente la locazione fisica dei dati**
2. **Rendere il failover da remoto a locale**
3. **Replicare parzialmente i servizi locali**

Nel dettaglio, la soluzione consiste nell'usare un **Proxy** (fornito da una *ServicesFactory* di tipo Factory) in grado di gestire in maniera trasparente e sicura al client eventuali guasti, mascherando la locazione e la disponibilità dei servizi. Nel caso in cui un servizio dovesse essere non disponibile, il proxy effettua un failover verso un servizio secondario, senza che il client debba fare nulla.

Altro obiettivo del Proxy è far sì che realizzi delle implementazioni locali dei servizi remoti che hanno problemi, in modo che abbiano un comportamento semplificato e vincolato.

Nel pratico, decidiamo di creare una cache locale di oggetti di tipo *ProductDescription*. In questo modo, avremo un aumento delle performance e dell'affidabilità.

5.5.1 Accesso alla cache locale



L'accesso alla cache locale avviene tramite il *ServicesFactory*, questo ritorna un adapter a un servizio di informazioni relativo ai prodotti in locale. L'adapter sarebbe l'entità che andrà a implementare le responsabilità del servizio in locale, questo servizio sarà inizializzato con un riferimento verso un secondo adapter, ma quest'ultimo è relativo al corrispondente servizio ma remoto.

In sintesi, nel primo adapter abbiamo le responsabilità del servizio **locale**, il secondo adapter è relativo al corrispondente servizio locale ma in **remoto**. Se il servizio locale trova il prodotto nella cache (locale), allora lo ritorna, altrimenti inoltra la richiesta all'adapter del servizio remoto.

Esistono due tipologie di cache:

1. Cache che mantengono in memoria una collezione di grandezza limitata ma dinamica (es. il *ProductCatalog* che ha in memoria oggetti di tipo *ProductDescription*)
2. Cache che mantengono in memoria una collezione più ampia di elementi su supporti persistenti (es. hard disk), questo implica che, se il sistema dovesse malfunzionare, la cache rimane persistente.

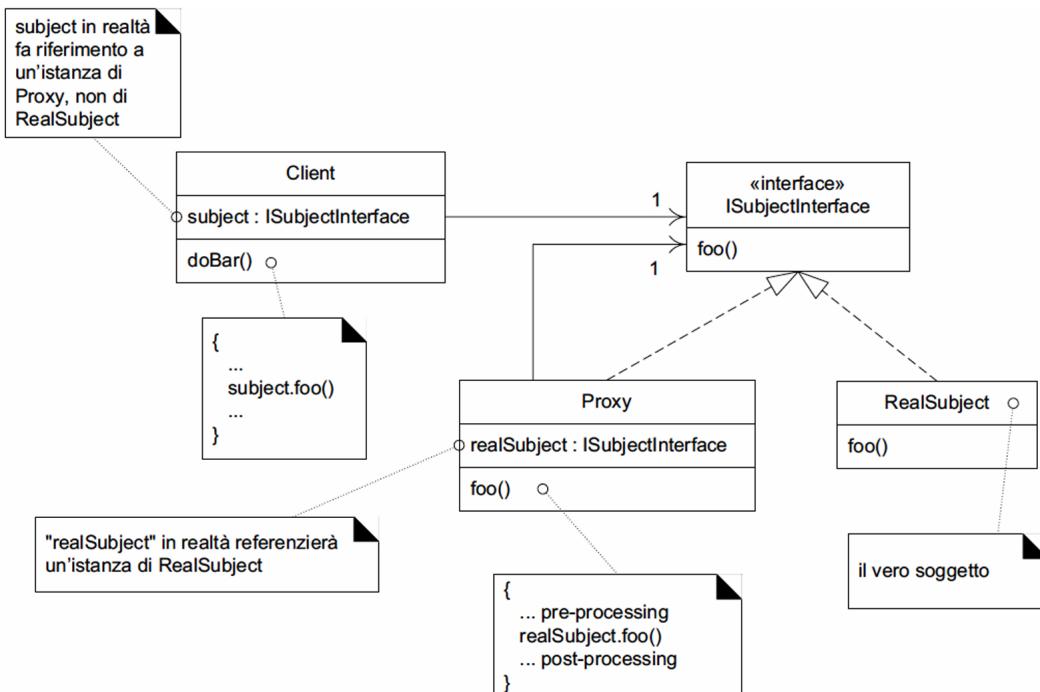
Cosa succede se l'accesso alla cache locale non dà risultato e l'accesso al servizio esterno fallisce? Si verificherà un **fallimento**, quindi bisogna gestirlo.

Una soluzione comune sarebbe quella di **lanciare un'eccezione**, utile se l'errore si verifica a livello hardware. L'eccezione non deve essere percepita a livello di presentazione (l'utente non deve vedere l'eccezione).

Il pattern a cui si fa ricorso è il Design Pattern **Convert Exceptions** (descritto dopo il pattern Proxy).

5.6 Design Pattern Proxy

Il pattern permette l'accesso a quegli oggetti che non sono direttamente accessibili o non si desidera fornire l'accesso diretto. Viene creato un oggetto proxy che implementa la stessa interfaccia dell'oggetto ed è responsabile del controllo o del miglioramento dell'accesso a questo oggetto.



I componenti del pattern sono:

1. **SubjectInterface** definisce un'interfaccia comune per l'oggetto reale **RealSubject** e il **Proxy**, permettendo di usarli in modo alternabile.
2. **RealSubject** è l'oggetto reale, contiene l'implementazione concreta della logica principale.
3. **Proxy** è la classe che implementa la **SubjectInterface** e controlla l'accesso all'oggetto reale, implementando logica extra, contiene un riferimento al **RealSubject**.

Esistono due strategie di gestione della cache:

1. **Inizializzazione lazy**: la cache viene riempita lentamente man mano che vengono inseriti gli oggetti.
2. **Inizializzazione greedy**: la cache viene caricata all'inizio.

5.6.1 Ultime considerazioni sul Proxy

1. Il Proxy è un oggetto esterno che nasconde un oggetto interno.
2. Il client non sa che sta facendo riferimento al Proxy, ma crede di comunicare direttamente con il sistema interno.
3. Il Proxy è utile nel caso sia necessaria una ridirezione dei messaggi.

5.7 Design Pattern Convert Exceptions

Il pattern permette di convertire un'eccezione di livello basso in un'eccezione più significativa a livello del sottosistema dove viene lanciata l'eccezione. L'eccezione di livello più alto avvolge quella di livello più basso e aggiunge informazioni per renderla più significativa nel contesto del livello superiore.

Un consiglio è quello di rinominare l'eccezione in base al **perchè è stata lanciata, non come**, in modo da rendere più semplificare al programmatore capire il problema.

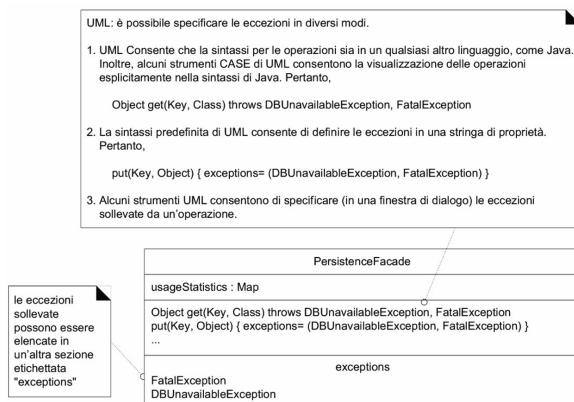


Figura 1: Eccezioni in un diagramma delle classi

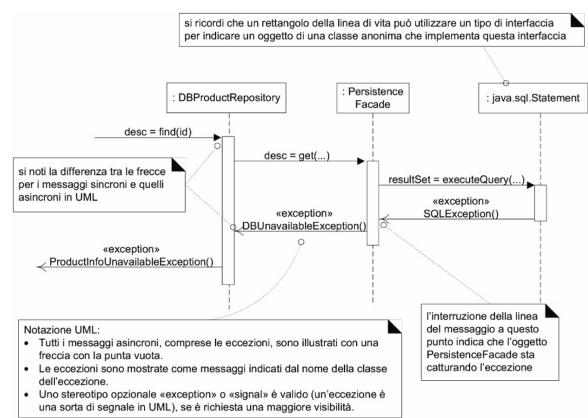


Figura 2: Eccezioni in un diagramma di sequenza

Per gestire l'eccezione, può essere gestita tramite due pattern, il **Centralized Error Logging** e l'**Error Dialog**.

5.8 Design Pattern Centralized Error Logging

Il pattern funge da oggetto centrale per il logging degli errori a cui si accede come un Singleton, e si riportano in esso tutte le eccezioni. In caso di sistema distribuito, ciascun Singleton locale collaborerà con un logger degli errori centrale.

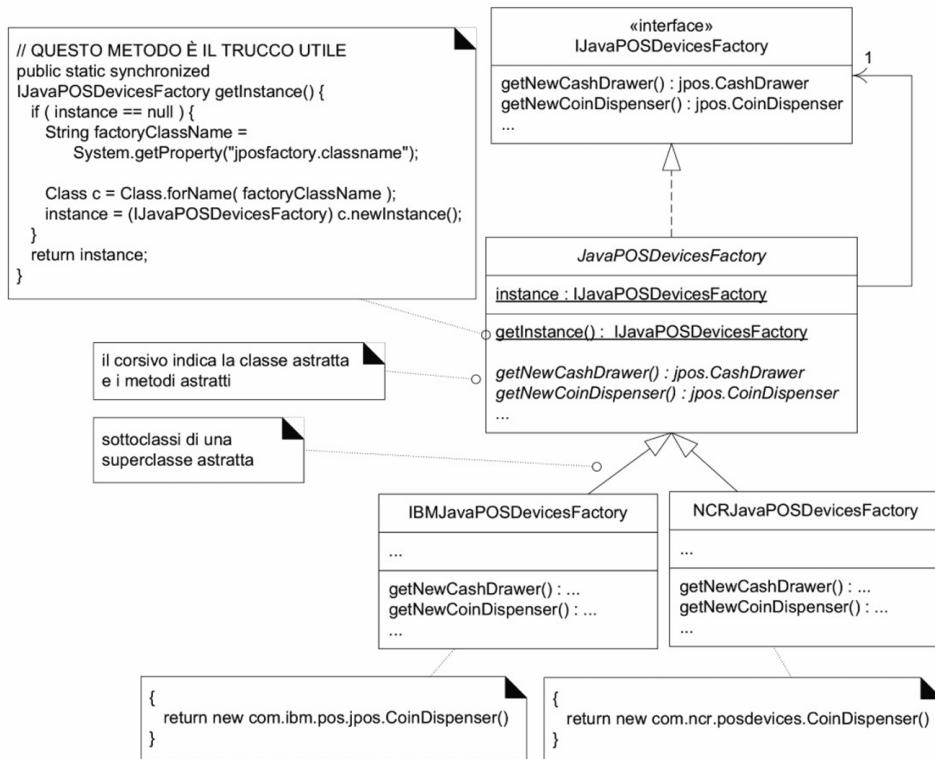
5.9 Design Pattern Error Dialog

Il pattern permette di utilizzare un oggetto non appartenente all'interfaccia utente e accessibile attraverso un Singleton con lo scopo di comunicare gli errori agli utenti. Il pattern "avvolge" uno o più oggetti della UI (es. una finestra di testo), delegandogli la notifica dell'errore. Inoltre, ripoterà l'errore al logger centralizzato degli errori. Una *Factory* ha il compito di creare l'oggetto UI appropriato in base ai parametri di sistema ricevuti, quindi in base all'errore.

5.10 Design Pattern Abstract Factory

Il pattern permette di definire un'interfaccia astratta *Factory* e altre classi **Factory** per ogni famiglia di elementi da creare. In questo modo, si possono creare delle famiglie di classi correlate tra loro che implementano un'interfaccia comune.

Figura 1: Esempio di Abstract Factory



Spesso, l'*Abstract Factory* viene creata come **classe astratta**, non come interfaccia.

La Factory legge dalle proprietà di sistema quale famiglia di oggetti creare, questo grazie all'aiuto del pattern **Singleton** (con il metodo *getInstance()*).

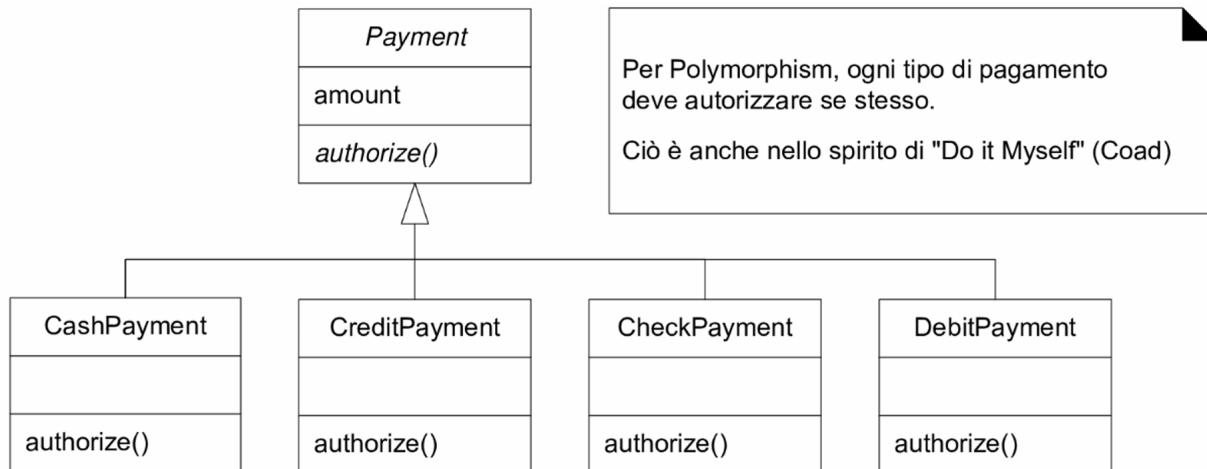
5.11 Design Pattern Do It Myself

Chiamato così da Peter Coad, il pattern ha l'obiettivo di applicare il **Polimorfismo**, quindi far sì che un oggetto software faccia quelle operazioni che normalmente verrebbero fatte all'oggetto reale di cui loro sono un'astrazione (loro intesi come gli oggetti software).

Detto in modo informale:

- Gli oggetti Quadrato creano se stessi.
- Gli oggetti Cerchio creano se stessi.
- Gli oggetti Text eseguono il controllo ortografico di se stessi.

Figura 1: Uso del Polimorfismo con i mezzi di pagamento



5.12 Progettazione di Framework di persistenza dei dati

In questa esercitazione, si useranno 4 design pattern già visti per la progettazione di framework di persistenza dei dati:

1. Template Method
2. State
3. Command
4. Proxy

Quando parliamo di **Oggetti persistenti**, si intendono quegli oggetti che rimangono in memoria durante l'esecuzione del programma, quindi senza che vengano ripristinati alla fine dell'esecuzione.

Solitamente, si fa ricorso ai *database relazionali* che permettono di modellare un mapping tra la rappresentazione *record oriented* del database e quella *object-oriented* del sistema.

Ecco alcune definizioni riguardo la persistenza:

Per **Framework di Persistenza** si intende un insieme di classi e interfacce riusabili, estendibili e *general purpose*, che forniscono le funzionalità per la gestione degli oggetti persistenti.

Per **Servizio di Persistenza** si intende un sottosistema che fornisce le funzionalità per cooperare con il database e generalmente creato dal framework di persistenza.

Per **Oggetti Persistenti** si intendono quegli oggetti che necessitano di memorizzazione persistente.

5.12.1 Proprietà dei framework

1. È un insieme coeso di interfacce e classi che collaborano tra loro per fornire servizi per la parte fondamentale e invariabile di un sottosistema logico.
2. Contiene classi concrete e astratte che definiscono le interfacce a cui conformarsi e le interazioni tra gli oggetti.
3. Di solito, richiede all'utente di definire delle sottoclassi di classi esistenti del framework per personalizzare ed estendere i servizi del framework.
4. Ha delle classi astratte che possono contenere metodi astratti e concreti.

5.12.2 Servizio di persistenza

Un compito fondamentale del **Servizio di Persistenza** è quello di **effettuare il mapping fra la rappresentazione a oggetti e rappresentazione relazionale dei dati**, per farlo, compie due operazioni:

1. **Materializzazione**: traduzione di record in oggetti (caricamento in memoria).
2. **Dematerializzazione**: traduzione di oggetti in record (memorizzazione nel DB).

Per creare un servizio di persistenza, si deve realizzare un framework di persistenza che abbia le seguenti funzionalità:

1. Memorizzazione e recupero degli oggetti in e da un sistema di storage persistente.
2. Fornire transazioni coerenti di **commit** e **rollback**.
3. Deve essere estendibile per poter supportare diversi meccanismi e formati di memorizzazione (es. XML).

I **punti chiave** per progettare un framework di persistenza sono:

1. Mapping.
2. Indetificazione univoca degli oggetti.
3. Materializzazione e Dematerializzazione degli oggetti (mediante pattern *Template*).
4. Gestione dello stato degli oggetti persistenti (mediante pattern *State*).
5. Modellazione delle operazioni relative a una transazione quali commit e rollback (mediante pattern *Command*).
6. Lazy materialization (mediante pattern *Proxy*).
7. Aumento delle performance con l'uso della cache.

5.12.3 Mapping

Il mapping risponde alla domanda *"Come facciamo a far corrispondere un oggetto a un record in un DB relazionale?"*

5.12.4 Design Pattern Representing Objects as Tables

Un pattern utile allo scopo è il **Representing Objects as Tables**.

Il pattern permette di definire una tabella per ciascuna classe di oggetti persistenti dentro un DB relazionale. Gli attributi degli oggetti corrisponderanno alle colonne della tabella e conterranno oggetti di tipo primitivo.

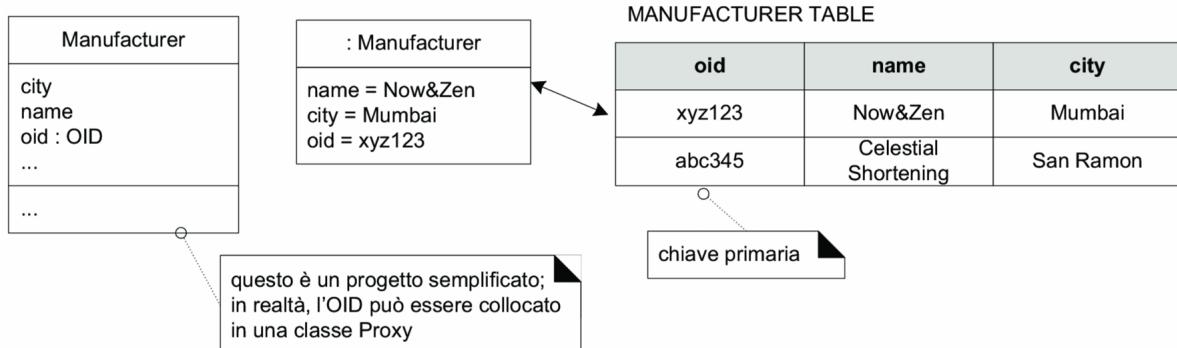
5.12.5 Identificazione univoca di oggetti

Questo punto chiave permette di non ripetere la materializzazione di oggetti molteplici volte, col rischio di creare oggetti duplicati.

5.12.6 Design Pattern Object Identifier

Il pattern utile allo scopo è l'**Object Identifier**.

Il pattern permette di assegnare a ogni record di tabella e a ogni oggetto un valore alfanumerico chiamato Object ID che consente l'immediata identificazione di ogni istanza.

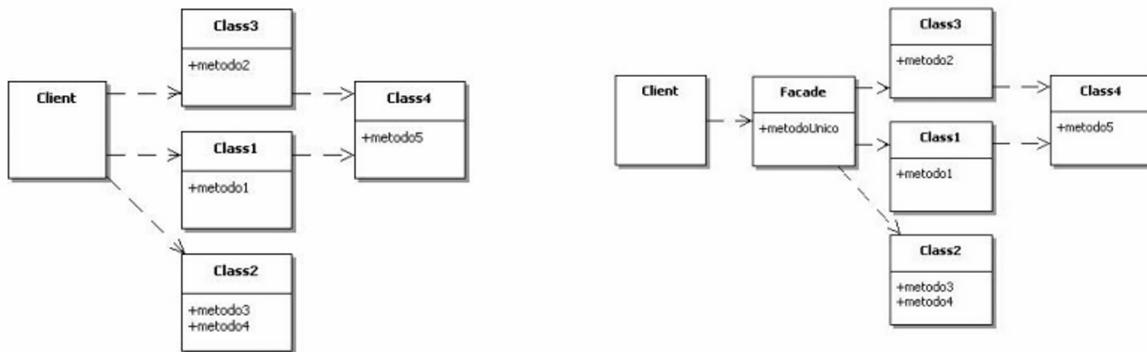


5.12.7 Accesso a un Servizio di Persistenza

Un pattern utile a fornire un unico punto di accesso è il **Facade**.

5.12.8 Design Pattern Facade

Il pattern fornisce un'interfaccia unificata ai servizi forniti di un certo sottosistema. La Facade delega le richieste provenienti dai client verso gli oggetti del sottosistema che nasconde. Il suo intento è quello di rendere più semplice l'uso di un sottosistema.



I partecipanti del pattern sono:

1. **Facade**: Conosce le classi responsabili per una richiesta e delega le richieste dei client agli oggetti appropriati del sottosistema.
2. **Classi del sottosistema**: Implementano le funzionalità del sottosistema e gestiscono il lavoro assegnato dal Facade, questo però senza aver alcun riferimento di quest'ultima.

Bisogna fare alcune considerazioni riguardo questo pattern:

1. Diminuisce l'accoppiamento tra client e sottosistema.
2. Nasconde al client le componenti del sottosistema.
3. Il client può comunque usare direttamente le classi del sottosistema.

Si può anche rendere il Facade una classe astratta con sottoclassi concrete per diminuire ulteriormente l'accoppiamento. I client possono comunicare con il sottosistema mediante l'interfaccia della classe astratta Facade.

5.12.9 Facade vs Adapter

1. Entrambi sono dei wrapper.
2. Entrambi si basano su un'interfaccia, però:
 - (a) Facade lo semplifica.
 - (b) Adapter lo converte.

5.12.10 Materializzazione e Dematerializzazione

In questa sezione vedremo chi si occupa della Materializzazione e l'archiviazione degli oggetti.

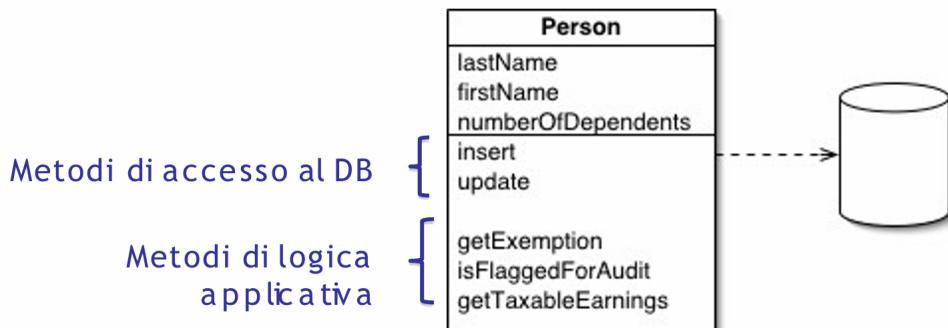
Esistono due tipi di **Mapping**:

1. **Direct Mapping**: una classe che rappresenta un oggetto persistente definisce al suo interno il codice per il suo salvataggio nel DB (il codice è generato automatico).
2. **Indirect Mapping**: esistono classi apposite volte al recupero e al salvataggio dei dati in un DB.

Questi due tipi di mappature si possono rappresentare mediante due pattern: **Active Record** per il Direct e **Database Mapper** per l'Indirect.

5.13 Active Record - Direct Mapping

Il pattern architetturale permette di convertire ciascuna classe di dominio in tabella del DB. Combina i dati e il comportamento di un'entità in un'unica classe e inserisce la logica di accesso ai dati nell'oggetto di dominio, in modo che tutti sappiano come leggere e scrivere i dati da/al DB.



Active Record fa uso di alcuni metodi e strutture dati:

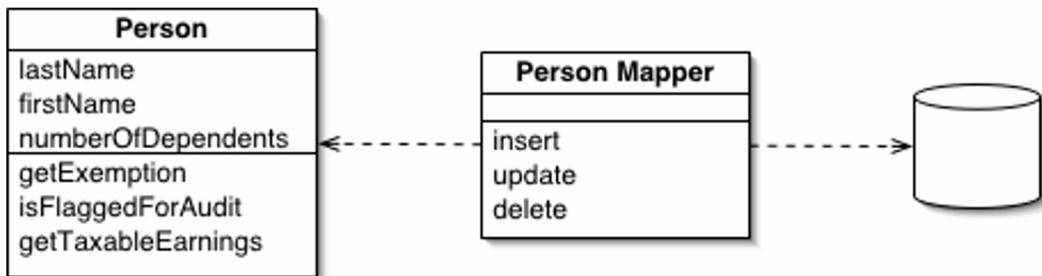
1. **Metodo load:** costruisce un'istanza partendo dai risultati generati da una query SQL.
2. **Metodi finder statici:** encapsulano le query SQL e ritornano le istanze degli oggetti.
3. **Costruttore classico:** costruisce nuove istanze da inserire successivamente nel DB.
4. **Metodi di aggiornamento del DB:**
 - 4.1 **update:** aggiorna un record esistente con i valori degli attributi.
 - 4.2 **insert:** aggiunge un record utilizzando i valori degli attributi.
 - 4.3 **delete:** elimina il record corrispondente dell'oggetto corrente.
5. **Metodi accessors:**
 - 5.1 I metodi **set** e **get** per accede ai campi.
 - 5.2 Effettuano la conversione dei dati per memorizzare i valori degli attributi in un formato SQL-oriented.
 - 5.3 Possono richiedere un'immediata sincronizzazione del DB.
6. **Metodo di logica di business**

Vantaggi	Svantaggi
Semplice da implementare e usare	L'accoppiamento fra la logica applicativa e il DB rende difficile il refactoring dei due progetti Si cerca di mantenere una corrispondenza stretta (o quasi) fra lo schema del DB e l'entità del modello OO

5.14 Database Mapper - Indirect Mapping

Il framework (o pattern di progettazione) permette di definire una classe che si occupi della gestione dei processi di materializzazione da un DB, la dematerializzazione della memoria verso il DB e il caching degli oggetti con l'obiettivo di aumentare la performance del sistema.

Il pattern definisce un DB Mapper per ogni classe di oggetti persistenti, possono esistere diversi tipi di Mapper a seconda dei meccanismi di memorizzazione.



Person Mapper è uno strato separato di componenti dedicati a trasferire i dati fra l'applicazione (**Person**) e il **Database**, trattando indipendentemente i due schemi (OO e ER). Infatti, la logica di business è inconsapevole dell'esistenza del Database.

Generalmente, il pattern ne coinvolge di ulteriori per la gestione della sincronizzazione fra i due schemi (es. *UnitOfWork* o *IdentityMap*).

Usiamo questo pattern per **gestire un mapping complesso fra DB e logica di business**.

Vantaggi	Svantaggi
Isola totalmente i due strati	Difficoltà nell'implementazione

5.15 Materializzazione in cache

Vogliamo mantenere all'interno della cache locale gli oggetti che sono stati materializzati in modo da:

1. Aumentare le prestazioni
2. Supportare la gestione delle transazioni

Il pattern utile allo scopo è il **Cache Management**

5.15.1 Design Pattern Cache Management

Il pattern estende il design pattern **Database Mapper**, facendo sì che siano responsabili per la gestione della cache. Ogni Mapper può mantenere e gestire una propria cache privata. Il pattern verifica prima di tutto se gli oggetti sono in cache prima di recuperarli dal DB per evitare materializzazioni inutili.

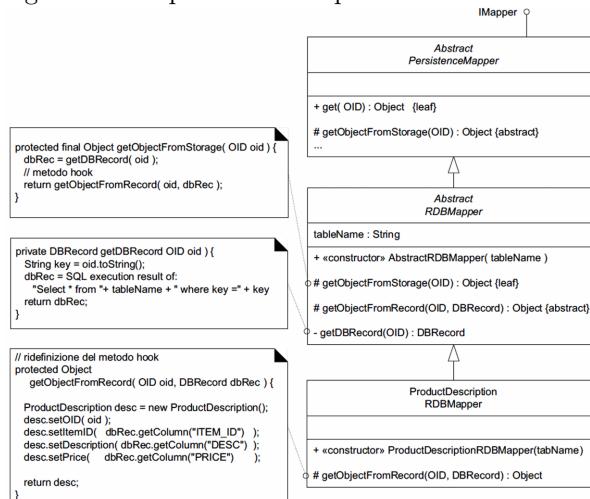
L'algoritmo di materializzazione, in pseudocodice, è strutturato in questo modo:

Algorithm 1 Gestione Cache

```
1: if l'oggetto è in cache then
2:   ritorna l'oggetto
3: else
4:   materializza l'oggetto dal database
5:   salva l'oggetto in cache
6:   ritorna l'oggetto
7: end if
```

Per progettare questa funzionalità, il pattern utile allo scopo è il **Template Method** (visto nella prima esercitazione). Il template method sarà il metodo **get** in una superclasse astratta. Ogni mapper specifico darà poi la sua implementazione di come ottenere i propri oggetti dal repository.

Figura 1: Template Method per la materializzazione

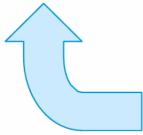


Fino ad'ora, le query SQL sono sempre state inserite dentro al codice dei metodi.

```
protected Object getObjectFromStorage(OID oid) {  
    ...  
    dbRec = execStatement("select * from PROD_SPEC where key = " + oid.toString());  
    ProductSpecification ps = new ProductSpecification();  
    ps.setPrice(...);  
    ...  
    return ps;  
}
```

Per migliorare l'implementazione, si può mantenere una classe **Singleton** dove memorizzare tutte le query SQL necessarie (RDBOperations).

```
class RDBOperations {  
    public ResultSet getProdSpecData(OID oid) {...}  
    public ResultSet getSaleData(OID oid) {...}  
    ...  
}  
  
class ProdSpecRDBMapper extends AbstractPersistenceMapper {  
    protected Object getObjectFromStorage(OID oid) {  
        ResultSet rs = RDBOperations.getInstance().getProdSpecData(oid);  
        ProductSpecification ps = new ProductSpecification();  
        ps.setPrice(rs.getDouble('PRICE'));  
        ...  
        return ps;  
    }  
}
```



Risulta vantaggioso isolare le query in modo da rendere più facile la manutenzione e migliorare la performance.

5.16 Transazioni e Stati transazionali

Una **Transazione** è un'unità di lavoro i cui compiti devono essere completati tutti con successo, oppure nessuno di essi deve essere completato.

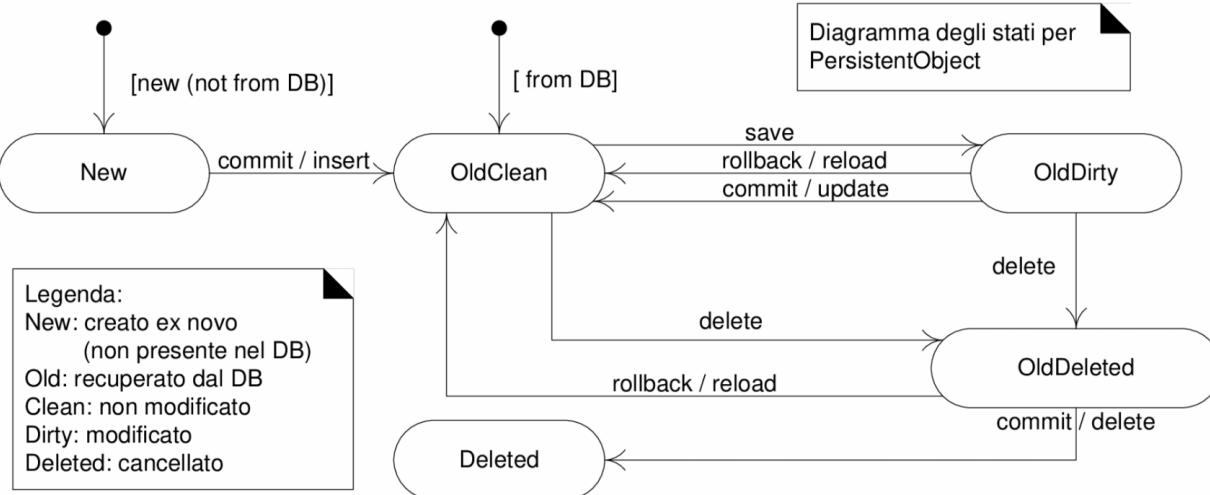
Quando lavoriamo su oggetti persistenti (li inseriamo, cancelliamo o aggiorniamo), questi non vengono aggiornati subito nel database, bisogna quindi eseguire un'operazione di commit esplicita. Possiamo definire **3 luoghi** in cui possono trovarsi gli oggetti persistenti:

1. nell'applicazione
2. nel database
3. nella cache

Si possono anche definire gli **stati** associati agli oggetti persistenti:

1. **New**: appena creato e non ancora presente nel DB
2. **Old**: recuperato dal DB (quindi creato in precedenza)
3. **Clean**: non modificato
4. **Dirty**: modificato
5. **Deleted**: cancellato

Figura 1: Diagramma degli Stati



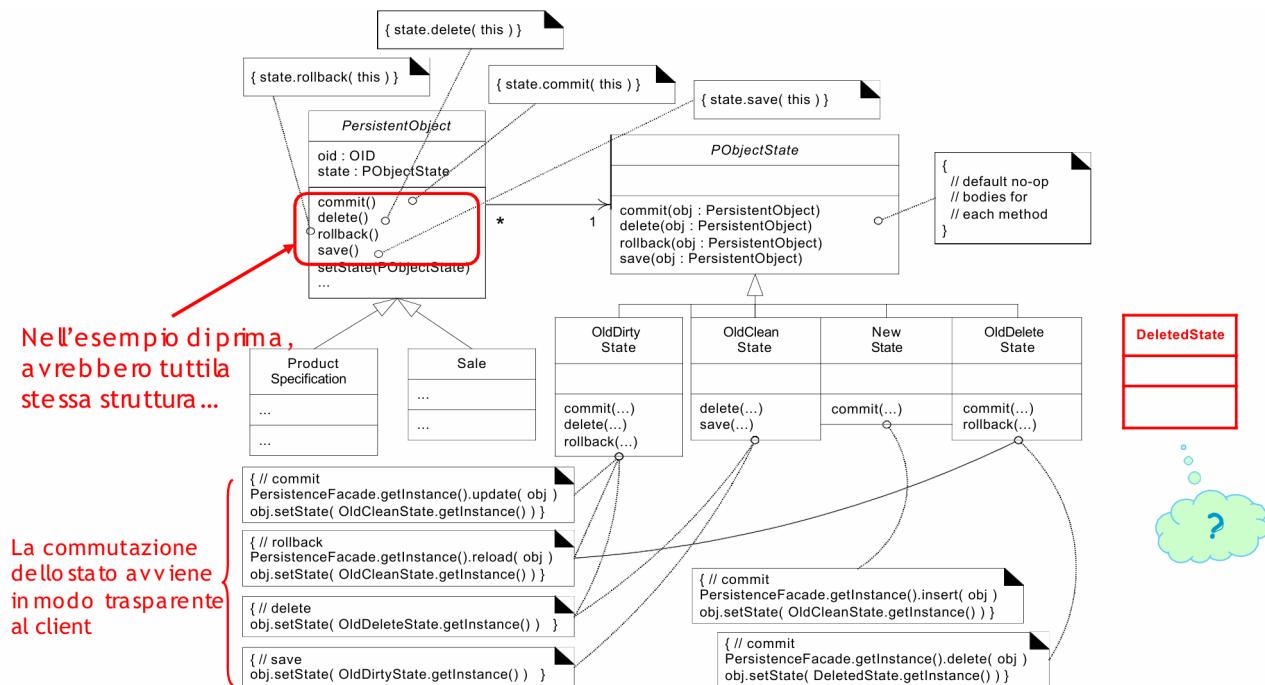
Nota: quando salviamo o cancelliamo un oggetto persistente, questo non viene immediatamente salvato/cancellato dal DB, ma vanno in uno stato appropriato (OldDirty o OldDeleted) in attesa di un eventuale rollback (annullo l'operazione) o commit (procedo con l'operazione).

Il comportamento di **commit** e **rollback** sono molto simili tra loro (in termini di codice)

```
public void commit() {
    switch (state) {
        case OLD_DIRTY: ...;
        break;
        case OLD_CLEAN: ...;
        break;
        ...
    }
}

public void rollback() {
    switch (state) {
        case OLD_DIRTY: ...;
        break;
        case OLD_CLEAN: ...;
        break;
        ...
    }
}
```

Per evitare una ripetizione di codice, usiamo il pattern **State**:



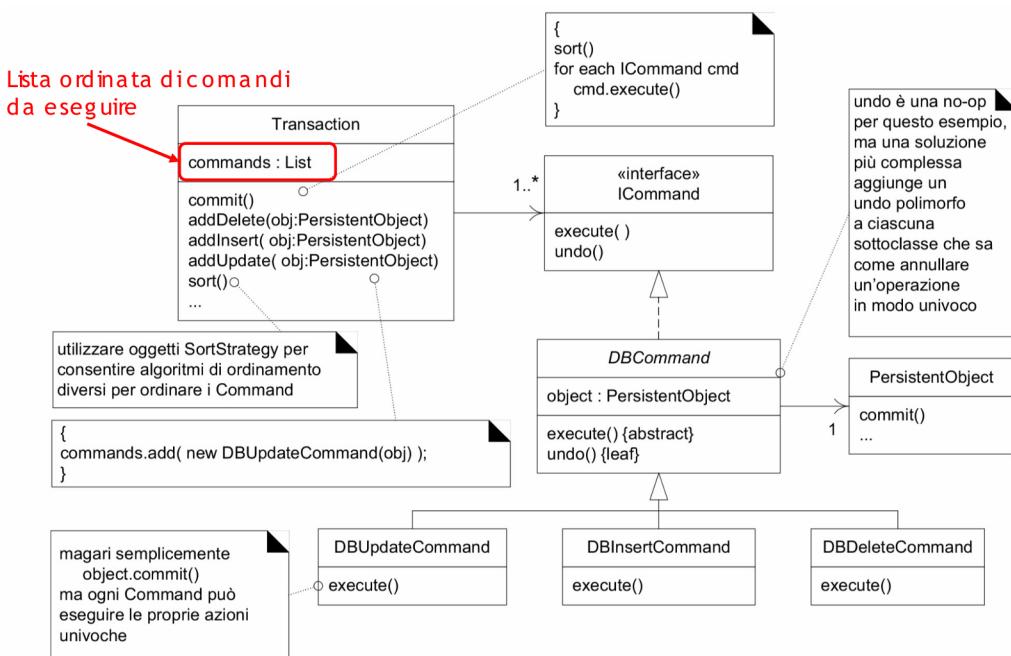
Ricordando cosa fa lo State, crea delle classi che indicano gli stati, le classi implementano un'interfaccia comune. Questa interfaccia delega le operazioni all'oggetto **contesto** in base al suo stato corrente.

Nell'esempio sopra il *PersistentObject* rappresenta l'oggetto **Context**, mentre *PObjectState* l'interfaccia **State**. Per ogni stato concreto, definiamo le operazioni appropriate.

5.16.1 Modellazione delle operazioni di una transazione

Le transazioni, ovviamente, possono essere molteplici. In base all'ordine in cui vengono eseguite le operazioni all'interno di una transazione, si può influenzare la performance della transazione. Bisogna quindi trovare una soluzione, ovvero un modo per poter riordinare tutte le operazioni uguali prima di essere eseguiti. In questo ambito, interviene il pattern **Command**.

Ricordando cosa fa Command, definisce per ciascun compito una classe che implementa un'interfaccia comune, a differenza di State che rappresentava uno stato, **nel Command ciascuna classe rappresenta un comando** e le azioni diventano oggetti.



Nell'esempio, la classe **Transaction** è l'**invoker**, ovvero la classe che richiede all'interfaccia ***ICommand*** le operazioni da eseguire. L'interfaccia riceve le operazioni e la classe **DBCommand** (la **ConcreteCommand**) incapsula le operazioni, per poi farle eseguire dal **Receiver**, ovvero la **PersistentObject**.

5.16.2 Lazy materialization con l'uso di Proxy

Ricordando la definizione di materializzazione, si tratta della traduzione di un record di un DB a un oggetto.

A volte si vuole evitare il processo di materializzazione, per questioni di performance e a meno che non sia strettamente necessario. Si può risolvere questo problema tramite un processo di materializzazione "ritardata", ovvero la *Lazy Materialization*.

Il pattern utile allo scopo è il **Virtual Proxy**.

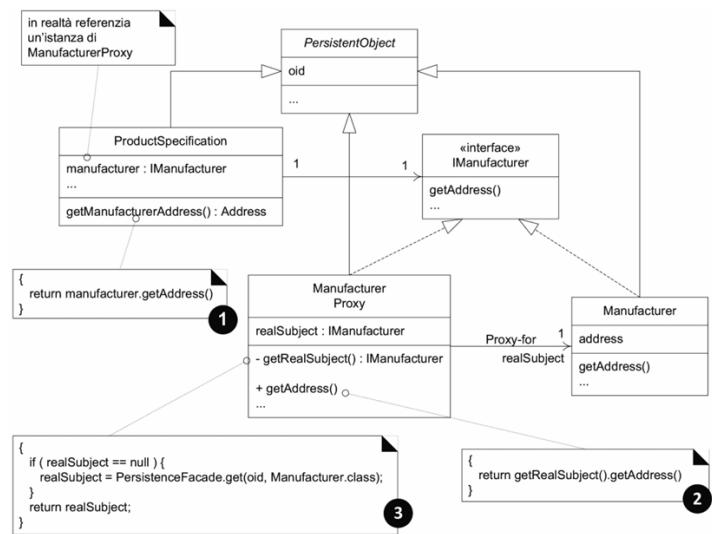
5.16.3 Design Pattern Virtual Proxy

Il Virtual Proxy è un Proxy per l'oggetto "reale", questo proxy virtuale materializza l'oggetto solo la prima volta a cui gli si fa riferimento, rappresentandoli come oggetti leggeri che fungono da sostituto all'oggetto reale (che possono essere più pesanti).

In sostanza, il Virtual Proxy funge da sostituto dell'oggetto reale, ma in un formato più leggero. Sarà questo sostituto a gestire l'accesso all'oggetto reale e a caricarlo in memoria solo quando è strettamente necessario, in modo da evitare materializzazioni inutili.

Nell'esempio, Manufacturer è il nostro oggetto reale, **Manufacturer Proxy** è il suo **Virtual Proxy**, quindi un suo sostituto che ne gestisce l'accesso e stabilisce se caricarlo in memoria o meno. L'interfaccia comune *IManufacturer* permette al client di interagire con il Virtual Proxy senza sapere che si tratta di un suo sostituto.

Bisogna fare una considerazione, dato che parliamo di materializzazione ovviamente è coinvolto anche il pattern **DatabaseMapper**, che in questo esempio è rappresentato dalla classe **ProductSpecification**, infatti questa crea un'istanza di *IManufacturer*, con lo scopo di decidere quali oggetti materializzare subito e a quali invece può ritardare questo processo. Si parla quindi di **Eager** e **Lazy Materialization**.



5.16.4 Eager vs Lazy Materialization

```
// EAGER MATERIALIZATION per MANUFACTURER
class ProductSpecificationRDBMapper extends
    AbstractPersistenceMapper {
protected Object getObjectFromStorage( OID oid ) {
    ResultSet rs =
        RDBOperations.getInstance().
            getProductSpecificationData( oid );
    ProductSpecification ps = new ProductSpecification();
    ps.setPrice( rs.getDouble( "PRICE" ) );
}

// Eager
String manufacturerForeignKey = rs.getString( "MANU_OID" );
OID manuOID = new OID( manufacturerForeignKey );
ps.setManufacturer( (Manufacturer) PersistenceFacade.
    getInstance().get( manuOID, Manufacturer.class ) );
}
}

// LAZY MATERIALIZATION per MANUFACTURER
class ProductSpecificationRDBMapper extends
    AbstractPersistenceMapper {
protected Object getObjectFromStorage( OID oid ) {
    ResultSet rs =
        RDBOperations.getInstance().
            getProductSpecificationData( oid );
    ProductSpecification ps = new ProductSpecification();
    ps.setPrice( rs.getDouble( "PRICE" ) );
}

// Lazy
String manufacturerForeignKey = rs.getString( "MANU_OID" );
OID manuOID = new OID( manufacturerForeignKey );
ps.setManufacturer( new ManufacturerProxy( manuOID ) );
}
}
```

La differenza tra i due approcci sta nel caricamento in memoria del record in oggetto. Nel primo caso, avviene immediatamente non appena i dati richiesti sono necessari, nel secondo caso il caricamento viene ritardato fino a quando l'oggetto non è strettamente necessario.

Guardando il codice, entrambi gli approcci iniziano col prelevare la chiave esterna del manufacturer, per poi creare un'ID da associare all'oggetto (che verrà caricato in memoria) di quella chiave esterna (materializzazione: da record a oggetto).

1. L'approccio **Eager** preleva direttamente l'oggetto Manufacturer dal database (...get(manuOID, Manufacturer.class)) per poi settarlo al Manufacturer oggetto.
2. L'approccio **Lazy** agisce diversamente, crea prima un Virtual Proxy (ManufacturerProxy) dotato di identificativo unico (manuOID), settandolo nell'istanza di Manufacturer (ps). ManufacturerProxy rappresenta il sostituto di Manufacturer ma in formato più leggero e senza caricarlo in memoria direttamente.