

## ANALISI OBJECT ORIENTED

L'OBIETTIVO è: coprire come individuare le classi di analisi (per esempio per una prima bozza), e partire da una Use Case View

Immediatamente, deve essere ovvia la risposta alle seguenti domande:

- Le classi di analisi hanno tutte / sempre la STESSA FUNZIONE?

→ (NO) → Abbiamo visto già con i principi GRASP che hanno RUOLO e RESPONSABILITÀ diverse.

### • MODELLO BCE:

Il modello di processo RUP (Rational Unified Process) suggerisce agli analisti di assegnare ulteriori responsabilità alle classi d'analisi, oltre a GRASP, suddividendo così in 3 macro-area:

- BOUNDARY
- CONTROL
- ENTITY

} Modello BCE

### • MA PERCHÉ SI HA BISOGNO DI MODELLARE ASPETTI NON PREVISTI DAL LINGUAGGIO?

- (1) Il dominio di applicazione o il committente hanno bisogno di esprimere concetti generali che non sono inclusi nel linguaggio di modellazione (UML);
- (2) Il modellista ha bisogno di rappresentare aspetti legati allo specifico processo di sviluppo collettivo, dando DIFFERENTI INTERPRETAZIONI ad uno stesso elemento del linguaggio in base alle fasi del processo;
- (3) Il modellista vuole RAFFINARE i modelli esistenti in funzione del contesto, risolvendo eventuali ambiguità e per una generazione del codice più efficiente ed efficace.

\* In UML è stato introdotto il concetto di STEREOTIPO, che sono il principale meccanismo per la personalizzazione e l'ESTENSIONE del linguaggio UML. Uno stereotipo estende il significato di un elemento già esistente del linguaggio, dando vita ad un NUOVO ELEMENTO DEL LINGUAGGIO, introducendo:

- ulteriori interpretazioni per un elemento;
- ulteriori caratteristiche di un elemento;
- ulteriori relazioni con altri elementi.

Fornero uso dei seguenti stereotipi: <<boundary>>, <<control>>, <<entity>>.

• **BOUNDARY**: mediano l'INTERAZIONE tra sistema e ambiente esterno; regolano l'INTERAZIONE con gli ATTORI e rappresentano elementi al confine del sistema, secondo il punto di vista dell'attore.

- Di norma, ci si aspetta un numero di boundary pari al numero di coppie (attore, caso d'uso) collegati da une frecce nello Use Case Diagram.
- È una VISTA parziale delle funzionalità e operazioni che l'attore può fare in relazione al caso d'uso in questione.
- Enfatizzare le funzionalità che l'attore vuole poter eseguire, NON concentrarsi su dettagli implementativi della GUI!

• **CONTROL**: coordinare il COMPORTAMENTO durante un CASO D'USO del sistema; rappresentano comportamenti dipendenti dall'interazione offerta dal sistema e in particolare il comportamento descritto dal caso d'uso;

- Evidenziano che COSA AVVIENE quando ricevo un certo SINTACOLO!
- La descrizione delle classi control è indipendente dal modo di attivazione dell'interazione → il punto di attivazione è sempre lo stesso: ho ricevuto un messaggio da parte di una boundary!

- Dovrebbero scaturire dall'analisi di DOMINIO del sistema; ci si aspetta (almeno) **1** classe control per ogni caso d'uso.
- Definiscono il FLUSSO DI ESECUZIONE e di controllo del caso d'uso.

• **ENTITY**: rappresentano le astrazioni chiave (KEY ABSTRACTIONS) del DOMINIO del sistema;

- sono INDEPENDENTI dall'ambiente di esecuzione;
- dipendono dall'ANALISI DEL DOMINIO del sistema (glossario, use case, business model, etc...);
- modellano il comportamento di un'entità di dominio, incapsulando un insieme COTESO di dati;
- di solito NON prevedono attori, o meno che tali attori non abbiano una loro rappresentazione di cui tenere traccia nel dominio e quindi nel sistema.

\* BCE porta all'individuazione di un VOPC (View Of Participating Classes) per OGNI CASO D'USO! Tali VOPC costituiscono un Diagramma delle Classi di ANALISI, ottenuto basandosi solo sul Dominio del PROBLEMA! Non ci sono "input" del Dominio delle Soluzioni! Questi infatti porteranno all'introduzione di classi di ingegnerizzazione e ad un Diagramma delle Classi di Progettazione!

## PATTERN GOF

Immaneutto, diamo la definizione di cosa sia un pattern:

**PATTERN := PROBLEMA RICORRENTE + SCHEMA DI SOLUZIONE**

Abbiamo già incontrato dei pattern in precedenze:

- pattern delle metamorfosi → come gestire il CAMBIAMENTO di RUOLO nel TEMPO?
- responsabilità GRASP → a chi assegnare le responsabilità di determinate azioni? (Legge di Demetra)
- pattern BCE → come individuare e collegare tra loro le classi?  
Come passare da uno schema e così d'uso ad un'altra strutturale delle classi del sistema?
- polimorfismo → come far compiere un'operazione in maniera differente a RUN-TIME, facendole collegare a metodi diversi?
- **DESIGN PATTERN:**

E' un pattern con una descrizione un po' più precisa e raffinata, così come introdotte dai Gang of Four (GOF).

- E' una descrizione di un PROBLEMA RICORRENTE nella PROGETTAZIONE.  
Inoltre, ad ogni problema viene associato:
  - un NOME;
  - una SOLUZIONE che puo' essere applicata in differenti circostanze, anche eterogenee tra loro;
  - una DISCUSSIONE sulle conseguenze e sulle variazioni che conseguono l'applicazione delle soluzioni.

- \* I design pattern NON sono "isolati"; bensì sono pensati per essere CORRELATI e COMBINATI tra di loro, applicandoli in cascata;
- \* Il punto focale è FORMALIZZARE e strutturare PROBLEMI RICORRENTI e supportare l'applicazione di TECNICHE CONSOLIDATE (non fornir nuove idee di progettazione)!

611-1038

## MVC : MODEL - VIEW - CONTROLLER

- Differenze con BCE:

### BCE

- Pattern di ANALISI
- Progettazione a CASI D'USO e del punto di vista dell' ATTORE
- Ancore vicino al Dominio del Problema
- Individua le MACRO-CLASSI del sistema

### MVC / MVP

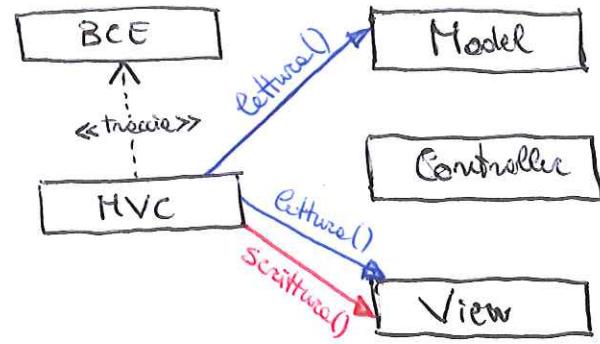
- Pattern di PROGETTAZIONE, è un raffinamento del BCE
- Introduce aspetti relativi all' IMPLEMENTAZIONE e all' INGEGNERIZZAZIONE del sistema
- Grande passo in avanti verso il Dominio delle Soluzioni
- Individua le CLASSI vere e proprie, anche quelle per ingegnerizzare.

- MVC:

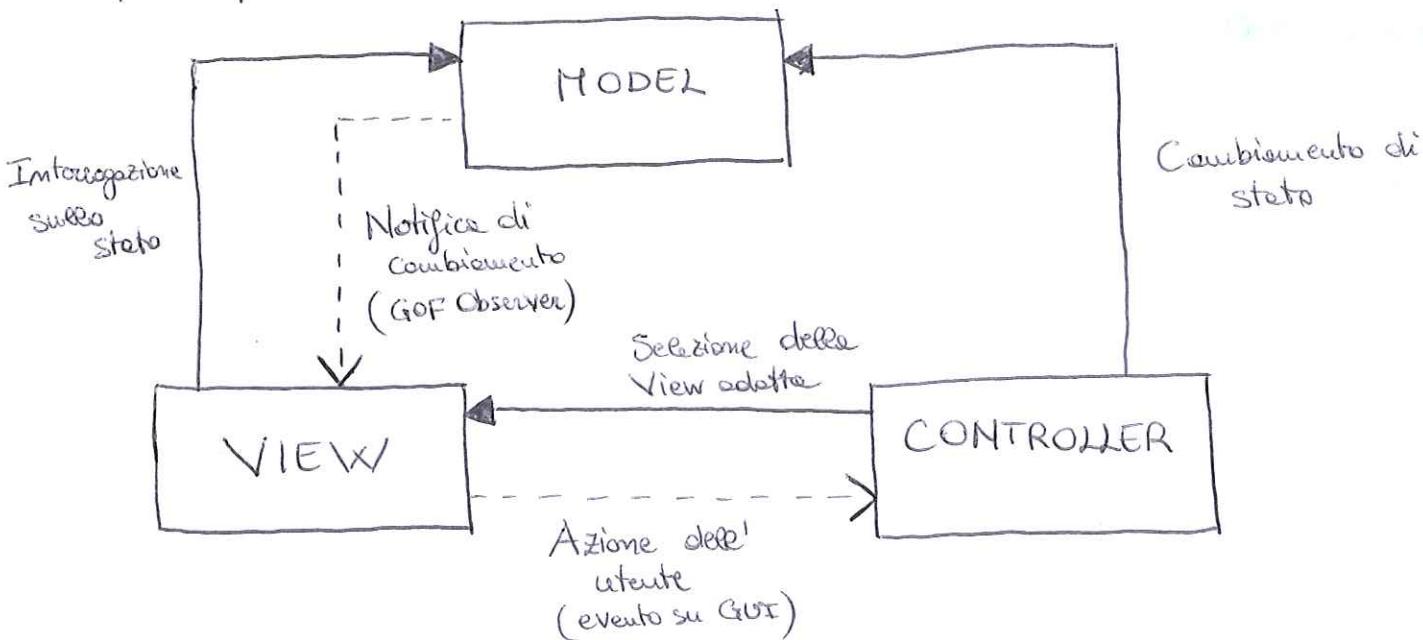
È un pattern che traccia da BCE.

Consente:

- Lettura(): mediante la View o anche DIRETTAMENTE dal Model
- Scrittura(): SOLO mediante dalla View, passa per il Controller e arriva sul Model (e poi sulla persistenza, se necessario).



Agisce secondo il seguente schema, in cui le frecce piene rappresentano invocazioni di metodi, mentre quelle tratteggiate degli eventi o input dell'utente, come per esempio la pressione di un bottone sulla GUI:



→ : Invocazione metodo

→ : Evento/ input dell'utente

## • VIEW (vs Boundary):

La View è un raffinamento delle Boundary nel paradigma BCE:

- BOUNDARY: ha il compito di esportare le operazioni che l'utente può effettuare col sistema nell'ambito di un determinato caso d'uso.  
Tuttavia, indica solo QUALI, cioè il COSA, NON IL COME!
- VIEW: implementa integralmente una boundary! Ma è responsabile anche per altre cose, cioè per:
  - logica di presentazione dei dati;
  - costruzione e gestione di una GUI! Questo fa la differenza, perché descrive il COME è gestita l'interazione con l'utente;
  - realizza l'eccisione dei dati in input dell'utente: la scrittura deve passare per forze da qui; può, in alcuni casi, accedere in lettura direttamente al model, per poter rappresentare in logica di presentazione i dati aggiornati
  - se non ne è responsabile il controller, può gestire la conversione dei dati del formato esterno (presentazione) al formato interno (rappresentazione nel model, come key abstractions).

La View deve quindi mantenere aggiornati i dati che presenta. Ci sono 2 modi:

1) **PUSH MODEL**: applicazione del pattern GoF OBSERVER; in questo scenario, il model è un ConcreteSubject e la View è un ConcreteObserver del model, cioè si "iscrive" come observer del model.

E' il model che notifica la View quando il proprio stato cambia! I cambiamenti sono proposti REAL TIME in presentazione.

2) **PULL MODEL**: E' la View a chiedere i dati del model (direttamente o passando per un controller) quando lo ritiene opportuno.

\* Di solito, la View viene vista insieme come boundary + GUI; tuttavia, è possibile vederla in modo leggermente differente:

- si separa la boundary dalle GUI, e la boundary continua ad avere l'eccezione che avviene in BCE, cioè esportare le possibili azioni che l'utente può svolgere.
- La GUI si occupa solo di eventi del punto di vista grafico, e INVOCA i metodi offerti delle "boundary".

## ● CONTROLLER:

Si suddivide in 2 controller per semantica e responsabilità: controller grafico e controller applicativo.

- CONTROLLER GRAFICO: gestisce l'interazione tra la View e il Controller Applicativo, così da giungere fino al Model per la realizzazione di una funzionalità.

- realizza il mapping tra l'input dell'utente e i processi eseguiti dal model;
- può avere le responsabilità di istanziare i controller applicativi giusti.
- seleziona e crea le istanze di View richieste
- può fare la conversione da formato esterno dei dati a formato interno.

- CONTROLLER APPLICATIVO: coordina l'esecuzione del caso d'uso.

- raffina direttamente il concetto di Control di BCE (e in GRASP)
- implementa la **LOGICA DI CONTROLLO** dell'applicazione
- ha le responsabilità di gestire un'azione dell'utente (sulla View), traducendole in una o più azioni eseguite sulle istanze del Model
- deve essere il più possibile STATELESS:  
quando termina l'esecuzione del caso d'uso, il controller deve poter "morire", essere deallocato. Al termine del caso d'uso, lo stato è stato trasferito alle entity del model.

Dunque, il controller deve essere tale che ogni volta inizia il caso d'uso associato, inizia sempre allo stesso modo, non è influenzato dagli stati, dalle istanze precedenti. I valori degli attributi NON vengono mantenuti tra un'interazione e l'altra.

## \* In un UNICO BUNDLE o DISACCOPPIATI?

- 1) UNICO BUNDLE: il controller grafico implementa direttamente la logica di controllo e sceglie e coordina la View.

Dunque, la View corrisponde ad una boundary BCE.

→ POCO FLESSIBILE, HIGH COUPLING, SCARSO RIUSO e MODULARITÀ X

- 2) DISACCOPPIATI: il controller grafico invoca operazioni sul controller applicativo.

Dunque: boundary BCE = View + controller grafico

→ LOW COUPLING, logica del sistema indipendente dalla GUI, FLESSIBILITÀ, HIGH COHESION, MODULARITÀ, RIUSO ✓

## • MODEL:

Costituisce la **RETE DI ENTITA'** partecipanti alle logiche di applicazione, facenti parte del DOMINIO REALE e/o del DOMINIO DI BUSINESS del sistema. Mappano le entity BCF. Per ogni entità:

- descrive i comportamenti esposti in forme di operazioni
- definisce le regole di business per l'interazione coi dati
- realizza e gestisce le relazioni con le altre entità
- incapsula lo stato dell'entità (attributi PRIVATI)
- un buon Model non dovrebbe esporre direttamente lo stato di un oggetto

• Può avere le responsabilità di **NOTIFICARE** ai componenti View eventuali aggiornamenti verificatisi (**PUSH MODEL**, pattern Graf Observer), oppure farlo in seguito a richieste del Controller.

## • BEAN:

Abbiamo in MVC che la View può INTERROGARE direttamente il Model: sostanzialmente ha un riferimento al Model e fa le get() dei dati.

In tal modo, quando cambia il Model, dovrà modificare anche tutte le View, per mantenere consistenti i dati e la loro presentazione → **HIGH COUPLING**

Bisogna disaccoppiare View e Model; cioè **DISACCOPPIARE NETTAMENTE LA LOGICA DI PRESENTAZIONE DALLA LOGICA DI RAPPRESENTAZIONE DEL DOMINIO NEL SISTEMA.**

Per questo motivo si introducono le classi **BEAN**:

- si interpongono tra View e Model
- disaccoppiano le Boundary e le Entity
- duplicano il concetto di entity e fanno da container stagne, da veicolo, per il passaggio di informazioni delle boundary (controller grafico) al controller applicativo e viceversa
- effettuano la VALIDAZIONE SINTATTICA sull'inserimento dei dati in input
- eventuale gestione dei riferimenti per i dati in visualizzazione (pull o push)  
model

\* Se cambia il Model, devo cambiare solo le Bean!

\* Boundary e Controller si scambiano dati secondo Bean, non secondo Model!

• Una bean generalmente realizza un **POJO** (**Plain Old Java Object**):

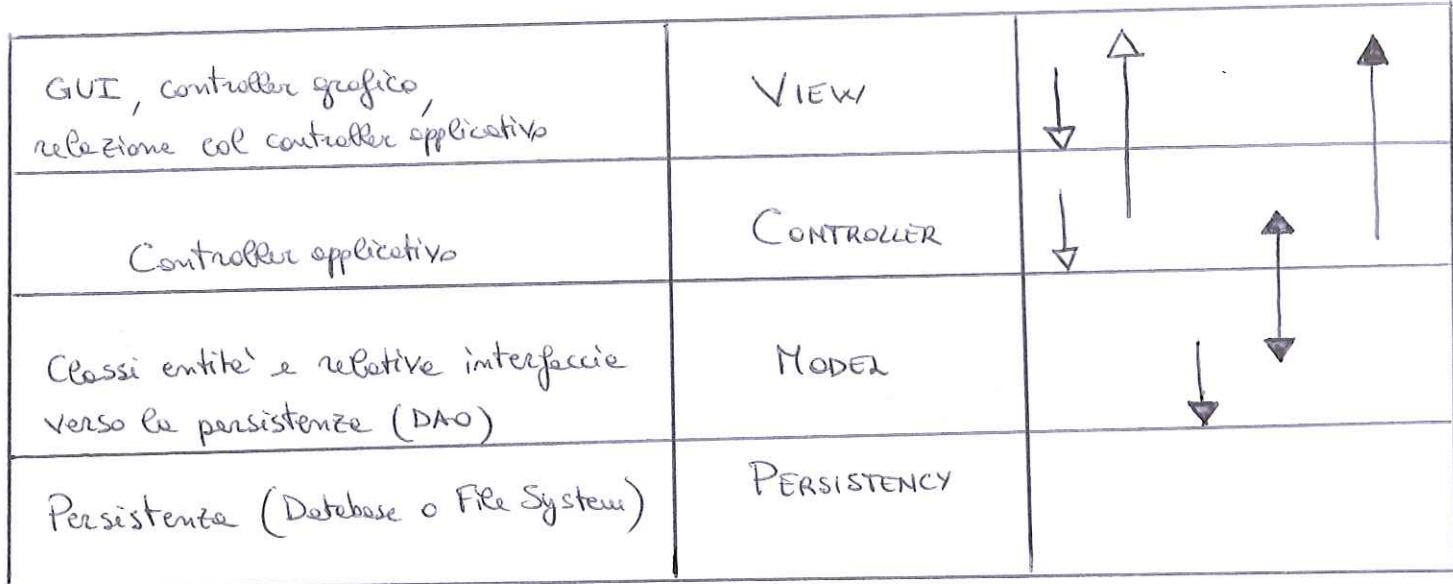
- attributi privati, destinati a contenere solo dati di I/O;
- metodi pubblici sono solo GETTER e SETTER;
- l'accesso agli attributi avviene solo mediante getter e setter;
- metodi PRIVATI per il CONTROLLO SINTATTICO.

## MVC in architettura STANDALONE:

- Tutti e 3 gli strati (View, Controller e Model) RISIEDONO INTERAMENTE sulle macchine dell'utente, Beta Client
  - Condividono lo stesso spazio di memoria messo a disposizione dalla JVM, ed è più semplice passare riferimenti ad istante.

Nel seguente schema:

- → : Lettura / scrittura mediata da BEAN
- → : Lettura / scrittura diretta



## MVC in architettura WEB:

Le app web sono pensate come mativamente distribuite, accessibili da remoto tramite dei THIN CLIENTS (e.g. Browser Web).

\* Controller e Model risiedono Server-side. La View è a metà strada; è istanziata server-side, ma PRESENTATA client-side!

→ Diventano necessarie delle classi di supporto / appoggio per la gestione di I/O (bean)

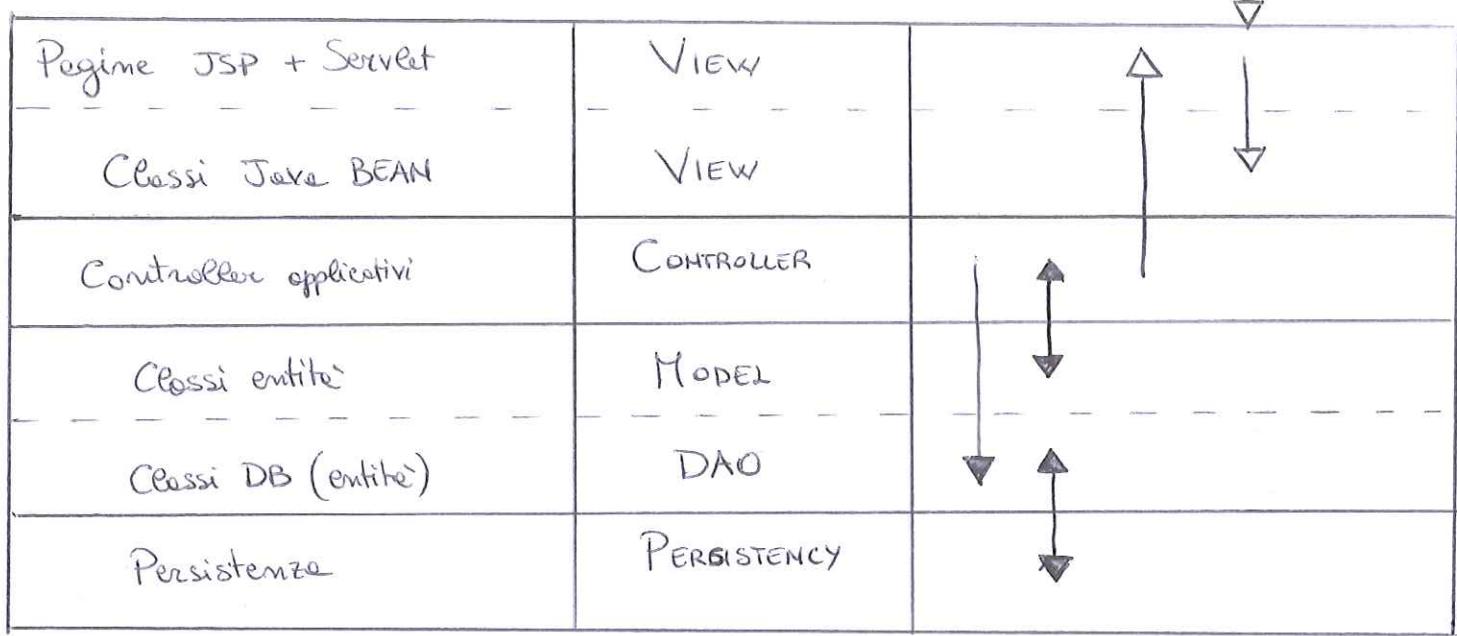
LE CLASSI BEAN SONO NECESSARIE!

\* Normalmente, la View è realizzata con le tecnologie JSP (Java Server Pages): una pagina JSP viene elaborata (la parte Java) dal Server per ottenere come risultato una view HTML da restituire come risposta al Client.

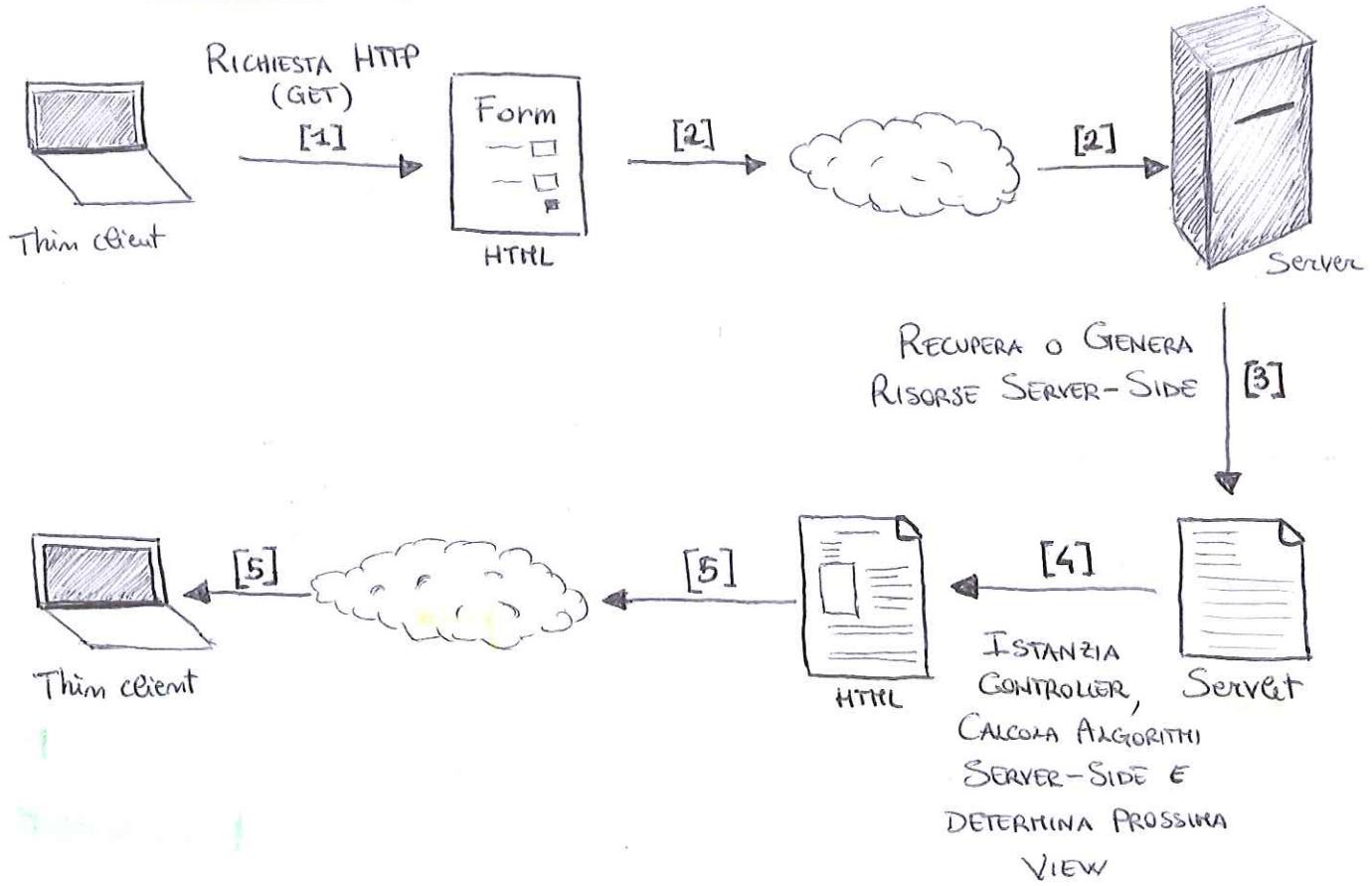
Sostanzialmente, insieme alla servlet, svolge il ruolo del CONTROLLER GRAFICO.

\* Al contrario dell'architettura stand-alone, il controller <sup>(si può)</sup> NON è responsabile dell'identificazione delle View da presentare. Perciò, si introducono le SERVLET.

\* SERVLET: oggetto scritto in Java che opera in un Application Server Web, processa le richieste HTTP del client, elabora le pagine JSP e ISTANZIA IL CONTROLLER APPLICATIVO; se non lo fa le pagine JSP, può anche selezionare le View successive.



### • WEB, CONTESTO GENERALE :



## PERSISTENZA

In linguaggi di programmazione Object-Oriented, gli oggetti / istanze hanno il seguente CICLO DI VITA comune:

- 1) allocazione di uno spazio di memoria ("new") e invocazione di uno dei costruttori;
- 2) deallocazione, che può essere esplicita (C++) , implicita tramite GARBAGE COLLECTOR (Java) oppure avvenire a causa della terminazione dell'app.

- Tuttavia, ci sono scenari in cui si desidera che:

- lo stato di un oggetto continui ad esistere (PERSISTA) anche dopo la terminazione dell'esecuzione dell'applicazione;
- un'applicazione possa configurare il suo contesto di esecuzione attraverso il RIPRISTINO dello stato di 1 o più oggetti;
- lo stato di un oggetto possa migrare tra un nodo e l'altro di una applicazione distribuita.

→ Queste necessità possono essere in parte soddisfatte utilizzando FILE o DBMS, ma con i seguenti svantaggi:

- FILE: richiede una gestione esplicita delle convenzioni di memorizzazione e del formato dei dati utilizzati;

- DBMS: generalmente non adatto a sistemi di dimensioni limitate. Inoltre, c'è da gestire il MISMATCH tra le rappresentazioni Object-Oriented del dominio e le rappresentazioni specifiche da coltivare nel DB.

\* Per questi motivi, Java sfrutta il concetto di STREAM per proporre un modo semplice per la persistenza di Istanze, in particolare per persistere su FILES, che prende il nome di serializzazione/deserializzazione.

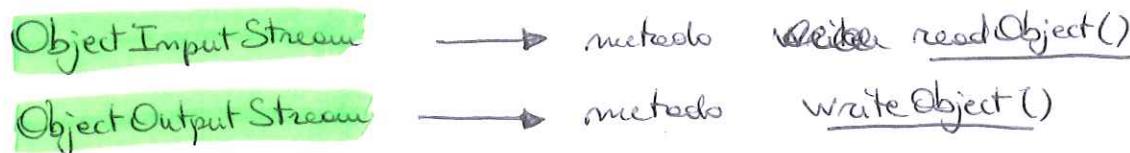
- SERIALIZZAZIONE / DESERIALIZZAZIONE:

- SERIALIZZAZIONE: salvataggio dello stato di un oggetto, che viene convertito in una RAPPRESENTAZIONE BINARIA, che posso essere scritte sequenzialmente su uno stream.

- DESERIALIZZAZIONE: ripristino dello stato di un'istanza, convertendo, tramite un algoritmo Java, da byte letti sequenzialmente da uno stream a Java object.

Uno stream di byte creato tramite la serializzazione è PLATFORM-INDEPENDENT!

- Le classi che supportano la serializzazione e deserializzazione di oggetti sono:



\* La serializzazione di un oggetto prevede che venga salvato non solo l'oggetto stesso, ma anche tutti i RIFERIMENTI contenuti in esso e così via.

### Cosa è SERIALIZZABILE in Java?

- 1) Tutti i TIPI DI DATO PRIMITIVI sono serializzabili;
  - 2) Le sole CLASSI che implementano l'interfaccia Serializable, la quale non ha alcuna operazione, è solo una TAGGING INTERFACE.
- Quando si serializza o deserializza una classe che implementa Serializable, vengono salvati gli attributi primitivi e, inoltre, viene percorso l'albero dei RIFERIMENTI e si cerca di serializzare anche le istanze riferite!  
Se anche le istanze riferite implementano Serializable, il meccanismo si innesta A CATENA, altrimenti si salva solo il loro stato costituito da attributi di tipo di dato primitivi.
  - In Java, il serializzatore e il deserializzatore lavorano su oggetti allocati nell'HEAP di memoria.

\* Però, le VARIABILI/ATTRIBUTI DI CLASSE (static) non sono nell'heap, bensì nel runtime constant pool e vengono istanziati sempre e soltanto al momento del caricamento dinamico della classe e inizializzati sempre con il valore statico definito nella classe.

→ NON sono oggetto di serializzazione / deserializzazione!

\* Se si vuole salvare / ripristinare un valore, bisogna prevedere dei MECCANISMI ESPICITI di salvataggio e ripristino!

- Se una classe è NON serializzabile, ottengo un errore → NotSerializableException().  
Senza modificare la classe, quali sono le SOLUZIONI?
  - 1) Creare una classe figlia che sia Serializable, con dei costruttori che settino lo stato delle classi parent in un determinato modo (costruttori public e protected)
  - 2) Gestione dell' Eccezione!

## ECCEZIONI

Nel software, possono presentarsi diversi tipi di errore in momenti differenti:

- ERRORI DI SINTASSI : a tempo di compilazione
- ERRORI DI COMPORTAMENTO ; in fase di test delle singole unità software
- ERRORI DI INTERAZIONE : in fase di test di integrazione delle varie unità software

Non si può quasi mai essere certi delle 'esse' di errori in un sistema software, perciò è opportuno progettare e sviluppare delle **LOGICHE DI CONTROLLO** e **GESTIONE DEGLI ERROTI**, parallelamente alla progettazione dei comportamenti desiderati del sistema.

- Storicamente, questo problema è stato gestito tramite convenzioni:
  - in C / Unix-like, è gestito in base al **VALORE DI RITORNO**
  - flag globali, da esaminare prima di consumare risultati di elaborazioni
- Linguaggi di progettazione / programmazione esistenti prevedono dei costrutti motivi per la definizione delle **LOGICA DI ERRORE**:
  - UML State Machine : prevedono gli **EXIT POINTS**
  - Java : prevede la classe **java.lang.Exception**

### • ECCEZIONE :

E' un **EVENTO** che si verifica durante l'esecuzione di una sequenza di azioni in **LOGICA DI CONTROLLO** e che ne interrompe il flusso, così come definito dal progettista / sviluppatore.

Esempi di eccezioni:

- si rompe l'Hard Disk
- si eccede ad un indice fuori da un array
- la connessione di rete diventa assente
- La (de) serializzazione di un oggetto non va a buon fine
- si tenta di accedere ad un file protetto in lettura

- Si potrebbero controllare delle eccezioni tramite degli **if-statement** ; ma il codice si complica, diventa illeggibile, poiché le **logiche di controllo** e **le logiche di errore sono meschiate!** → **SCARSA MANUTENIBILITÀ** 
- Bisogna separare la **LOGICA DI CONTROLLO** dalle **LOGICA DI ERRORE**!  
→ In Java, ciò è reso possibile dal costrutto **TRY & CATCH**!

## • TRY & CATCH :

```
try {  
    < istruzioni Logiche Di Controllo >  
} catch (<TipoEccezione> <identificatore>) {  
    < istruzioni Logiche Di Errore >  
}
```

- La clausola TRY definisce un blocco di istruzioni in cui puo' verificarsi un'eccezione; puo' essere eseguito da 1 o piu' clausole catch.
  - La clausola CATCH definisce il tipo di eccezione che viene gestita.
- \* Al verificarsi di un'eccezione, la computazione "salta" alla PRIMA istruzione delle PRIMA clausole CATCH che gestisce un'eccezione del tipo (o sottotipo) di quelle sollevate!
- Dunque, il matching avviene in base al tipo → CATENA DI GENERALIZZAZIONE
- La ~~clausola~~ gestione dell'eccezione è soddisfatta al termine della clausola catch trovata, cioè dopo aver eseguito le istruzioni della logica di Errore.

## DOVE SI VA DOPO IL TRY & CATCH ?

(1) Se l'eccezione viene gestita, quindi si entra in un catch, NON si ritornere' MAI nel blocco Try! → Si continua dalle 1<sup>e</sup> istruzione successive al blocco try/catch;

(2) Se l'eccezione NON viene gestita, cioè non è catturata da alcun catch, essa viene PROPAGATA al chiamante (come un valore di ritorno), il quale a sua volta puo' gestirle oppure propagarle.

Se l'eccezione viene propagata al livello più alto possibile, cioè al Sistema Operativo, tipicamente viene terminata l'esecuzione dell'intera applicazione.

## • FINALLY :

E' una clausola che si puo' aggiungere al blocco try/catch. E' facoltativa se c'è almeno 1 catch. Funziona così, nei vari casi:

- 1) Non viene sollevata alcuna eccezione → il blocco di istruzioni nella clausola finally viene eseguito dopo il try
  - 2) Un'eccezione viene catturata da un catch → la finally viene eseguita dopo il catch
  - 3) Un'eccezione non viene catturata da alcun catch e viene propagata all'interno → la finally viene eseguita prima di ridere il controllo al chiamante
- \* La finally viene eseguita SEMPRE, a meno che non ci sia l'istruzione System.exit (int n);

- Il motivo dell'esistenza delle finally è per garantire congruenze nello stato e per EVITARE CODICE DUPLICATO.

Spesso è utilizzato per liberare risorse utilizzate all'interno del blocco try.

### THROWS :

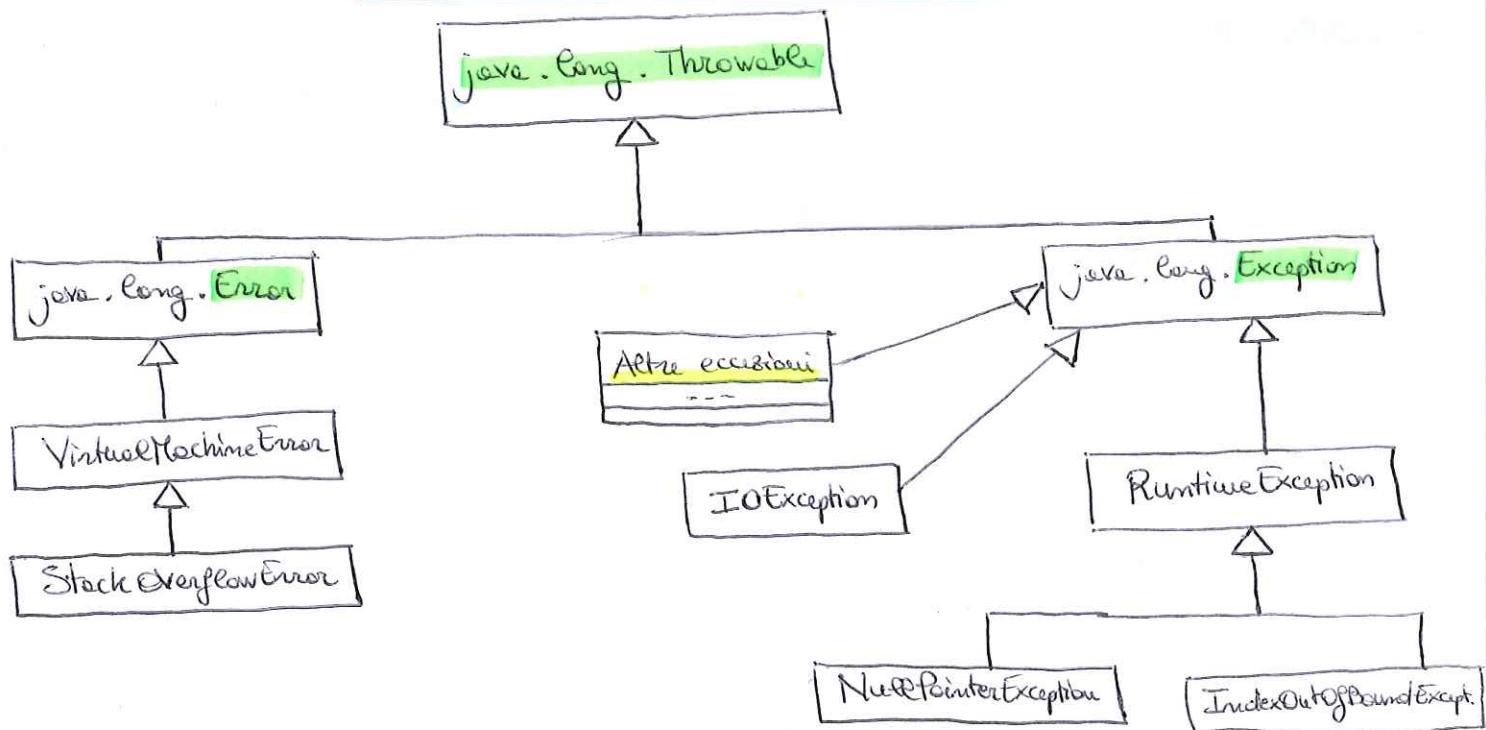
Se un metodo NON è in grado di gestire opportunamente un'eccezione, deve essere marcato come possibile "sorgente d'errore", presentando la clausola THROWS nella sua dichiarazione : `void fooMethod() throws <ExceptionType>`

### ENTRARE IN LOGICA D'ERRORE: comando THROW

Java mette a disposizione il comando THROW (imperativo!) che indica il lancio "volontario" di un'eccezione da parte del programmatore.

E' un'istruzione che comunica alla JVM che deve ENTRARE IN LOGICA DI ERRORE !

### GIERARCHIA DI ERRORI / ECCEZIONI



- ERROTI: Sono anomalie che si verificano all'interno della VIRTUAL MACHINE, in genere non si riesce a gestirli in modo adeguato, poiché sono legati a condizioni anomali e ad anomalie gravi.

Esempi : `StackOverflowError`, `OutOfMemoryError`

- ECCEZIONI: Si verificano in logiche di controllo e si suddividono in 2 categorie:

- java.lang.RuntimeException e sottoclassi : sono dette UNCHECKED, poiché NON è obbligatorio gestirle né segnalarle col `throws`. Si verificano in seguito ad un errore di programmazione. Esempi : `NullPointerException`, `ClassCastException`

- tutte le altre : sono dette CHECKED, poiché E' OBBLIGATORIO GESTIRLE con un costrutto `try...catch` oppure rinviarle al chiamante e dichiararle tramite `throws`. Si verifica quando eccede qualcosa di imprevisto. Esempi : `FileNotFoundException`,

## • CONVERSIONE E CHAINING DI ECCEZIONI :

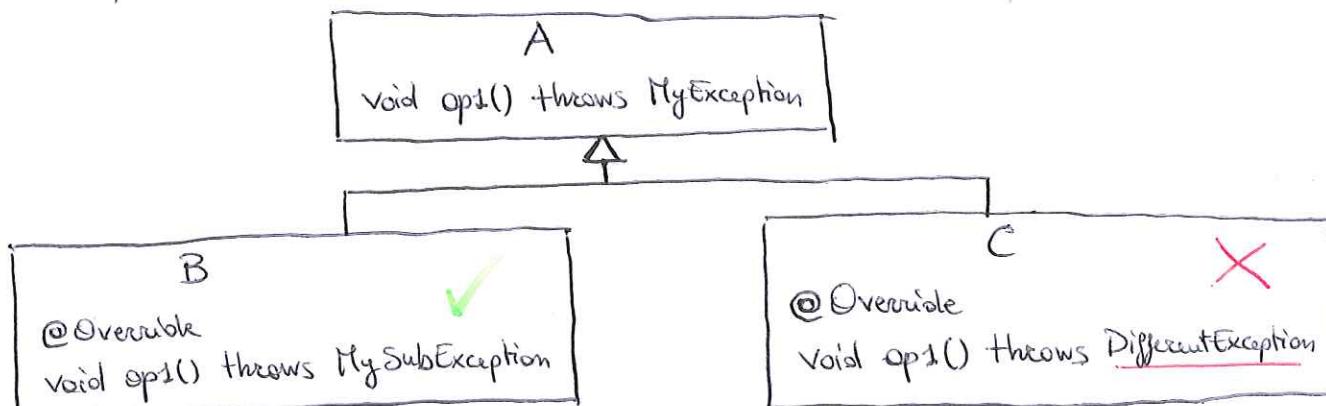
E' possibile gestire un'eccezione semplicemente lanciandone un'altra di tipo diverso.  
Ma a quale scopo? → Questo meccanismo ha un'utilità a LIVELLO APPLICATIVO quando si vuole gestire eccezioni ad un DIVERSO LIVELLO DI ASTRAZIONE, in modo tale che al livello UI l'eccezione possa essere gestita con un approccio più USER FRIENDLY, magari mostrando e schermi dei messaggi significativi per l'utente.

Ci sono 2 strategie:

- **CONVERSION**: l'eccezione originale  $e_1$  viene gestita e convertita lanciandone una nuova  $e_2$ ; il parametro di  $e_2$  sarà  $e_1.getMessage()$ .  
Così facendo, cambia solo il Tipo! NON c'è alcuna legame tra le 2 eccezioni!  
→ Più utile per interazione con l'utente, con l'uso di messaggi user friendly.
- **CHAINING**: l'eccezione originale  $e_1$  viene gestita lanciandone una nuova  $e_2$ , ma stavolta il parametro di  $e_2$  sarà  $e_1.getCause()$ .  
Stavolta c'è LEGATURA tra  $e_1$  ed  $e_2$ , perché  $e_2$  si porta dietro  $e_1$  come CAUSA! (chaining/wrapping).  
→ Utile per fare debugging; con  $e.printStackTrace()$  si può stampare su un log tutte le catene di eccezioni fino alla causa originale.

## • OVERRIDING ED ECCEZIONI :

In caso di override di metodi che fanno il "throws" di eccezioni di tipo CHECKED, si hanno degli impatti anche sulle implementazioni delle sottoclassi.



- B va bene, poiché MySubException "is-a-kind-of" MyException ed è quindi compatibile con le eccezioni gestite dal parent A → NON PROBLEMI CON SOLUZIONI POLIMORFE
- C NON VA BIENNE! DifferentException non è compatibile con le eccezioni gestite da A, quindi non potrai applicare il polimorfismo → **BISOGNA FARE** CONVERSION o CHAINING

- Tale principio assicura che il codice che funziona all'interno delle classi base funzioni anche con qualsiasi oggetto derivato dalla classe base (PRINCIPIO DI SOSTITUIBILITÀ DI LISKOV).

\* Tutto ciò NON vale per i COSTRUTTORI!

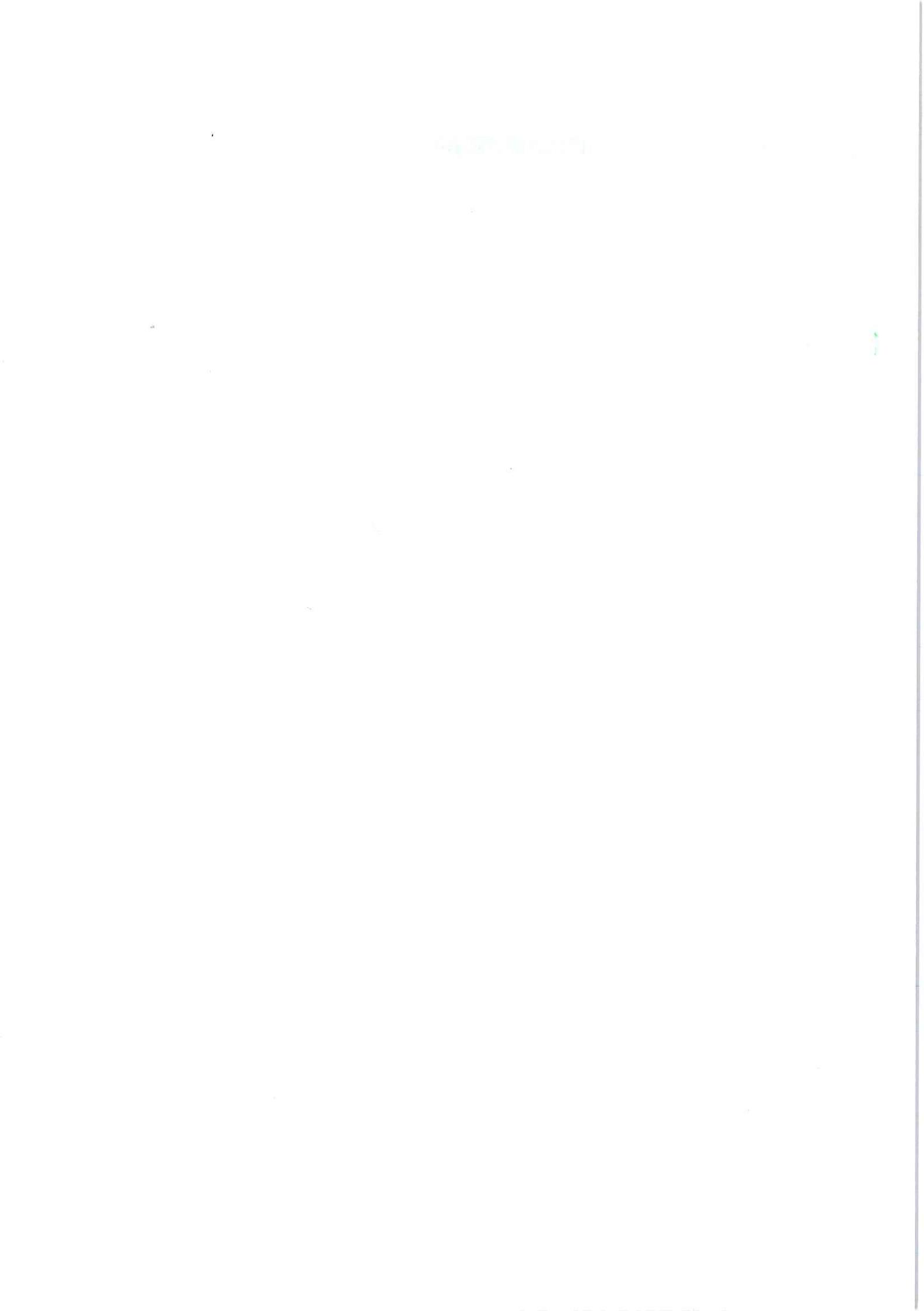
- i costruttori delle sottoclassi possono sollevare QUALSIASI ECCEZIONE
- tutte le eccezioni concionabili nel costruttore della classe madre devono essere esplicitate anche dal costruttore delle classi figlie, poiché i costruttori delle sottoclassi chiamano sempre (implicitamente o esplicitamente) il costruttore della super-classe!

### • MOTIVI:

\* Le sottoclassi hanno una PARTE DELLO STATO IN PIÙ rispetto alla classe madre, che il costruttore deve configurare!

In tale configurazione, potrebbero verificarsi delle eccezioni non previste dalla classe madre!

D'altra parte, è impossibile prevedere tutte le possibili eccezioni concionabili delle sottoclassi  $\Rightarrow$  i costruttori fanno eccezione e possono generare qualsiasi tipo di eccezione!



## METAMODELLAZIONE

I linguaggi di modellazione consentono di modellare i diversi aspetti di un sistema, sia secondo viste STATICHE che DINAMICHE.

- L'uso di modelli è tanto utile quanto più si riesce ad **AUTOMATIZZARE** il passaggio dal mondo delle idee/dei diagrammi al mondo della sintetizzazione effettiva del sistema.

Ma come si fa? Come si fa a far interagire in modo automatico i modelli con le loro controparte?

- **Come MANIPOLARE IN MODO AUTOMATICO MODELLI SOFTWARE?**

Prima di rispondere, dobbiamo capire com'è fatto un linguaggio di modellazione.  
È strutturato in 3 MACRO-PARTI:



- **SINTASSI ASTRATTA**: descrive la STRUTTURA del linguaggio e il modo in cui le primitive possono essere combinate tra loro.  
E' **INDIPENDENTE** da qualsiasi forma di rappresentazione/codifica.  
Per esempio: Classe = ( nome, attributi, operazioni ).
- **SINTASSI CONCRETA**: descrive una possibile NOTAZIONE (testuale o grafica) che può essere associata ad una sintassi astratta.  
E' una RAPPRESENTAZIONE SPECIFICA di un linguaggio.  
E' usata dai progettisti per creare modelli.
- **SEMANTICA**: definisce il SIGNIFICATO di ogni elemento del linguaggio. Può essere sia una definizione formale che semi-formale (in linguaggio naturale, spiega il significato degli oggetti).  
Una parola o scritto specifico della Semantica cause incisioni ed usi scorretti del linguaggio.

- **Come MANIPOLARE AUTOMATICAMENTE MODELLI SOFTWARE?**

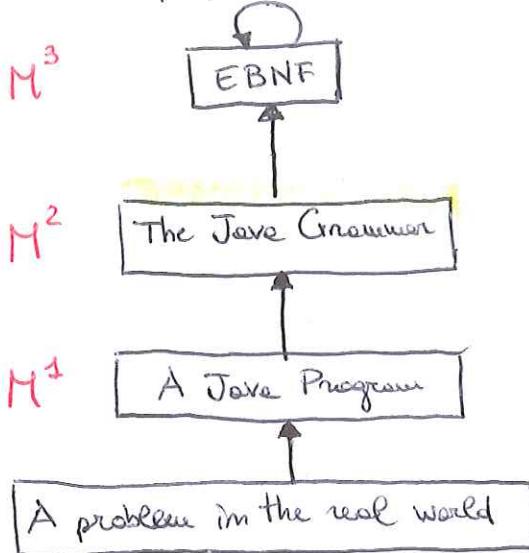
Abbiamo 2 tecniche differenti per farle:

- **META-MODELLAZIONE**

- Trasformazioni automatiche.

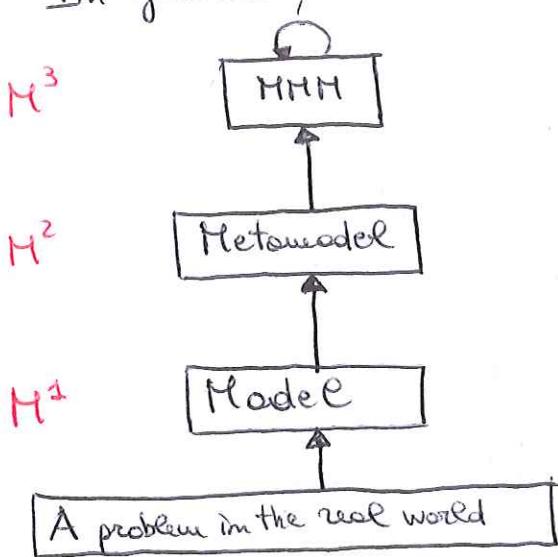
- Come i linguaggi naturali, i linguaggi di programmazione sono definiti attraverso **GRAMMATICHE FORMALI** (sintassi astratta), o loro volte espresse in termini di altri linguaggi formali.

Per esempio, con Java:



- EBNF = Extended Backus-Naur Form: è una grammatica per descrivere grammatiche, incluse sé stesse. È una grammatica libera del contesto.

- Le stesse cose avviene con i linguaggi di MODELLOAZIONE, come ad esempio UML.  
In generale, abbiamo il seguente schema:



- Model = un linguaggio di modellazione, ad esempio UML
- Metamodel = un METAMODELLO è una grammatica per definire modelli
- M-M-M (Meta-Metamodel) = un META-METAMODELLO è una grammatica per definire grammatiche per linguaggi di modellazione
- \* UML è definito mediante il metamodello di UML, che a sua volta è definito dal meta-metamodello MOF = Meta-Object Facility.

- Per poter estendere le funzionalità di un modello, devo modificare le sue grammatiche (il metamodello); ma per fare ciò, devo conoscere il META-METAMODELLO!

### DISEGNARE VS Modellare:

#### DISEGNARE:

- 1) Non riferiscono una grammatica
- 2) NON associano alcun SIGNIFICATO agli elementi modellati
- 3) Gestiscono solo le rappresentazioni grafiche, ma non ci controlla (sintattico) sulle relazioni tra gli elementi modellati.

#### MODELLARE:

- 1) Rappresentano gli elementi secondo una GRAMMATICA
- 2) Definiscono relazioni tra elementi solo se previsti dalla grammatica (controllo SINTATTICO)
- 3) Ogni elemento ha un SIGNIFICATO nel modello.

## REFLECTION

È il meccanismo che consente di percorrere le relazioni / associazioni "meta" e sfruttare il metamodello.

### REFLECTION

La Reflection è la capacità di un programma di eseguire elaborazioni che hanno per oggetto il programma stesso e, in particolare, la struttura del suo codice sorgente. È un meccanismo abilitato A TEMPO D'ESECUZIONE che consente di:

- Analizzare le strutture di una classe
- Richiedere la creazione dinamica di oggetti
- Richiedere dinamicamente l'invocazione di metodi
- Modificare dinamicamente la struttura del programma stesso.

- In Java, ogni istanza è un oggetto; ciò vuol dire che ogni classe è una sottoclasse di Object (is-a-kind-of)

Java offre un REFLECTION MODEL che definisce le meta-classi consultabili operando, tramite reflection, a livello del metamodello.

\* La meta-classe "Class" NON è figlia di Object → Una classe NON è "un-tipo-di" istanza!

- META-CLASSE "CLASS":

La classe "Class" è una METACLASSE, cioè una classe del METAMODELLO Java: rappresenta un metatipo di dato che descrive i tipi di dato che possono essere caricati all'interno delle JVM e indica come è definita una classe (specifica) all'interno delle grammatiche Java.

Un'ISTANZA di Class può rappresentare classi ed ~~o~~ interfacce in esecuzione in una applicazione Java. Può essere:

- una enumerazione (enum);
- un'annozione (@);
- un array (un'istante per coppia [tipi array, dimensione]);
- un tipo Java primitivo o lo keyword "void".

\* Class NON definisce un costruttore pubblico: un'istante di Class esiste se ha caricato almeno 1 Oggetto nella JVM

→ E' compito della JVM caricare in memoria le istanze (oggetti, Object) di Class!

- Altre metaclassi, figlie di Object, presenti nel reflection model di Java sono:  
Attribute, Field, Method, Constructor.

Class<?> c = Class.forName(className); // c è il tipo di dato caricato nella JVM che si chiama "className"  
(Dichiarazione del tipo di dato)

this.foo = (FooInterface) c.newInstance();

restituisce un Object

E' RESPONSABILITÀ DEL PROGRAMMATORE dire che oggetto sia!

- Method methodToBeCalled = ReflectionExample.class.getMethod(this.methodName, String.class);  
METACLASSE  
(dichiarazione di un metodo)  
istanze di java.lang.Class  
(è la dichiarazione del tipo di dato ReflectionExample)
- Object returnedObject = methodToBeCalled.invoke(reflectionObj, this.methodParam);  
dichiarazione del metodo da chiamare  
istanze e cui inviare MESSAGGIO, su cui chiamare il metodo
- String msgUsingReflection = (String) returnedObject;

### SVANTAGGI:

- Il programmatore deve esplicitamente assumersi le responsabilità di conoscere come sono implementate le classi su cui applica la reflection;
- Bisogna gestire le logiche d'errore; ad esempio: ClassNotFoundException, InstantiationException, IllegalAccessException, ...
- Non si può fare un CONTROLLO SINTATTICO in fase di compilazione!  
Eventuali errori saranno svelati solo a runtime! IR che rende molto difficile il debugging.

### MA QUANDO È UTILE LA REFLECTION?

È utile per "stressare" il concetto di DYNAMIC LOADING, che consente di caricare classi in memoria DINAMICAMENTE

### DYNAMIC LOADING:

All'esecuzione di un programma Java, la 1<sup>a</sup> classe che viene caricate in memoria è quella il cui metodo main è stato invocato; le altre classi sono caricate solo successivamente in memoria, in modo DINAMICO!

Ma come?

- Quando si cerca di istanziare una classe non presente in memoria, si entra in LOGICA DI ERRORE:
    - viene sollevata un'eccezione di tipo ClassNotFoundException;
    - è la JVM stessa a cogliere e gestire l'errore, chiedendo DINAMICAMENTE al ClassLoader di caricare la classe tramite REFLECTION:
- "`Class.forName(classFullNameString)`"

- Dynamic Java Class è un'importante funzionalità delle JVM perché consente di installare componenti software e runtime *on-the-fly*; ciò segue il principio del LAZY LOADING ("caricamento pigrò") : rinvià il caricamento in memoria delle classi il più tardi possibile, solo quando servono effettivamente.

(1) Il 1° uso della REFLECTION è INTERNO → usato dalla JVM per non consumare risorse quando non è necessario

↓

Caricamento dinamico delle CLASSI  
in memoria

(2) Uso di REFLECTION come PROGRAMMATORE:

nel caso in cui, per esempio, bisogna supportare diverse tecnologie, tra cui sceglierne una, anziché implementare un meccanismo polimorfo che comporterebbe il caricamento in memoria delle varie classi e/o librerie per tutte le diverse opzioni, si può usare la REFLECTION leggendo da un file di configurazione quello che si deve caricare.

• JUnit: utilizza la reflection quando viene chiamato su una classe, di cui, tramite REFLECTION, prende la dichiarazione (`Class.forName(...)`) e prende la dichiarazione dei metodi; per ogni metodo vede se è annotato con `@Test` (`method.isAnnotated("@Test")`) e, se sì, lo invoke!

ancora una volta DYNAMIC LOADING.

(3) Si può usare la REFLECTION in combinazione con pattern GoF CREAZIONALI quali Factory Method e Abstract Factory.



## JDBC

JDBC sta per Java DataBase Connectivity. È un framework per la connessione e l'interrogazione di DBMS relazionali.

- \* È **INDIPENDENTE** del particolare DBMS utilizzato, dunque consente di progettare applicazioni Java che siano agnostiche rispetto alle scelte di un DBMS.
- Tuttavia, resta **RESPONSABILITÀ DEL PROGRAMMATORE** la realizzazione di una completa estrazione rispetto al DBMS, in quanto deve cercare nell'interazione con JDBC di utilizzare solo sintassi SQL Standard!
- L'architettura di JDBC prevede l'utilizzo di un "DRIVER MANAGER", che espone alle applicazioni un insieme di interfacce standard e si occupa di caricare e runtime i DRIVER opportuni per pilotare uno specifico DBMS.  
Dunque, fa il binding su implementazioni alternative sulle stesse API comune.
- JDBC implementa concetti tipici delle progettazione O.O. e offre un'estensione delle funzionalità offerte attraverso una API comune: **java.sql.\***

### STRUTTURA TIPICA DI INTERAZIONE CON UN DBMS con JDBC:

1. Load a driver for a specific DBMS → // per driver di tipo 4, quest'operazione è fatta esclusivamente dall'applicazione
2. Connect to a DB
3. for each query to the DB {
  4. instantiate an interaction Statement
  5. query the DB
  6. do something with the results
7. }
8. Close the connection with the DBMS

- \* E' buone pratica minimizzare il numero di connessioni aperte con il DBMS; se possibile aprire 1 sola connessione per applicazione e chiuderla quando non c'è più bisogno di interrogare col DBMS.

### java.sql.Connection :

È una classe che rappresenta una SESSIONE di comunicazione con il DBMS. È necessaria aprire una per poter leggere e scrivere sul DB. L'apertura della connessione va richiesta al DriverManager, specificando:

{ URL del DBMS  
Nome del DB  
Username e Password dell'utente che ha i permessi per accedere al DB (utente sul DBMS)



**DriverManager.getConnection(**  
"db\_address",  
"username",  
"password");

## • java.sql.Statement :

È una classe che estrae il concetto di **OPERAZIONE** sulla base di dati. Deve essere usato per incapsulare il concetto di query SQL.

Vengono istanziati a partire da un'istante di connessione attiva sul DBMS:

connection.createStatement();

## • java.sql.ResultSet :

È una classe che modella i dati in forma tabellare ( JDBC è orientato a DBMS relazionali ), così come estratti dal DB.

- ResultSet offre l'operazione next() , che scorre i record estratti dal DB
- per ogni record , si puo' accedere ai singoli campi del DB in 2 modi :
  - 1) per INDICE DI COLONNA , con la numerazione che parte da 1;
  - 2) per NOME DELLA COLONNA , così come definito nel DB.
- i valori dei campi possono essere già prelevati convertiti nel tipo di dato di interesse , ad esempio con i metodi :
 `getString()`, `getDouble()`, `getInt()`, `getBoolean()`, ...

\* ATTENZIONE : C'è un **COUPLING MOLTO ALTO** tra chi deve effettuare le query e la struttura del DB ; deve sapere com'è fatto il DB.  
Ma questo Coupling è necessario .

Inoltre , s'arriva al seguente **PROBLEMA** : trasformazione logica dei dati dal contesto Object-Oriented al contesto del DB relazionale ( e viceversa ).

- Nel mondo Object Oriented : le istanze sono rese note tra loro mediante i riferimenti in memoria
- Nel mondo dei DB relazionali : le entità sono rese note tra loro mediante l'utilizzo delle stesse **CHIAVE** nelle corrispondenti relazioni ( TABELLE ).

Chi ha le responsabilità di conoscere com'è fatto il DB ?

E chi fa il mapping tra le istanze di entità Object Oriented con entità di un DB relazionale ?



Si introduce una nuova "categoria" di classi,  
le **DAO** !

# DAO

DAO = Data Access Object

Sono classi particolari introdotte per INGEGNERIZZARE il sistema con il seguente scopo.

## RESPONSABILITÀ:

Hanno il compito di mediare tra le rappresentazione Object Oriented in memoria delle istanze e la rappresentazione sul layer di persistenze, ad esempio DBMS relazionali.

- Ad ogni classe Entity che si vuole mandare in persistenza dovrebbe corrispondere un DAO che ne gestisce il salvataggio ed il recupero delle istanze sul layer di persistenze.
- Sono delle "Factory speciali"! Non hanno le responsabilità di fare le new(), bensì di recuperare dati dalla persistenza
  - Una FACTORY potrebbe erelarsi di un DAO; oppure un Controller potrebbe interrogare direttamente con un DAO, bypassando la Factory.
- \* Per minimizzare il numero di connessioni aperte con il DBMS, si puo' introdurre una classe Connector che realizzi il pattern GoF SINGLETON, avendo un attributo privato static Connection comm e un metodo per restituire le istanze di connessione comm.  
Così facendo, le DAO possono richiedere la SINGOLA ISTANZA di Connessione su cui fare le query.

100