ISPW – CONCETTI CHIAVE DELLA PARTE JAVA

# Lezione5 – Java: linguaggio e piattaforma

**Il modificatore final**:  il modificatore final può essere utilizzato, per definire una variabile che di fatto diverrà una **costante** per l'istanza della classe. Final può essere applicato anche ai metodi di una determinata classe e un metodo definito come final (finale non modificabile) implica che se eredito la classe che contiene il metodo su questo non potrà essere eseguito l'override. Anche per quanto riguarda una classe questa può essere definita come final e di conseguenza una classe definita in questo modo non potrà essere ereditata. Ciò è logicamente comprensibile in quanto se dichiaro una classe con il modificatore final intendo una classe finale ovvero una classe completa che non necessita di specializzazioni o estensioni e dunque è più che logico che non sia possibile ereditarla.

**Il modificatore static**: il modificatore static può essere applicato a metodi e a variabili di una classe. L'effetto di dichiarare un metodo static è quello di rendere il metodo di fatto comune a tutte le istanze della classe. Dunque se dichiariamo un metodo static potremo evitare di istanziare la classe che lo contiene e richiamare il metodo statico con la sintassi: **nomeClasse.nomeMetodo()**. Quando si usa il modificatore static per una variabile, di fatto si rende l'attributo della classe comune a tutte le istanze della classe. Dunque all'atto pratico se dichiariamo una variabile di istanza utilizzando il costrutto **public static** avremo come risultato il definire una **variabile globale** accessibile a tutte le istanze di una classe. Le variabili globali sono un arma a doppio taglio in quanto se il fatto di poter condividere una variabile con lo stesso valore per tutte le istanze della classe è sicuramente un vantaggio, si ha che nel caso in cui, per errore, un'istanza modifichi la variabile globale, tutte le altre istanze della classe troveranno il valore modificato compromettendo la robustezza dell'intera applicazione.

Nell’ereditarietà assumono importanza i modificatori Public, Private e Protected:

**Il modificatore di accesso public**: permette di dichiarare dati e metodi visibili e quindi utilizzabili dalla sottoclasse senza limitazioni.

**Il modificatore di accesso private**: nasconde completamente dati e metodi dichiarati tali.

**Il modificatore di accesso protected**: permette l’accesso a metodi e dati a tutte le sottoclassi, all’interno o meno dello stesso package, e alle sole classi dello stesso package se si tratta di una variabile reference\*.

\*Tutte le istanze di classi create in Java sono definite mediante l’utilizzo di *reference*. Un reference è, dunque, una variabile speciale che tiene traccia di istanze di tipi non primitivi. I reference possono tenere traccia soltanto di oggetti di tipo compatibile: ovvero un reference ad un oggetto di tipo MyClass non potrà tenere traccia di oggetti di diverso tipo.

\* class package subclass world  
\* public si si si si  
\* protected si si si no  
\* default si si no no  
\* private si no no no  
\*

# Lezione6 – I linguaggi di modellazione && UML

**UML** (Unified Modeling Language) è un linguaggio visuale per la specifica, la costruzione e la documentazione degli elaborati di un sistema software.

Si tratta di un vero e proprio **linguaggio di modellazione**, che riunisce costrutti e concetti di approcci già esistenti, come la programmazione object-oriented, le macchine a stati e la *objectory vision* (visione oggettiva). Inoltre: - Prevede meccanismi di estensibilità (è possibile aggiungervi nuove librerie). - Astrae dai linguaggi di programmazione (è un linguaggio più ad alto livello rispetto a qualsiasi linguaggio di programmazione). - Rappresenta i concetti (molto spesso tramite una notazione visuale).

UML mette a disposizioni tre tipi di diagrammi principali per descrivere:

struttura statica del sistema: **Structure diagrams** (e.g. class diagrams, object diagrams, component diagrams, package diagrams)

comportamento del sistema: **Behavior diagrams** (e.g. state diagrams, activity diagrams, use case diagrams)

interazione tra gli oggetti: **Interaction diagrams1** (e.g. sequence diagrams, communication diagrams, timing diagrams) [sono un sottotipo dei behavior diagrams]

Un sistema software espone molteplici aspetti che devono essere trattati appositamente:

- **Aspetti funzionali** (legati ai requisiti utente, ai requisiti implementativi e alle leggi)

- **Aspetti extra-funzionali** (legati agli aspetti di qualità del servizio offerto, che sono relativi alla soddisfazione del cliente, e ai vincoli sul tempo e sulle risorse a disposizione)

- **Aspetti organizzativi** (legati agli aspetti tecnologici e non)

UML consente di scomporre un processo di sviluppo software secondo **cinque prospetti** (o viste), ciascuno dei quali enfatizza la descrizione di aspetti specifici del sistema in diverse fasi dello sviluppo.

1) **Use case view**: si occupa dell’analisi / modellazione dei requisiti utente e di come il sistema viene percepito dall’utente (progettazione black-box); descrive quali funzionalità devono essere progettate (e non come). Nella pratica, consiste nell’individuare tutti gli attori, i casi d’uso e le loro relazioni. Target: clienti, progettisti, sviluppatori, tester.

2) **Logical view**: si occupa della progettazione della struttura del sistema (progettazione white-box); descrive come devono essere progettate e realizzate le funzionalità del sistema. Target: progettisti, sviluppatori.

3) **Implementation view**: si occupa di strutturare i moduli implementativi organizzando il codice del sistema in moduli. Target: progettisti, sviluppatori, tester.

4) **Process view**: comprende modelli che descrivono la dinamica del sistema, specificando i processi da eseguire e le entità che eseguono tali processi; si usa per un utilizzo efficace delle risorse, per stabilire l’esecuzione parallela degli oggetti e per gestire eventi asincroni (esterni al sistema). Target: sviluppatori, integratori di sistema.

5) **Deployment view**: si occupa di come e dove devono avvenire le installazioni dei sistemi software. Comprende modelli che descrivono la topologia e l’organizzazione delle macchine fisiche (e.g. computer, dispositivi mobili, connessioni fisiche tra i nodi) e modelli che descrivono come le parti del sistema software sono mappate sull’architettura fisica. Target: sviluppatori, integratori di sistema, tester.

# Lezione9 – Classi, oggetti e information hiding

**Oggetto**: colloquialmente il termine viene spesso utilizzato (in modo ambivalente) per indicare una classe o una sua istanza. Tuttavia, sarebbe più opportuno usarlo solo per queste ultime.

**Classe**: modella una famiglia di entità del dominio di applicazione o un elemento che non fa parte del nel dominio ma che è introdotto durante il processo di sviluppo. In entrambi i casi include la definizione delle proprietà(attributi) e del comportamento(operazioni). Una classe raggruppa un insieme coeso di entità (insieme coeso: tutti gli elementi modellano aspetti utili al raggiungimento dello stesso fine/funzionalità). Viceversa, la non-coesione è sinonimo di accoppiamento e consiste nel raccogliere all’interno della medesima classe elementi o funzionalità semanticamente molto distanti tra loro (da evitare nella progettazione / programmazione object-oriented). Un oggetto si relaziona ad una classe allo stesso modo di come un dato si relaziona ad un tipo (nei ling. di progr.). Tutti gli oggetti(istanze) afferenti una classe condividono lo stesso insieme di comportamenti, proprietà e relazioni; in generale differiscono nei valori delle proprietà.

**Istanza**: A una classe, che cattura in modo astratto un concetto, possono afferire delle **istanze** (=oggetti), che rappresentano gli elementi effettivamente appartenenti al mondo reale e, quindi, sono manifestazioni concrete di un’astrazione (ad esempio, la penna di Marco può essere vista come un’istanza della classe “penna”).

**Dinamica di un sistema**

- **Approccio procedurale** (tipico del C): esiste un *main* che coordina il comportamento del sistema in base allo scheletro del diagramma di flusso codificato nelle istruzioni. Il *main* gestisce direttamente i dati del sistema, occupandosi esplicitamente della loro transizione tra le varie funzioni (o procedure o subroutine, a seconda dello specifico linguaggio di programmazione).

- **Approccio object oriented** (tipico del Java): ogni oggetto ha i suoi “dati” (ovvero i valori per gli attributi di istanza). Gli attributi di istanza sono allocati e inizializzati alla creazione dell’oggetto e vengono mantenuti dall’istanza anche dopo l’esecuzione dei metodi (a differenza di quanto accade per le tradizionali invocazioni a funzione nell’approccio procedurale): tutti i dati sono mantenuti fino alla distruzione (esplicita o implicita) dell’oggetto. Qui il *main* non fa più da “direttore d’orchestra”: il comportamento del sistema è generato dall’interazione di più istanze tra loro. In particolare, l’interazione comporta lo scambio di **messaggi** tra oggetti diversi e può produrre transizioni di stato nelle entità coinvolte. A valle di tutto questo, con l’approccio object oriented è possibile costruire un programma con tante classi *main* e si può decidere come far funzionare il programma in base alla classe da cui deve partire l’esecuzione; è inoltre possibile avere classi senza alcun *main*.

**Operazione vs metodo vs messaggio**

- **Operazione**: è una funzionalità che può essere eseguita su oggetti di una determinata classe. Ha semplicemente una segnatura (=firma), che specifica il nome dell’operazione e il tipo degli eventuali parametri passati in input. Può specificare anche il tipo di ritorno. In altre parole, è la specifica di un “*segnatura*” (prototipo) che un oggetto mette a disposizione di altri oggetti; essa rappresenta il tipo di funzionalità che una classe può usare nei confronti di un’altra classe.

- **Metodo**: è l’implementazione vera e propria di un’operazione messa a disposizione da una classe e, quindi, è un sottoprogramma associato in modo esclusivo a tale classe. Oltre alla segnatura e al tipo di ritorno, è caratterizzato anche da un corpo, che contiene una o più sequenze (o blocchi) di istruzioni scritte per eseguire una determinata azione sulla base dei parametri passati in input. Inoltre, è in grado di restituire al chiamante un valore di ritorno (=output) dello stesso tipo di quello dichiarato inizialmente insieme al nome e ai parametri. Nei linguaggi che dispongono di un meccanismo di gestione delle eccezioni (come Java), il blocco del metodo può terminare sollevando un’eccezione nel caso si verifichi una situazione anomala che impedisce il corretto completamento delle istruzioni. Il concetto di metodo, così come quello di operazione, è “statico”, cioè è definito nel momento in cui si sta programmando (a design-time) ma non è detto che venga attivato a run-time.

- **Messaggio**: è la richiesta a run-time dell’invocazione di un metodo fornito da un’istanza B da parte di un’istanza A (in tal caso si dice che “A invia un messaggio a B”). In quanto tale, potrebbe fallire.

**Class diagram**

È uno dei tipi di diagrammi che possono comparire in un modello UML e ritrae la struttura statica del sistema (infatti comprende elementi definiti a design-time). Ha una rappresentazione logica a grafo, in cui i nodi raffigurano **classi** e **interfacce**, mentre gli archi raffigurano **relazioni**. Una classe all’interno del class diagram è caratterizzata da un nome, da degli attributi e dalle operazioni che possono essere eseguite sugli attributi. È in pratica lo stesso concetto che in O.O. (object orientation): semplificando, rappresenta un tipo di dato non primitivo. Talvolta, può essere usata per raggruppare elementi e contenere package o sottosistemi. D’altra parte, le relazioni corrispondono alla definizione di possibili interazioni tra le classi di un modello e, in particolare, di legami (link) che possono sussistere tra gli oggetti di tali classi. Possono essere corredate da un insieme di informazioni aggiuntive, come il ruolo svolto da ogni classe o la molteplicità (che indica il numero di oggetti delle due classi che possono essere coinvolti in un collegamento). In particolare, una relazione tra una classe A e una classe B implica che A è a conoscenza di B e, in qualche modo, può interagirvi, al fine di fornire, in modo cooperativo, un comportamento complesso. Perciò, l’assenza di relazioni in una determinata classe comporta l’isolamento di tale classe, la quale si ritrova impossibilitata a interagire con le altre classi. Esistono più tipi diversi di relazione, ciascuno dei quali definisce il modo di interazione tra le due classi coinvolte. Questi tipi di relazione sono:

- Aggregazione

- Associazione

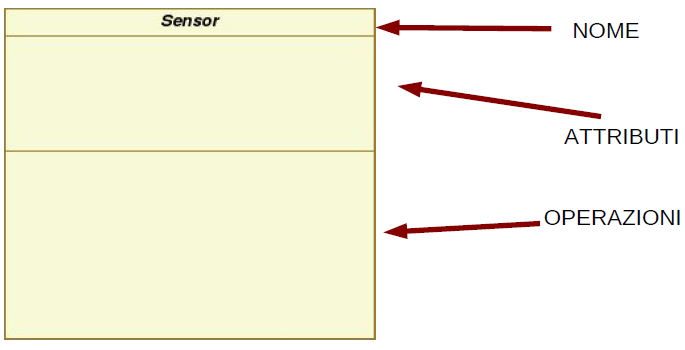
- Composizione

- Dipendenza

- Generalizzazione

- Realizzazione

In UML, una classe è strutturata da tre comparti: **nome**, **attributi** e **operazioni**.



**Comparto nome:** Definisce il nome di un’entità e consiste in una stringa di testo che, per convenzione, ha la lettera iniziale maiuscola. Definizione completa:

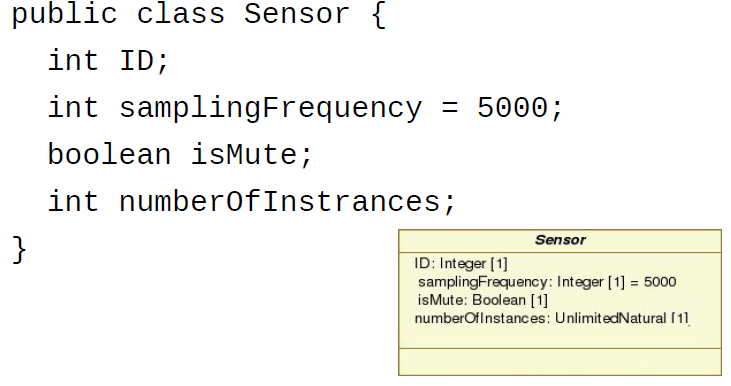
- Java: package + “.” + nome\_classe

- UML: prefisso + “::” + nome\_classe

(notare che se il nome della classe in UML è in *corsivo* allora in Java è richiesto il modificatore “abstract” e viceversa)

**Comparto attributi:** Modella le proprietà di una classe: ogni attributo descrive un insieme di valori che la proprietà può avere quando vengono istanziati oggetti di quella determinata classe. Infatti, le proprietà sono condivise tra tutti gli oggetti appartenenti a una particolare classe. In generale, sono i valori a non essere condivisi tra le istanze.

NB: Tra i tipi di dato degli attributi in Java e i tipi di dato degli attributi in UML non c’è sempre una corrispondenza univoca o esatta. Per esempio, in UML è possibile definire un attributo di tipo *UnlimitedNatural* che, tuttavia, non esiste in Java e, quindi, in fase di programmazione, può essere tradotto in più modi possibili, anche in base alle scelte dell’analista o del progettista: una possibilità è considerare l’*UnlimitedNatural* come un semplice *integer* e imporre al relativo attributo la condizione per cui deve obbligatoriamente assumere valori non negativi; un’altra possibilità è creare una nuova classe Java *UnlimitedNatural* in grado di rappresentare tutti e soli i numeri naturali. In ogni caso, è buona norma diminuire il più possibile il gap che si potrebbe creare tra la fase di progettazione e la fase di programmazione.



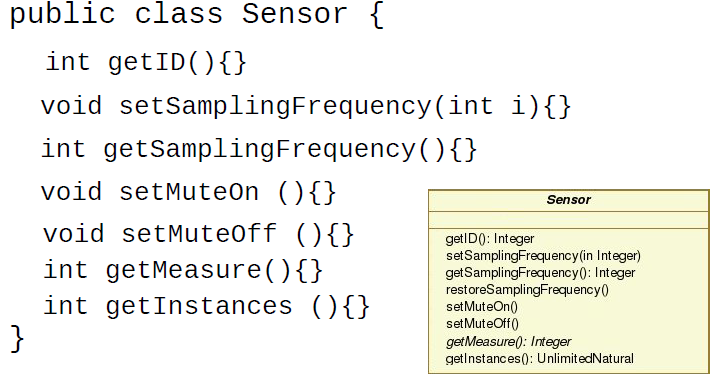
**Comparto operazioni:** specifica i servizi che la classe offre, ovvero che cosa essa può fare, non come. Infatti, in UML non vengono mai specificati i metodi delle varie operazioni. Notiamo che la totale assenza di relazioni per una determinata classe è possibile causa della mancanza di operazioni che agiscono sullo stato di tale classe. Le operazioni manipolano lo stato degli oggetti, ovvero il valore degli attributi di una particolare classe. Hanno una segnatura, che consiste di:

- Un **tipo** (che in realtà appartiene alla segnatura solo in UML e in alcuni linguaggi di programmazione, ma non in Java)

- Un **nome**

- Una **lista di parametri**

NB: Come già accennato, in Java il tipo di ritorno non appartiene alla segnatura di un’operazione o di un metodo: benché sia sempre da specificare, non è in grado da solo di distinguere un’operazione dalle altre.



**Costruttore**: è un’operazione speciale che serve a creare nuove istanze delle classi. La chiamata è effettuata automaticamente all’atto della creazione di un nuovo oggetto di una classe e, nella maggior parte dei linguaggi, non è possibile effettuare un’invocazione manualmente in un secondo tempo. Il costruttore ha un ambito di classe e non di istanza e, infatti, pre-esiste agli oggetti. È utile anche per inizializzare lo stato delle nuove istanze e definire un contesto di esecuzione.

Generalmente, una classe può avere più costruttori, i quali rappresentano più modi differenti di creare le istanze e devono avere tutti lo stesso nome, ovvero quello della classe in cui sono definiti; perciò, si distinguono l’uno dall’altro esclusivamente per il numero e l’ordine dei loro parametri. Per giunta, non vogliono che sia indicato esplicitamente un tipo di ritorno, sia perché è implicitamente dato dal nome dei costruttori stessi, sia perché il tipo di ritorno è di default l’oggetto stesso. Tra i possibili tipi di costruttore ricordiamo quello semplice (che ha uno o più parametri qualsiasi), il **default constructor** (che non prevede alcun parametro) e il **copy constructor** (che riceve come parametro un’altra istanza della medesima classe e ne copia lo stato sulla nuova istanza creata). In genere il costruttore rappresenta un comportamento puramente implementativo, per cui non viene mai esplicitato nei class diagram e viene considerato direttamente in fase di programmazione.

**Distruttore**: ha il compito di deallocare lo spazio occupato da una specifica istanza. In alcuni linguaggi di programmazione object-oriented (come Java) non è previsto che il distruttore venga invocato dal programmatore: questi linguaggi vengono detti *garbage collected* perché mettono a disposizione il cosiddetto **garbage collector**, che è un thread a bassa priorità della Java Virtual Machine (JVM) e lavora nell’heap per verificare se esistono istanze isolate; se sì, dealloca queste istanze tramite una chiamata implicita del distruttore della classe di interesse. In effetti, le istanze isolate sono inutili e sprecano solo memoria, dato che non sono in alcun modo raggiungibili o referenziabili poiché Java e, più in generale, i linguaggi *garbage collected* non prevedono l’algebra dei puntatori. Neanche il distruttore viene specificato all’interno dei class diagram e nelle fasi di analisi e progettazione in generale.

**L’istanza speciale *this***: indica il riferimento all’istanza corrente ed è implicitamente definita come attributo (*NomeClasse this*; ). È per lo più utilizzata nei seguenti ambiti:

- All’interno di un costruttore per invocarne un altro della medesima classe (e.g. *this(“ciao”)* invoca il costruttore di sé stesso che accetta una stringa come unico parametro).

- All’interno di metodi e/o costruttori per disambiguare i riferimenti agli attributi e ai metodi della specifica istanza.

**Attributi e operazioni di classe**:

- **Attributo di classe**: è valido indipendentemente dall’esistenza di istanze della classe. Il suo valore è condiviso tra tutte le eventuali istanze, le quali contengono un campo corrispondente a un puntatore all’unica area di memoria (nell’heap) in cui si trova l’attributo di classe.

- **Operazione di classe**: non richiede l’esistenza o l’impiego di un’istanza della classe per poter essere chiamata: l’invocazione può infatti avvenire direttamente mediante il nome della classe stessa (NomeClasse.nomeMetodo()).

Sia gli attributi che le operazioni di classe si indicano:

- Col modificatore ***static*** in Java

- Con la sottolineatura in UML

**Incapsulamento e information hiding**: nei linguaggi di programmazione object-oriented, il termine “incapsulamento” può essere usato per riferirsi a due concetti o alla combinazione dei due:

- Un meccanismo del linguaggio di programmazione atto a limitare l’accesso diretto agli elementi dell’oggetto

- Un costrutto del linguaggio di programmazione che favorisce l’integrazione dei metodi all’interno della classe

I termini “incapsulamento” e “information hiding” vengono spesso usati come sinonimi, anche se tra loro esiste una sottile differenza concettuale: l’information hiding è il principio teorico su cui si basa la tecnica dell’incapsulamento. Secondo il concetto di information hiding, i dettagli implementativi di una classe sono nascosti all’utente. Pertanto, una parte di programma può nascondere informazioni incapsulandole in un costrutto dotato di interfaccia, permettendo appunto l’information hiding. Tuttavia, l’incapsulamento non è garanzia di information hiding, poiché potrebbe, se mal utilizzato o per motivi particolari, non nascondere i dettagli implementativi. L’incapsulamento riduce il costo da pagare per correggere gli errori in fase di sviluppo di un programma. Questo risultato viene ottenuto strutturando l’intero progetto, e i moduli che lo compongono, in modo che un’errata decisione presa nell’implementazione di un singolo modulo non si ripercuota sull’intero progetto e possa essere corretta modificando soltanto quel modulo. Si potrà così evitare di dover modificare anche i moduli *client*, che interagiranno con quello incapsulato soltanto attraverso interfacce.

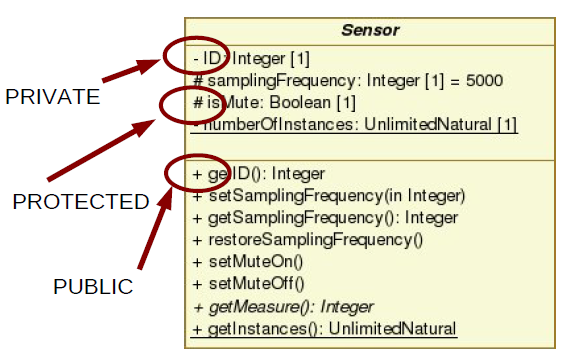
Un altro possibile motivo per ricorrere all’incapsulamento è la necessità di applicare dei controlli sull’accesso e/o sulla manipolazione delle proprietà delle istanze; ad esempio, potrebbe essere opportuno che un attributo di tipo intero di una data classe assuma in realà solo valori naturali. Per avere un controllo sui valori assunti da tale attributo, bisogna mantenere quest’ultimo “nascosto” ai *client*. Per fare ciò (o anche per nascondere le scelte che possono essere soggette a cambiamenti), occorre introdurre il concetto di **visibilità** degli attributi e delle operazioni, che indica quali classi hanno la possibilità di accedere a tali attributi / operazioni e può essere:

- **Pubblica**, se l’attributo / operazione può essere acceduto/a da qualunque classe raggiungibile. In Java si indica col modificatore *public* anteposto all’attributo / operazione. In UML si indica col simbolo “+” anteposto all’attributo / operazione.

- **Privata**, se l’attributo / operazione può essere acceduto/a esclusivamente dalla classe nella quale è definito/a. In Java si indica col modificatore *private* anteposto all’attributo / operazione. In UML si indica col simbolo “-“ anteposto all’attributo / operazione.

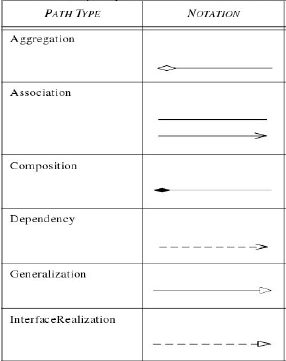
- **Protetta**, se l’attributo / operazione può essere acceduto/a esclusivamente dalla classe nella quale è definito/a e dalle eventuali classe figlie (questo aspetto sara maggiormente chiaro più avanti). In Java si indica col modificatore *protected* anteposto all’attributo / operazione. In UML si indica col simbolo “#” anteposto all’attributo / operazione.

La regola pratica che consente di applicare correttamente la tecnica dell’incapsulamento consiste nel porre gli attributi sempre privati o protetti, mentre le operazioni possono essere pubbliche; i *client* dovrebbero poter accedere agli attributi solo se strettamente necessario ed esclusivamente per mezzo di operazioni *get* / *set* (queste ultime in particolare consentono di introdurre qualunque controllo sul nuovo valore da assegnare al relativo attributo).



# Lezione10 – Ereditarietà

Ci sono diverse relazioni per i class diagrams:



L’ereditarietà rappresenta una delle relazioni che possono essere stabilite tra due classi: la **generalizzazione**. Se la classe B eredita dalla classe A, si dice che B è una **sottoclasse** di A (o classe figlia o classe derivata) e che A è una **super-classe** di B (o classe parent o classe base).



La generalizzazione:

- In Java viene definita tramite l’espressione “**extends** NomeClasseParent” all’interno della definizione della classe figlia.

- In UML è raffigurata tramite una freccia con un triangolo bianco rivolto verso la super-classe.

Nel contesto in cui tra due classi si ha una relazione di generalizzazione, la super-classe definisce un concetto generale, mentre la sottoclasse rappresenta una variante specifica di tale concetto generale. In particolare, la sottoclasse:

- Eredita (ha implicitamente) tutte le variabili di istanza e tutti i metodi della super-classe.

- Può avere variabili o metodi aggiuntivi, per cui tipicamente contiene più informazione della super-classe.

- Può ridefinire i metodi ereditati dalla super-classe.

La generalizzazione è una relazione uniforme per tutti gli oggetti, senza eccezioni: non è possibile che alcune istanze ereditino proprietà da una super-classe mentre altre istanze della stessa classe non ereditino dalla super-classe. Una sottoclasse eredita tutte le caratteristiche della superclasse una ed una sola volta.

**Principio di sostituibilità di Liskov**

Date due classi legate tra loro mediante una relazione di generalizzazione, si dice che la classe figlia **is-a-kind-of** (is-a) la classe parent.

La relazione is-a-kind-of viene spesso esplicitata facendo riferimento al cosiddetto **principio di sostituibilità di Liskov**: “Se q(x) è una proprietà che si può dimostrare essere valida per oggetti x di tipo T, allora q(y) deve essere valida per oggetti y di tipo S, dove S è un sottotipo di T”.

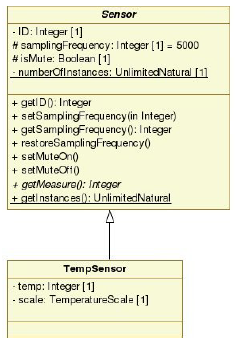
In altri termini: “Sia T una classe e sia S una sua sottoclasse; in tutti i contesti in cui si usa un’istanza di T deve essere possibile utilizzare una qualsiasi istanza di S (o di una qualunque altra sottoclasse a qualsiasi livello)”.

In poche parole, la sottoclasse deve avere la stessa semantica della super-classe. Tuttavia, è tecnicamente possibile estendere una classe violando il principio di sostituibilità di Liskov, in quanto le regole imposte dai linguaggi di programmazione non possono andare oltre la correttezza formale del codice scritto.

Esempi di violazione del principio:

- Ridefinizione di un’operazione della sottoclasse che ne alteri la semantica

- Uso di strumenti per l’occultamento di visibilità dei metodi (*limitation*)



**Generalizzazione vs specializzazione**

La generalizzazione può essere anche vista come una tecnica di modellazione che consiste nel ricavare una classe parent più “generale” a partire da una o più classi figlie con determinate caratteristiche in comune. La specializzazione, al contrario, consiste nel rappresentare inizialmente un concetto in modo generale tramite la definizione di una super-classe, per poi ricavare le classi figlie più “specializzate”.

Nella modellazione O.O. si tendono a utilizzare entrambe le tecniche. Tuttavia, nel processo di design di un sistema software, l’esperienza pratica in genere consiglia di individuare e modellare i concetti generici il prima possibile, poiché esprimono meglio *cosa* bisogna modellare piuttosto che *come*.

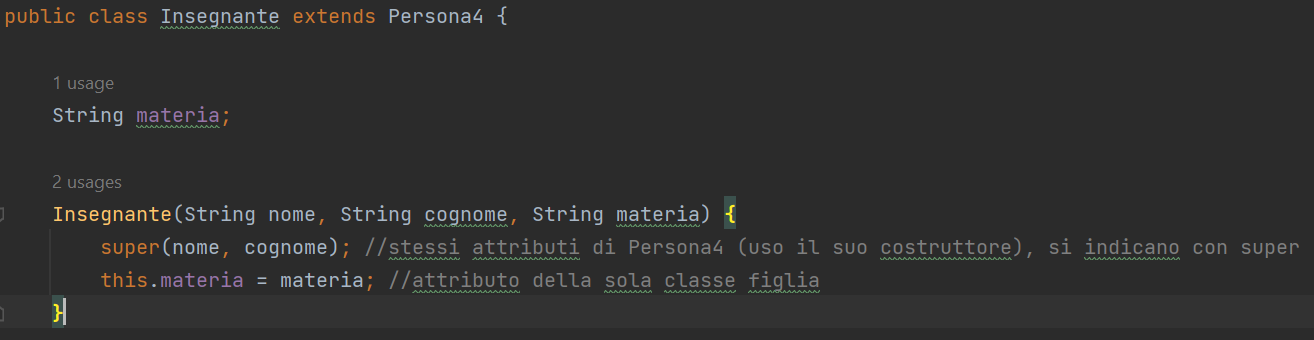
Esempio:

Sia Figura la classe parent e siano Rettangolo, Cerchio e Triangolo le sue classi figlie. È evidente come all’interno della classe Figura l’operazione calcolaPerimetro() sia solo dichiarata, mentre le relative implementazioni possono essere specificate soltanto nelle tre sottoclassi.



**Generalizzazione e costruttori**

Durante il processo di creazione di un oggetto di una classe derivata, il costruttore della classe base viene sempre chiamato come prima operazione. Se la classe figlia non chiama esplicitamente un costruttore della classe parent tramite il comando super(), viene implicitamente invocato il costruttore di default della classe base. Se però la classe figlia non chiama esplicitamente un costruttore della classe parent e quest’ultima non dispone del costruttore di default, allora viene sollevato un errore di compilazione.



**Ereditarietà singola vs ereditarietà multipla**

A seconda del linguaggio, l’ereditarietà può essere singola / semplice (per cui ogni classe può avere al più una super-classe diretta, come in Java) oppure multipla (per cui ogni classe può avere più superclassi dirette, come in UML e in C++). Poiché l’ereditarietà è una relazione transitiva, il suo utilizzo dà luogo a un ordinamento e a una gerarchia di classi: in particolare, nel caso di ereditarietà singola, la gerarchia avrà una struttura ad albero, che diventerà una struttura a foresta se si ha più di una super-classe “radice”; d’altra parte, nel caso di ereditarietà multipla, la gerarchia avrà una struttura a grafo aciclico diretto.

Vantaggi dell’ereditarietà multipla:

- Fornisce la possibilità di comporre velocemente oggetti anche molto complessi, aggregando molteplici funzionalità diverse all’interno di un’unica classe.

- È una soluzione elegante e utile in molti casi pratici, poiché porta a una semplificazione della sintassi e a una migliore rappresentazione della realtà.

Svantaggi dell’ereditarietà multipla:

- Porta a una complicazione notevole del linguaggio che la implementa.

- Gestire un linguaggio con ereditarietà multipla può risultare complesso e poco chiaro; una possibile causa di ambiguità è la seguente: se due classi B, C ereditano dalla classe A, la classe D eredita sia da B che da C, e un metodo in D chiama un’operazione definita in A, da quale classe viene ereditata questa operazione? Tale ambiguità prende il nome di **problema del diamante**.

- Porta a un rischio elevato di **name clash**: infatti, se si ereditano metodi con la stessa segnatura (ma con diverse implementazioni) da più di un genitore, avviene un conflitto. Per gestire tale situazione, si potrebbe applicare (ove possibile) dei criteri euristici per “linearizzare” la gerarchia (i.e. trovare un ordinamento dei nodi “fratelli” nella gerarchia), oppure forzare la risoluzione dei conflitti caso per caso.

# Lezione13 – Polimorfismo

Supponiamo di voler modellare una famiglia di sensori e, in particolare, sensori di temperatura e sensori di luminosità. Ormai sappiamo che una soluzione elegante a questo problema consiste nel definire una classe base (Sensor), in cui verranno definiti gli attributi e le operazioni in comune, e due classi derivate (TempSensor, LightSensor), in cui verranno aggiunti gli attributi e i metodi specifici per quelle sottoclassi. Sorge adesso un nuovo problema: alcune delle caratteristiche comuni che vengono raggruppate nella super-classe devono manifestarsi in modo diverso dipendentemente dalla sottoclasse considerata. Ad esempio, all’interno della classe Sensor potremmo aver definito l’operazione getMeasure(), che serve a effettuare una misurazione col sensore; tuttavia, il modo in cui avviene il rilevamento di una temperatura è presumibilmente diverso dal modo in cui avviene il rilevamento di un’intensità luminosa. Per questo motivo, è necessario che l’implementazione di getMeasure() all’interno della classe TempSensor sia differente dall’implementazione di getMeasure() all’interno della classe LightSensor, nonostante la semantica dell’operazione in questione rimanga la stessa (ovvero quella di effettuare una misurazione).

**Classe astratta**

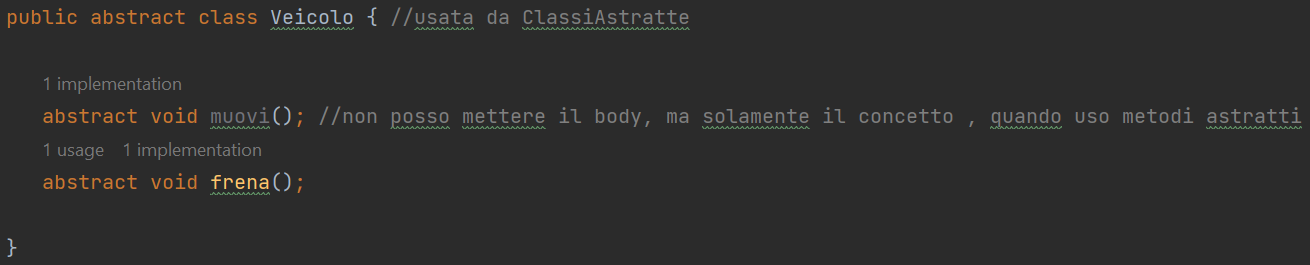
Una soluzione elegante a questo nuovo problema consiste nel dichiarare Sensor come classe **astratta**. Una classe astratta è una classe parzialmente definita, utile appunto per modellare contesti in cui un insieme di classi include operazioni con la stessa semantica ma che saranno implementate da metodi differenti. Tali operazioni, all’interno della classe astratta (che svolgerà il ruolo di super-classe), sono a loro volta astratte, ovvero senza implementazione.

Sia le classi che le operazioni astratte si indicano:

- Col modificatore ***abstract*** in Java

- Con il *corsivo* e/o con lo stereotipo {abstract} in UML (non c’è una simbologia grafica univoca)

NB: Non è mai possibile effettuare l’istanziazione di una classe astratta: il compilatore Java impedisce la creazione di oggetti di una classe marcata con la keyword *abstract*. Inoltre, se una classe contiene almeno un’operazione astratta, deve essa stessa essere dichiarata come tale, altrimenti si andrebbe incontro a un errore di compilazione. Normalmente, le variabili su classi astratte si usano più che altro per mantenere riferimenti a istanze (concrete) di sottoclassi. Rimane comunque possibile dichiarare una classe come astratta pur in assenza di operazioni astratte: benché in tal caso la classe sia, da un punto di vista prettamente strutturale, completamente definita, il compilatore ne impedisce, come richiesto, l’istanziazione diretta.



**Polimorfismo**

È la capacità di una classe di “comportarsi” in modi differenti a seconda delle specifiche situazioni. Nei linguaggi a oggetti tipizzati (come il Java), il polimorfismo si realizza tramite l’ereditarietà, introducendo una super-classe astratta A e delle sottoclassi B1, B2, …, Bn, dichiarando una variabile s di tipo A e, solo a run-time e in base a determinate condizioni, attribuendo a s un oggetto di tipo B1, B2, …, oppure Bn. Il più delle volte, anche con la tecnica del polimorfismo le classi figlie ridefiniscono o implementano alcune operazioni della classe parent. In tal caso, i relativi metodi all’interno delle sottoclassi sono detti **polimorfi**, in quanto la medesima operazione si comporterà in modo diverso a seconda del particolare tipo di oggetto su cui sarà invocata.

In poche parole, il polimorfismo consente di progettare e realizzare sistemi che “perfezionano” il loro comportamento solo a tempo di esecuzione, e permette alle istanze di classi differenti di rispondere allo stesso messaggio in modi diversi, a seconda delle specifiche implementazioni dell’operazione invocata. Questo ha come vantaggio il fatto che l’utilizzatore del sistema (identificato tramite la classe Client), pur

avendo a disposizione soltanto la variabile s (che ha come tipo la super-classe) di cui sopra, è in grado di trattare in modo omogeneo tutti gli oggetti che forniscono un dato insieme di servizi, a prescindere dalle loro implementazioni interne definite dalle rispettive sottoclassi. I sistemi polimorfici hanno anche il vantaggio di essere molto flessibili e facili da estendere: in particolare, se si vuole introdurre un nuovo oggetto che offre gli stessi servizi di B1, B2, …, Bn ma con qualche differenza nell’implementazione, basta aggiungere una nuova sottoclasse Bn+1 in modo del tutto trasparente al Client.

**Condizioni necessarie per avere una soluzione polimorfa**

1) Avere una gerarchia di classi.

2) Avere una variabile s che abbia come tipo la super-classe astratta.

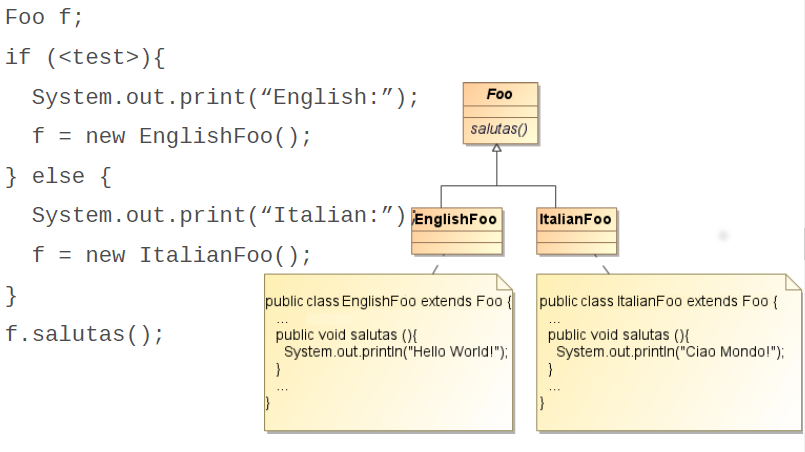
3) Introdurre un **meccanismo di scelta**, in base al quale sarà possibile creare un’istanza per una sottoclasse e associarne il riferimento alla variabile s (nota: senza il meccanismo di scelta, il polimorfismo verrebbe meno poiché il sistema si comporterebbe comunque in un modo deterministico e già noto a tempo di compilazione).

4) Utilizzare un linguaggio di programmazione che abbia un supporto di **binding dinamico** (e.g. Java).

Binding = meccanismo di associazione delle operazioni agli effettivi metodi da eseguire.

Dinamico = che avviene solo a run-time.

Nota: rivedere incapsulamento per accesso ad attributi con get() e set().

 **🡨 Meccanismo di scelta**

**Overriding vs overloading**

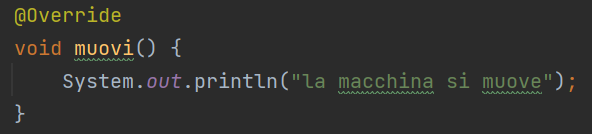
- **Overriding:** si ha quando si sovrascrive il comportamento di un’operazione ereditata dalla classe parent (o da qualunque classe antenata) cambiandone il metodo. Precondizioni affinché la classe B possa effettuare un overriding del metodo m definito nella classe A:

1) Avere una gerarchia di classi (i.e. A generalizza B).

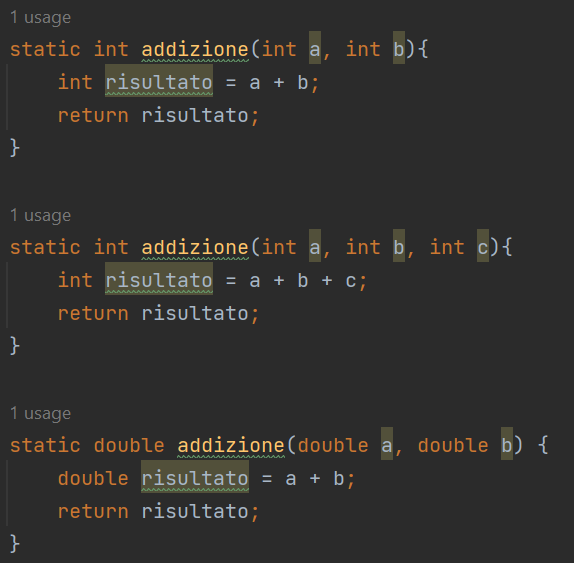
2) I metodi A.m, B.m devono avere la stessa segnatura.

3) A.m è un metodo pubblico o protetto, oppure B.m sovrascrive un altro metodo che a sua volta sovrascrive A.m.

NB: È fortemente consigliato contrassegnare i metodi soggetti a overriding all’interno della classe figlia con l’annotazione @Override. Così, quando si commettono degli errori di battitura nella segnatura di un metodo da ridefinire, il compilatore di Java riconosce la discrasia e genera un errore in compilazione.



- **Overloading**: si ha quando si sovraccarica il comportamento di un’operazione ereditata dalla classe parent (o da qualunque classe antenata) aggiungendo un secondo metodo a quello che esiste già; i due metodi dovranno avere lo stesso nome ma segnature diverse (e, quindi, liste di parametri di tipo diverso).



Nota: si può fare anche con i costruttori.

# Lezione14 – Binding in java

- **Binding**: meccanismo di associazione tra l’invocazione di un’operazione con l’effettivo metodo da eseguire.

- **Early binding**: viene effettuato a tempo di compilazione / linking prima dell’esecuzione del programma, basandosi solo sul tipo della variabile che referenzia l’oggetto coinvolto nell’esecuzione; viene definito anche **static binding**.

- **Late binding**: viene effettuato a tempo di esecuzione basandosi sull’effettivo tipo dell’oggetto coinvolto nell’esecuzione; viene definito anche **dynamic binding** o **runtime binding**.

Java effettua sia l’early binding con **riferimenti simbolici** a tempo di compilazione / linking, sia il late binding a tempo di esecuzione.

Nei linguaggi che supportano il binding dinamico il compilatore non puo (o meglio non vuole) inferire l'esatto tipo degli oggetti; si utilizzano meccanismi per recuperare il tipo degli oggetti a run-time ed invocare il metodo corretto. Non esiste una strategia univoca per implementare binding dinamico: in generale, una parte delle informazioni sul tipo di dato sono incluse nella rappresentazione interna degli oggetti.

Esempio: Supponiamo di voler eseguire il seguente estratto di codice all’interno del *main*:

*Foo f;*

*if(<test>) {*

*f = new EnglishFoo();*

*} else {*

*f = new ItalianFoo();*

*}*

*f.salutas();*

- Early binding in Java: “Considera il metodo che si chiama salutas() dentro la classe Foo”.

- Late binding in Java: “Considera il metodo che si chiama salutas() e vallo a cercare dentro la dichiarazione della classe cui fa riferimento la variabile f a runtime”.

**Casi in cui Java non effettua il late binding**

Fondamentalmente in Java la fase di late binding non è prevista nel momento in cui viene invocato un metodo caratterizzato da almeno uno dei seguenti tre modificatori:

- **Static**: poiché i metodi statici hanno un ambito di classe e non di istanza, non hanno bisogno di un linking con un particolare oggetto. Ne consegue che, se viene invocato un metodo statico su una variabile f di tipo Foo, a prescindere dal tipo specifico dell’eventuale oggetto riferito da f, verrà eseguito deterministicamente quel metodo all’interno della classe Foo.

- **Final**: un metodo caratterizzato dal modificatore *final* all’interno della classe Foo non può essere soggetto a overriding nelle eventuali sottoclassi: con questa keyword si sta dunque dichiarando esplicitamente al compilatore di disabilitare il binding dinamico.

- **Private**: un metodo caratterizzato dal modificatore *private* all’interno della classe Foo non può essere proprio acceduto dalle eventuali sottoclassi: infatti, in Java i metodi privati sono implicitamente *final*.

**Stack di programma**

È un’area di memoria utilizzata per mantenere lo stato locale di una computazione. Ne esiste uno per ogni thread in esecuzione nella JVM. A seguito della chiamata di una sotto-funzione, viene allocata una nuova area di stack (una nuova porzione) dove vengono salvati:

- I parametri formali (inizializzati ai valori dei parametri attuali) e le variabili locali durante la chiamata alla sotto-funzione.

- Il valore di ritorno computato dalla sotto-funzione dopo la terminazione di quest’ultima.

**Heap di programma**

È un’area di memoria dinamica utilizzata per lo più per mantenere i dati di un’applicazione (e.g. istanze delle classi), i quali sono globalmente accessibili per mezzo di un puntatore di riferimento (che spesso è memorizzato in un’area di stack). Si tratta di un’area di memoria unica e condivisa da tutti i thread in esecuzione nella JVM.

La gestione dell’heap è principalmente delegata al programmatore: infatti, la sua allocazione è esplicita e, in Java, avviene attraverso l’operatore *new*. D’altra parte, i dati contenuti in quest’area di memoria vi permangono finché:

- Non si effettua una **deallocazione esplicita** della memoria, come nel caso di C / C++ rispettivamente attraverso gli operatori free(), dispose().

- Non viene avviata una politica di **deallocazione implicita** della memoria, come nel caso di Java mediante l’esecuzione del garbage collector.

- Il programma non termina la sua esecuzione.

**JVM sotto il cofano** La JVM alloca e gestisce sia lo stack di programma, sia l’heap di programma. In particolare, nell’heap mantiene un **run-time constant pool** per ogni classe istanziata a runtime e una **JVM method area**.

- Il run-time constant pool è un’area di memoria relativa a una specifica classe A (di cui è stato istanziato almeno un oggetto) e contiene le costanti di A, i riferimenti di A risolti a tempo di compilazione e i riferimenti di A da risolvere a runtime. I vari run-time constant pool si trovano all’interno della JVM method area.

- La JVM method area è una zona di memoria unica e condivisa da tutti i thread, e fa logicamente parte dell’heap. Tuttavia, ciò non impone che la JVM method area sia soggetta a garbage collection o ad altri comportamenti associati alle normali strutture dati allocate nell’heap. Tale zona di memoria, oltre al run-time constant pool, mantiene per ogni classe altre informazioni, come ad esempio il codice di metodi e costruttori.

Principalmente, la JVM implementa l’invocazione di metodi attraverso le seguenti istruzioni del byte code: ***invokevirtual***, ***invokestatic***, ***invokespecial***.

- **Invokevirtual**: viene consultato il run-time constant pool accedendo in base alla corrente istanza effettiva e usando parametri attuali.

- **Invokestatic**: viene consultato il run-time constant pool accedendo in base alla classe referenziata (non in base all’istanza effettiva poiché, col modificatore static, potrebbe non esistere) e usando parametri attuali.

- **Invokespecial**: è un ibrido tra le soluzioni precedenti ed è usato per gestire casi particolari come il *main* e i metodi *private* / *final*.

In particolare, come nel caso di *invokestatic*, viene consultato il run-time constant pool accedendo in base alla classe referenziata e usando parametri attuali.

D’altra parte, come nel caso di *invokevirtual*, si tiene comunque traccia dell’istanza corrente nello stack, per cui il riferimento *this* (che referenzia appunto l’istanza corrente) è implicitamente passato al contesto di esecuzione del metodo.

**Invocazione di metodi**

– **invokevirtual**: il riferimento all'oggetto sul quale viene invocato il metodo e allocato nello STACK insieme ai parametri attuali. Il riferimento this è implicitamente passato al contesto di esecuzione del metodo.

– **invokestatic** : nello STACK vengono allocati solo i parametri attuali che devono essere passati al metodo.

– **invokespecial**: il riferimento all'oggetto sul quale viene invocato il metodo e allocato nello STACK insieme ai parametri attuali. Il riferimento this e implicitamente passato al contesto di esecuzione del metodo.

# Lezione17 – Interfacce

The fragile base-class problem

“Ogni volta che aggiungi un nuovo metodo o una nuova variabile di istanza a una classe A, qualunque altra classe referenzi A richiederà una ricompilazione, oppure si romperà”.

“Il Fragile Base Class Problem (FBCP) viene sollevato quando i cambiamenti alle sottoclassi o alla super-classe potrebbero indurre le istanze di queste classi a comportarsi in modo inaspettato. Definiamo una Fragile Base Class Structure (FBCS) come un insieme di due classi legate da una relazione (non necessariamente diretta) di generalizzazione e con dichiarazioni e definizioni di uno specifico metodo. Una FBCS è un contesto in cui il FBCP può presentarsi se, per esempio, la sottoclasse effettua l’overriding di un metodo della super-classe in modo tale da introdurre una mutua ricorsione”.

Esempio: Consideriamo la seguente FBCS. Supponiamo di voler modificare il metodo setMuteOn() all’interno della super-classe Sensor nel seguente modo (notiamo che le modifiche verranno ereditate dalla sottoclasse DualModeSensor):

*public void setMuteOn() {*

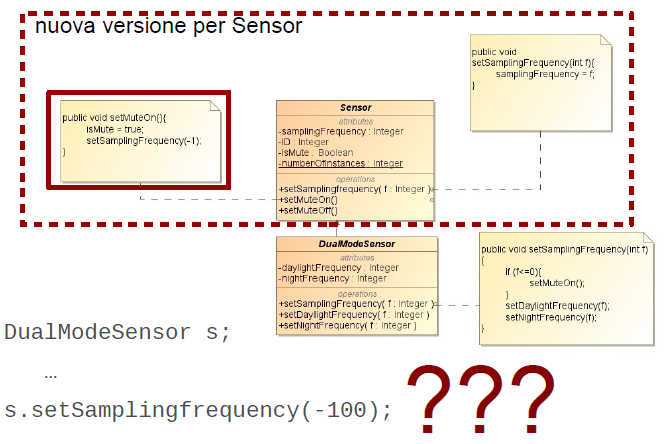
*this.isMute = true;*

*setSamplingFrequency(-1);*

*}*

Questo cambiamento ha perfettamente senso per la classe Sensor; tuttavia, causa dei grossi problemi all’interno della classe figlia: infatti, se si inviasse un messaggio a un’istanza di DualModeSensor effettuando una chiamata al metodo setMuteOn(), si otterrebbe una mutua ricorsione infinita, dato che setMuteOn() e setSamplingFrequency() si invocheranno a vincenda finché non si raggiungerà una condizione di stack overflow.

Il fragile base-class problem non si può propriamente risolvere ma si può comunque mitigare: vediamo come.



**Interfaccia**

È una collezione di operazioni utilizzata per specificare un servizio di una classe o di un componente del sistema. Definisce solo la segnatura delle operazioni (che, quindi, risultano essere tutte astratte) e le costanti: di conseguenza, non può contenere costruttori, variabili statiche, variabili di istanza e metodi statici. L’interfaccia rappresenta dunque una specifica parziale e, in quanto tale, non può mai essere istanziata.

Le interfacce si indicano:

- Con la keyword *interface* al posto della keyword *class* in Java

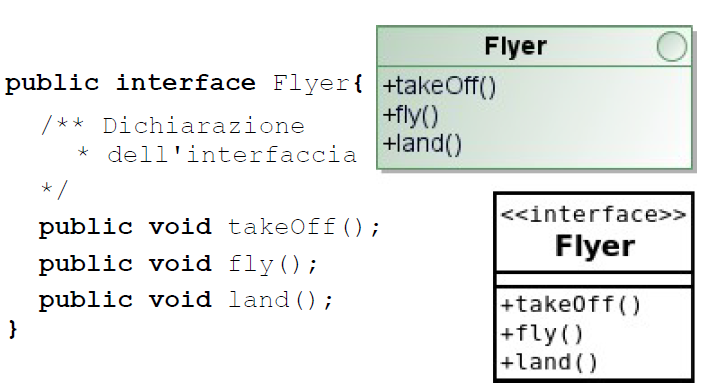
- Con un cerchio o con lo stereotipo *<<interface>>* in UML (di nuovo, non c’è una simbologia grafica univoca)

Regole pratiche per la dichiarazione di interfacce:

- Poiché le uniche variabili ammesse all’interno delle interfacce sono le costanti, devono necessariamente essere inizializzate e non potranno mai essere modificate successivamente: è come se fossero dichiarate *final*.

- I metodi dichiarati in un’interfaccia sono sempre pubblici.

NB: una classe può implementare più interfacce.



**Interfaccia vs classe astratta**

In sostanza, entrambi i concetti:

- Modellano operazioni che non sono associate ad alcun metodo.

- Impongono alle sottoclassi concrete l’overriding delle loro operazioni.

- Sono collezioni di operazioni utilizzate per specificare un servizio di una classe.

- Non consentono la creazione diretta di istanze.

A prima vista, i due concetti sembrerebbero abbastanza “sovrapposti”, per cui potrebbe non essere chiara la necessità di disporre sia di interfacce che di classi astratte. Proviamo dunque a evidenziare qualche sottile differenza.

Interfaccia:

1) Ha esclusivamente operazioni che non hanno associato alcun metodo, per cui non può contenere codice.

2) Non definisce alcun attributo che non sia una costante.

3) Modella, attraverso le operazioni, semplicemente un modo d’uso di un sottosistema.

4) Generalmente rappresenta una vista del sistema piuttosto che una sua parte.

Classe astratta:

1) Può anche non avere operazioni astratte; infatti, può essere usata per fornire codice comune alle sue sottoclassi.

2) Può definire qualunque tipo di attributi, per cui prevede il concetto di stato per l’elemento modellato.

3) Generalmente modella una parte della struttura statica del sistema, mettendo in risalto entità del dominio parzialmente definite o elementi parzialmente definiti utili nell’ingegnerizzazione del sistema.

In particolar modo, l’introduzione delle interfacce risulta importante perché esse vengono usate per:

- **Interconnettere sottosistemi diversi**: il sistema viene strutturato in base all’insieme delle interfacce definite dai vari sottosistemi.

- **Definire architetture astratte** basate sulle interazioni.

- **Aumentare la modularità del sistema**: gran parte delle attività di progettazione si concentra sull’individuazione e sulla modellazione delle principali forme di interazione per mezzo di interfacce.

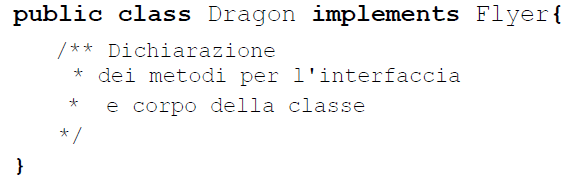
**Realizzazione**



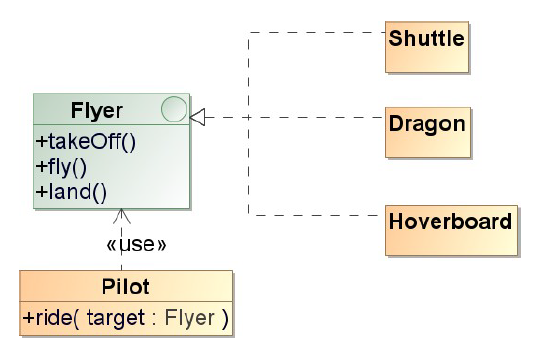
È una relazione che lega un’interfaccia J con una classe A in modo tale che A, se è concreta, deve implementare (realizzare, appunto) tutte le operazioni astratte di J, fornendo dei metodi che rispettino il contratto imposto da J e che, quindi, mantengano la stessa segnatura.

La realizzazione:

- In Java viene definita tramite l’espressione “**implements** NomeInterfaccia” all’interno della definizione della classe.



- In UML è raffigurata tramite una freccia tratteggiata con un triangolo bianco rivolto verso l’interfaccia.



**Ereditarietà vs realizzazione**

Facciamo un confronto tra la relazione di generalizzazione (ereditarietà) e la relazione di realizzazione.

Ereditarietà:

1) Permette la trasmissione delle caratteristiche comuni (attributi, relazioni, metodi) da una classe all’altra.

2) Può essere usata esclusivamente se tra due classi esiste una ovvia relazione *is-a-kind-of*: infatti, è la forma più forte di interdipendenza tra classi e un cattivo uso può causare il fragile base-class problem.

3) È davvero necessaria se e solo se lo scopo è ereditare anche dei dettagli implementativi dalla super-classe: è nata come la forma più basilare di riuso e, col tempo, ha assunto delle forti caratteristiche semantiche, come il principio di sostituibilità di Liskov.

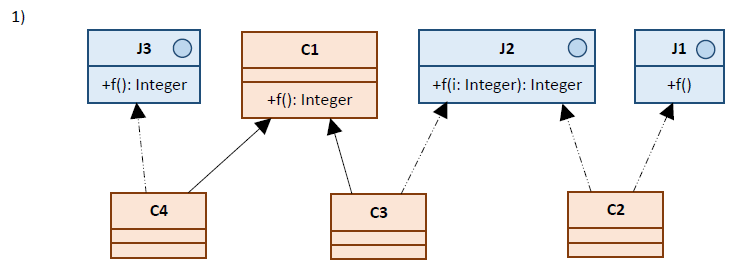
Realizzazione:

1) Implica l’accettazione delle specifiche di interazione previste dall’interfaccia, che sono descritte tramite le operazioni pubbliche dichiarate all’interno dell’interfaccia stessa.

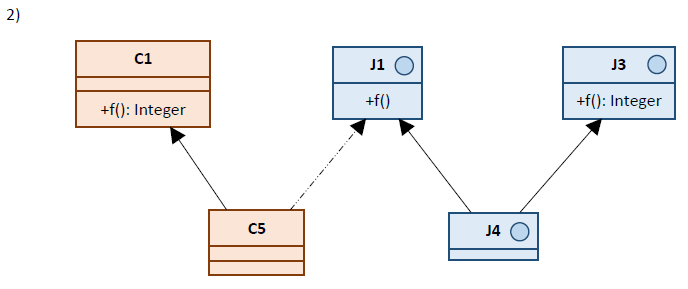
2) È utile quando si vuole definire un contratto senza accettare vincoli sui dettagli implementativi ma garantendo che tale contratto sia rispettato almeno sintatticamente: perciò, l’interfaccia non offre alcuna possibilità di riuso ed è più flessibile e robusta dell’ereditarietà.

Entrando più nello specifico nel contesto di Java, sappiamo già che non è ammessa l’ereditarietà multipla tra le classi; tuttavia, Java consente alle classi di implementare più interfacce contemporaneamente e, allo stesso modo, ammette l’ereditarietà multipla tra le sole interfacce. A livello prettamente semantico, una giustificazione a questo fenomeno è: una classe ha uno e un solo tipo (per cui essa *is-a-kind-of* una e una sola altra classe che, nel caso più generale, è Object); tuttavia, può comunque manifestarsi attraverso viste differenti, ciascuna delle quali definisce un modo d’uso della classe.

NB: Bisogna gestire opportunamente eventuali casi di collisione di nomi quando si combinano ereditarietà e realizzazione.



In questo caso non si ha alcun errore poiché non si presenta alcun conflitto tra le operazioni ereditate / da realizzare all’interno delle sottoclassi C2, C3, C4.



In questo secondo caso, invece, viene sollevato un errore in compilazione poiché sia nell’interfaccia J4, sia nella classe C5 si presenta un conflitto: in particolare, J4 e C5 devono contentenere due operazioni che, dal punto di vista di Java, hanno la medesima segnatura ma prevedono tipi di ritorno differenti.

# Lezione18 – Associazione, aggregazioni, composizioni e dipendenze

**Associazione**



È una relazione strutturale che collega oggetti di classi logicamente connesse tra loro, e indica la possibilità che le istanze di una classe possano inviare messaggi alle istanze delle classi associate. Un’associazione può essere:

- **Simmetrica** se è navigabile nelle due direzioni, **asimmetrica** altrimenti.

- **Riflessiva** se coinvolge istanze della medesima classe.

In UML viene rappresentata con:

- Una freccia continua con la punta rivolta verso la classe che può ricevere messaggi se si tratta di un’associazione asimmetrica

- Una linea continua semplice altrimenti

Inoltre, un’associazione può specificare:

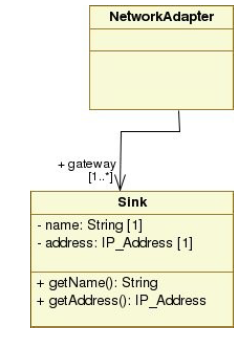
- Il **nome**

**-** La **visibilità**

- Il **ruolo** delle classi coinvolte

- Uno **stereotipo**

- La **cardinalità**, che indica il numero di oggetti di ciascuna delle due classi che possono partecipare all’associazione



Esempio:

*public class Sink {*

*private String name;*

*private IP\_Address address;*

*public Sink(String n, IP\_Address addr) {*

*this.name = n;*

*this.address = addr;*

*}*

*public String getName() {*

*return this.name;*

*}*

*public IP\_Address getAddress() {*

*return this.address;*

*}*

*}*

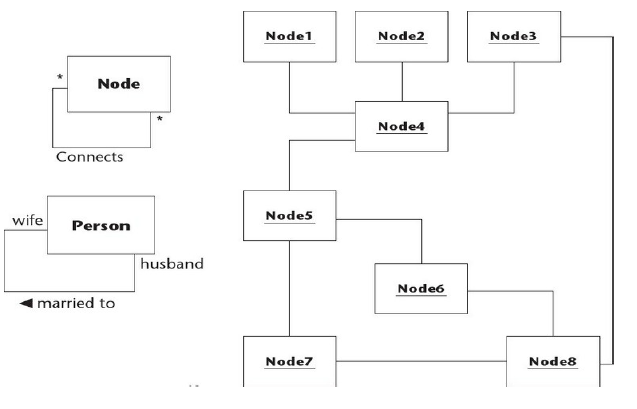
*public class NetworkAdapter {*

*private Vector<Sink> gateway;* // Qui si concretizza l’associazione dalla classe NetworkAdapter alla

… // classe Sink; si può osservare che il nome è *gateway*, la visibilità è *}*

// privata e la cardinalità è uno a molti.

*}*



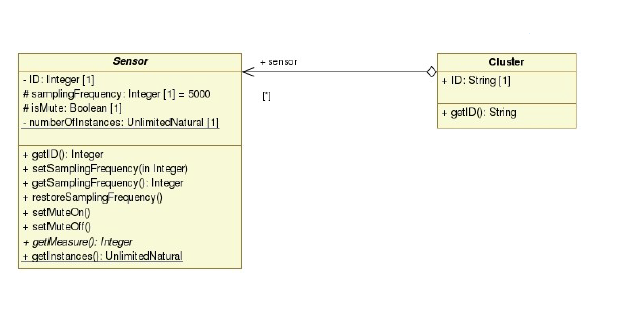
**Aggregazione**



È una relazione di tipo gerarchico che coinvolge una classe B che rappresenta un’entità autonoma e una classe A che aggrega l’entità autonoma; dal punto di vista delle istanze, si ha che un oggetto di A contiene un oggetto di B. Questa relazione è denominata anche ***whole-part*** (*intero-parte*).

In UML viene rappresentata attraverso una linea continua con un rombo bianco dalla parte della classe contenitore (classe aggregante).

Pur essendo una forma di associazione più forte, l’aggregazione non impone dei vincoli sul ciclo di vita degli oggetti aggregati: l’eliminazione di un’istanza della classe aggregante A (whole) non comporta l’eliminazione delle istanze della classe aggregata B (part). Viceversa, gli oggetti di A, per esistere, potrebbero necessitare di uno o più oggetti di B (dipende dal contesto).



Esempio:

*public class Sink { //****Sink come esempio sopra***

*private String name;*

*private IP\_Address address;*

*public Sink(String n, IP\_Address addr) {*

*this.name = n;*

*this.address = addr;*

*}*

*public String getName() {*

*return this.name;*

*}*

*public IP\_Address getAddress() {*

*return this.address;*

*}*

*}*

*public class NetworkAdapter {*

*private Vector<Sink> gateway;* // Si tratta di un’aggregazione uno a molti: un’istanza di

*public NetworkAdapter(Vector<Sink> v) {* // NetworkAdapter aggrega più istanze di Sink tramite l’uso

*this.gateway=v;* // di un vettore.

*}*

*public NetworkAdapter() {*

*this.gateway=null;* // Questo costruttore senza parametri mostra come, nel

*}* // nostro esempio, un’istanza di NetworkAdapter possa

*…* // aver senso di esistere anche in assenza di istanze di Sink.

*}*

NB:

Le aggregazioni circolari, come la seguente:

- A aggrega B –

B aggrega C

- C aggrega A

sono semanticamente errate!

**Composizione**

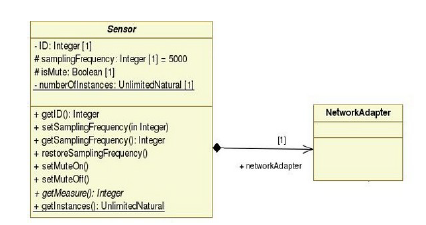


È un’aggregazione forte che coinvolge una classe B che rappresenta l’entità componente (part) e una classe A che è relativa all’entità composta (whole). Stavolta, il ciclo di vita degli oggetti componenti dipende da quello dell’oggetto composto. In particolare:

- In C++, quando viene invocato il comando dispose() su un oggetto composto, prima dell’eliminazione di quest’ultimo vengono deallocate le istanze componenti.

- In Java, nel momento in cui viene distrutta l’istanza composta per opera del garbage collector, devono necessariamente essere eliminate anche tutte le istanze componenti. Ne consegue che le classi componenti non possono tassativamente essere relazionate con classi diverse da quella composta (**principio di esclusività**). Perciò, ogni oggetto può far parte al più di un solo oggetto composto alla volta (mentre nell’aggregazione una parte può essere condivisa da più istanze aggreganti).

La composizione in UML viene rappresentata attraverso una linea continua con un rombo nero dalla parte della classe composta.



Esempio:

*public class Sink { //****Sink come esempio sopra***

*private String name;*

*private IP\_Address address;*

*public Sink(String n, IP\_Address addr) {*

*this.name = n;*

*this.address = addr;*

*}*

*public String getName() {*

*return this.name;*

*}*

*public IP\_Address getAddress() {*

*return this.address;*

*}*

*}*

*public class NetworkAdapter {*

*private Sink gateway;* // Non passare mai il riferimento di

*public NetworkAdapter(String n, IP\_Address addr) {* // this.gateway al di fuori di questa classe!

*this.gateway = new Sink(n, addr);*

*}*

*public NetworkAdapter(Sink s) {*

*this.gateway = new Sink(s.getName(), s.getAddress());*

*}*

*…*

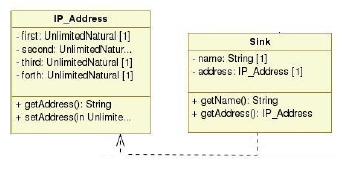
*}*

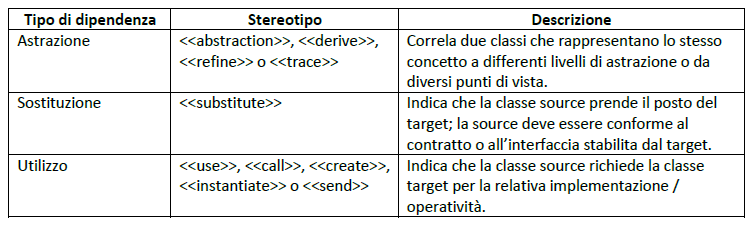
**Dipendenza**



È una relazione in cui una classe source utilizza o dipende da una classe target. In pratica, sta a indicare che, se la classe target subisce una variazione, allora deve essere cambiata anche l’implementazione della classe source. In UML viene rappresentata mediante una freccia tratteggiata con la punta rivolta verso la classe target.

Di fatto, la semantica di questa relazione non è univoca, per cui vengono spesso utilizzati degli stereotipi che mostrano la natura precisa della dipendenza (vedere tabella nella pagina seguente).





Comunque sia, il più delle volte, la relazione di dipendenza indica che la classe source svolge una delle seguenti funzioni:

- Utilizzare temporaneamente la classe target come variabile globale

- Utilizzare temporaneamente la classe target come parametro per una delle relative operazioni

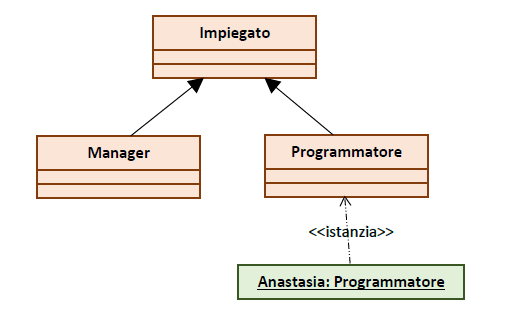
- Utilizzare temporaneamente la classe target come variabile locale per una delle relative operazioni

- Inviare un messaggio a un’istanza della classe target

NB: La dipendenza, all’interno del class diagram, non è una relazione da esplicitare sempre, bensì solo quando si vuole avvisare il programmatore del fatto che l’implementazione della classe source dipende fortemente dall’implementazione della classe target (altrimenti il class diagram diverrebbe illeggibile).

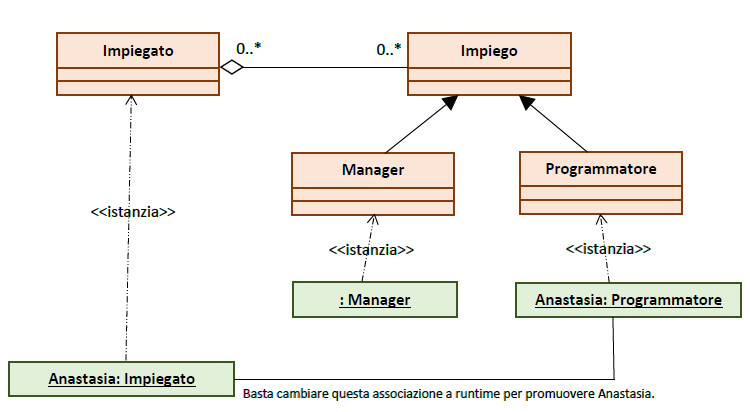
**Pattern della metamorfosi**

Supponiamo di voler modellare i ruoli organizzativi in un’azienda. Una possibile soluzione è la seguente:



Ma cosa succede se Anastasia da programmatore diventa manager? L’unica possibilità consisterebbe nel creare una nuova istanza *Anastasia* di tipo Manager, riversare i vecchi dati della vecchia istanza *Anastasia* all’interno di quella nuova e poi deallocare la vecchia istanza. Tuttavia, nella pratica, la porzione di stato che era presente all’interno della super-classe Impiegato andrebbe persa a causa dell’information hiding attuato dalla super-classe stessa.

È dunque opportuno raffinare il nostro modello, e lo facciamo applicando il cosiddetto **pattern della metamorfosi:**



**Raffinazione di relazioni**

Normalmente, la fase di analisi del processo di sviluppo del software enfatizza l’uso di generalizzazioni e associazioni. D’altra parte, la fase di progettazione enfatizza maggiormente l’utilizzo di realizzazioni (che emergono con l’introduzione delle interfacce), aggregazioni, composizioni e dipendenze.

– generalizzazioni → realizzazioni

● classi → interfacce

– associazioni → aggregazioni

– associazioni → composizioni

– associazioni → dipendenze

Vediamo qualche esempio di raffinazione di relazioni (in particolare di associazioni) nel passaggio dalla fase di analisi alla fase di progettazione del processo di sviluppo del software.

Associazioni uno-a-uno:

- Possono essere raffinate in relazioni di composizione.

- Altrimenti, la classe che avrebbe giocato il ruolo di *part* può essere definita come attributo della classe che avrebbe giocato il ruolo di *whole*.

Associazioni molti-a-uno:

- Poiché in questo caso un oggetto *part* potrebbe essere condiviso tra più *whole*, le associazioni molti-a-uno non possono evolvere in composizioni, bensì possono essere raffinate solo in aggregazioni.

- se non presenti, specificare molteplicita e ruoli

- verificare se esistono i presupposti per l'applicazione del “pattern” *whole-part*

– identificare l'estremo *whole*

– identificare l'estremo *part*

- controllare la molteplicita per *whole* :

– se 0..1 o 1 → è possibile (non obbligatorio) raffinare come composizione

– altrimenti → raffinabile solo come aggregazione

- verificare la navigabilita dell'associazione

– (in generale) nella progettazione si fa riferimento a relazioni unidirezionali

# Lezione21 – Progettazione con responsabilità

L’esperienza nella modellazione O.O. suggerisce principi generali, soluzioni ricorrenti e aspetti da considerare in funzione del contesto, della fase del processo adottato e degli obiettivi da raggiungere. Tuttavia, potrebbe comunque essere intesa come una “forma creativa” basata su abilità personali e attitudini derivanti dallo studio e dalla pratica (ovvero una forma di artigianato). Per evitare ciò, è stato introdotto un modo universale di pensare alla progettazione O.O.: l’**RDD** (**Responsibility Driven Design**), secondo cui gli oggetti software sono associati a una descrizione delle loro **responsabilità**. La responsabilità di un oggetto software è un obbligo sulla sua struttura o sul suo comportamento in relazione al suo ruolo all’interno del sistema; in genere, viene assegnata dal progettista.

Le responsabilità si classificano in:

- **Responsabilità di fare**: istanziare un oggetto, computare un calcolo, dare inizio alle attività di altri oggetti, controllare e coordinare parte delle attività di altri oggetti.

- **Responsabilità di conoscere**: essere a conoscenza dei propri dati privati incapsulati, degli oggetti correlati e delle cose che possono essere derivate/calcolate.

Uno strumento che si basa sui concetti di base di RDD è **GRASP** (**General Responsibility Assignment Software Patterns**), che mette in luce alcuni principi di supporto alla progettazione O.O., i quali sono utili per migliorare la documentazione del software e standardizzare i vecchi modelli di programmazione. GRASP comprende principalmente nove **pattern**:

1) **Creator**

2) **Information Expert**

3) **Low Coupling**

4) **High Coesion**

5) **Controller**

6) **Polymorphism**

7) **Pure Fabrication**

8) **Indirection**

9) **Protected Variations**

Di seguito ne analizzeremo qualcuno nel dettaglio.

NB: Non è sempre possibile applicare contemporaneamente tutti i pattern GRASP: è compito del progettista valutare le differenti soluzioni e scegliere cosa prediligere.

|  |
| --- |
| *Nota: un pattern è una coppia (problema, soluzione) ben conosciuta e con un nome, con consigli su come applicarla in nuovi contesti e con una discussione sui relativi compromessi, implementazioni, variazioni e così via. Notiamo che l’espressione “nuovo pattern” è un ossimoro, in quanto un pattern, per definizione, non esprime nuove idee della progettazione, bensì codifica idiomi e principi esistenti, di conoscenza comprovata e verificata.* |

**Creator**

Problema:

Chi deve essere responsabile della creazione di una nuova istanza di una classe?

La creazione degli oggetti è una delle attività più comuni in un sistema orientato agli oggetti. Di conseguenza, è utile avere un principio generale per l’assegnazione delle responsabilità di creazione. Se queste vengono assegnate bene, il progetto può sostenere un accoppiamento basso, una chiarezza maggiore, incapsulamento e riusabilità.

Soluzione:

Assegna alla classe B la responsabilità di creare un’istanza della classe A se almeno una delle seguenti condizioni è verificata:

- B aggrega o compone oggetti di tipo A.

- B registra A.

- B utilizza strettamente A.

- B possiede i dati per l’inizializzazione di A, che saranno passati ad A al momento della sua creazione.

B viene chiamato *creatore* di oggetti A. Se esistono più classi che hanno i requisiti per poter creare istanze di A, solitamente va preferita una classe B che aggrega o contiene A o comunque una classe B che rispetta il numero più alto di condizioni.

Controindicazioni:

Spesso la creazione è di notevole complessità, per esempio se l’oggetto da istanziare apparterrà a una famiglia di classi simili e la sua creazione va fatta in base a una proprietà esterna che determinerà il suo tipo esatto. In questi casi è consigliabile delegare la creazione a una classe di supporto (helper) chiamata *Concrete Factory* o *Abstract Factory* (che analizzeremo successivamente) anziché utilizzare la classe suggerita da Creator.

Vantaggi:

Creator favorisce un accoppiamento basso, il che implica minori dipendenze di manutenzione e maggiori opportunità di riuso. Ciò è dovuto al fatto che la classe creata deve essere già visibile di suo alla classe creatore, grazie alle associazioni esistenti che ne hanno motivato la scelta come creatore.

**Information Expert**

Problema:

Qual è il principio generale nell’assegnazione di responsabilità agli oggetti?

Un modello di progetto può definire centinaia o migliaia di classi software e un’applicazione può richiedere centinaia o migliaia di responsabilità da soddisfare. Durante la progettazione a oggetti, quando vengono definite le interazioni tra gli oggetti, si effettuano delle scelte sull’assegnazione delle responsabilità alle classi software. Se le scelte vengono fatte bene, i sistemi tendono a essere più facili da comprendere, da mantenere e da estendere e consentono maggiori opportunità di riuso dei suoi componenti.

Soluzione:

Assegna una responsabilità all’esperto delle informazioni, ovvero alla classe che possiede le informazioni necessarie per soddisfare la responsabilità.

Esempio: chi deve essere responsabile di conoscere il totale complessivo di una vendita? Si dovrebbe cercare la classe che possiede le informazioni necessarie per determinare il totale, come i singoli prodotti coinvolti nella vendita e la somma dei loro prezzi. Perciò, i singoli prodotti devono a loro volta avere la responsabilità di conoscere i relativi prezzi (altrimenti non sarebbe possibile calcolarne il totale).

Ma, per analizzare le classi che hanno le informazioni necessarie, bisogna cercare nel modello di dominio o nel modello di progetto? - Se ci sono già delle classi pertinenti nel modello di progetto, si analizzi questo per primo. - Altrimenti, si guardi nel modello di dominio, cercado di usare (o di estendere) le sue rappresentazioni per ispirare la creazione di classi di progetto corrispondenti.

Questo approccio favorisce un salto rappresentazionale basso in cui la progettazione software degli oggetti richiama i concetti relativi a come è organizzato il dominio reale.

Controindicazioni:

In alcune situazioni, una soluzione suggerita da Information Expert non è opportuna, solitamente a causa di problemi di accoppiamento e di coesione. Per esempio, chi deve essere responsabile di salvare una classe A in una base di dati? Certamente gran parte delle informazioni da salvare si trovano in A, per cui Expert potrebbe sostenere che la responsabilità spetti alla classe A. Per estensione logica di questa decisione, ogni classe avrebbe i propri servizi per salvare sé stessa nella base di dati. Ma, in base a questo ragionamento, sorgerebbero dei problemi di coesione, accoppiamento e duplicazione. Infatti, A deve ora contenere anche la logica relativa alla gestione della base di dati e non è più incentrata solo sulla logica applicativa pura.

Vantaggi:

- L’incapsulamento delle informazioni viene mantenuto, poiché gli oggetti usano le proprie informazioni per adempiere ai propri compiti. Di solito questo sostiene un accoppiamento basso, che dà luogo a sistemi più robusti e mantenibili.

- Il comportamento è distribuito tra tutte le classi che possiedono le informazioni richieste, incoraggiando definizioni di classi più coese, leggere e più facili da comprendere e da mantenere.

**Low Coupling**

Problema:

Come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e una maggiore opportunità di riuso?

L’**accoppiamento** (**coupling**) è una misura di quanto fortemente una classe è connessa ad altre classi, ha conoscenza di altre classi e dipende da altre classi. Le classi fortemente accoppiate possono essere inopportune e presentare i seguenti problemi:

- Devono necessariamente essere modificate ogni qual volta avviene un cambiamento nelle classi correlate.

- Sono più difficili da comprendere in isolamento, ovvero senza comprendere anche le classi da cui dipendono.

- Sono più difficili da riusare poiché il loro uso richiede la presenza aggiuntiva delle classi da cui dipendono.

Soluzione:

Assegna una responsabilità in modo che l’accoppiamento rimanga basso. Usa questo principio per valutare le possibili soluzioni alternative.

Discussione:

Information Expert “sostiene” (= è condizione necessaria ma non sufficiente per) Low Coupling.

Le forme più comuni di accoppiamento tra un tipo X e un tipo Y sono:

- La classe X ha un attributo (o una variabile d’istanza o un dato membro) di tipo Y, oppure referenzia un’istanza di tipo Y o una collezione di oggetti Y.

- Un oggetto di tipo X richiama operazioni o servizi di un oggetto di tipo Y.

- Un oggetto di tipo X crea un oggetto di tipo Y.

- Il tipo X ha un metodo che contiene un elemento (parametro, variabile locale oppure tipo di ritorno) di tipo Y o che referenzia un’istanza di tipo Y.

- La classe X è una sottoclasse, diretta o indiretta, della classe Y.

- Y è un’interfaccia e la classe X implementa Y.

Low Coupling incoraggia ad assegnare una responsabilità in modo tale che la sua collocazione non faccia aumentare l’accoppiamento del progetto a un livello “troppo alto”. Il caso estremo di Low Coupling è l’assenza di accoppiamento tra classi e dà luogo a un progetto mediocre, in cui ci sono pochi oggetti attivi non coesi, complessi e che eseguono tutto il lavoro, e molti oggetti passivi con accoppiamento nullo che agiscono come semplici contenitori di dati. Un certo grado moderato di accoppiamento tra le classi è normale, anzi, è necessario per la creazione di un sistema orientato agli oggetti in cui i compiti vengono svolti grazie a una collaborazione tra oggetti connessi.

Una sottoclasse è fortemente accoppiata alla sua superclasse. Per esempio, si supponga che degli oggetti debbano essere memorizzati in modo persistente in una base di dati: si potrebbe seguire una pratica di progettazione relativamente comune che consiste nel creare una super-classe astratta PersistentObject da cui derivano le altre classi persistenti. Lo svantaggio di questa derivazione delle sottoclassi è che gli oggetti di dominio sono fortemente accoppiati a un particolare servizio tecnico, mentre il vantaggio è l’ereditarietà automatica del comportamento di persistenza.

Una forma subdola e particolarmente cattiva di accoppiamento è la presenza di codice duplicato, di solito legata a un abuso della pratica della programmazione *copia-incolla-e-modifica*. Porzioni di codice duplicato sono fortemente accoppiate tra di loro: la modifica di una copia spesso implica la necessità di modificare anche le altre copie (aumentando così lo sforzo richiesto per gestire i cambiamenti).

Controindicazioni:

Un accoppiamento alto con elementi stabili o pervasivi costituisce raramente un problema. Per esempio, un’applicazione Java può essere accoppiata senza problemi alle librerie fondamentali di Java (come le classi del package *java.lang* e le collezioni di *java.util*), poiché sono stabili e largamente diffuse. Il problema, infatti, non è l’accoppiamento alto di per sé, bensì l’accoppiamento alto con elementi instabili e soggetti a modifiche frequenti.

Vantaggi:

Una classe con un accoppiamento basso:

- Non è influenzata dai cambiamenti nelle altre classi.

- È semplice da capire separatamente dalle altre classi e componenti.

- È conveniente da riusare.

**High Cohesion**

Problema:

Come mantenere gli oggetti focalizzati, comprensibili e gestibili e, come effetto collaterale, sostenere Low Coupling?

La **coesione** è una misura di quanto fortemente siano correlate e concentrate le responsabilità di una classe dal punto di vista funzionale. In altre parole, stima:

- Quanto fortemente la definizione di una classe sia concettualmente affine alle altre.

- Quanto fortemente le operazioni fornite dalla classe siano consistenti con le responsabilità associate alla classe stessa.

Soluzione:

Assegna una responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare le possibili soluzioni alternative.

Una classe con una coesione bassa fa molte cose non correlate tra loro o svolge troppo lavoro. Classi di questo tipo non sono opportune e presentano i seguenti problemi:

- Sono difficili da comprendere.

- Sono difficili da mantenere.

- Sono difficili da riusare.

- Sono continuamente soggette a cambiamenti.

Discussione:

Esistono diverse forme di coesione relative a diversi criteri di raggruppamento delle responsabilità negli elementi software. Alcune di queste forme di coesione sono buone, altre sono meno buone o addirittura cattive. Alcune forme di coesione sono:

- **Coesione dei dati**: una classe implementa un tipo di dati (buona).

- **Coesione funzionale**: gli elementi di una classe svolgono una singola funzione (buona).

- **Coesione temporale**: gli elementi sono raggruppati perché usati nello stesso intervallo di tempo (è la forma di coesione usata negli oggetti controller ed è buona in questo caso, ma meno buona in altri casi).

- **Coesione per pura coincidenza**: una classe viene ad esempio usata per raggruppare tutti i metodi il cui nome inizia con una certa lettera (cattiva).

Come già accennato, la forma di coesione a cui si riferisce il pattern High Cohesion è la coesione funzionale.

Di seguito sono descritti alcuni scenari che illustrano vari gradi di coesione funzionale:

- **Coesione molto bassa**: una classe è la sola responsabile di molte cose in aree funzionali molto diverse. Si supponga l’esistenza di una classe RDB-RPC-Interface che è completamente responsabile dell’interazione con le basi di dati relazionali e della gestione delle chiamate di procedure remote. Si tratta di due aree funzionali ampiamente diverse, ciascuna delle quali richiede una quantità notevole di codice di supporto. Le responsabilità devono essere suddivise in una famiglia di classi, di cui una correlata all’accesso al database, e l’altra correlata alla gestione delle chiamate di procedure remote.

- **Coesione bassa**: una classe ha da sola la responsabilità di un compito complesso in una sola area funzionale. Si supponga l’esistenza di una classe RDB-Interface che è completamente responsabile dell’interazione con le basi di dati relazionali. I metodi della classe sono tutti correlati ma sono molto numerosi, e la quantità di codice di supporto è molto consistente: possono esserci centinaia o migliaia di metodi. La classe deve essere suddivisa in una famiglia di classi leggere che condividono il compito di fornire l’accesso al database.

- **Coesione moderata**: una classe ha, da sola, responsabilità leggere in poche aree diverse, che sono logicamente correlate al concetto rappresentato dalla classe ma non l’una all’altra. Si supponga l’esistenza di una classe Company che è completamente responsabile di conoscere i suoi dipendenti e conoscere le proprie informazioni finanziarie. Queste due aree non sono fortemente correlate l’una con l’altra, anche se entrambe sono logicamente correlate al concetto di una Company. Inoltre, il numero totale di metodi pubblici è piccolo, così come la quantità di codice di supporto.

- **Coesione alta**: una classe ha responsabilità moderate in un’unica area funzionale e collabora con altre classi per svolgere i suoi compiti. Si supponga l’esistenza di una classe RDB-Interface che è responsabile solo in parte dell’interazione con le basi di dati relazionali. Essa interagisce con una dozzina di altre classi correlate all’accesso al database per il recupero e il salvataggio degli oggetti.

Controindicazioni:

In alcuni casi, è giustificabile accettare una coesione più bassa. Si supponga che un’applicazione contenga delle istruzioni SQL che, in base ad altri buoni principi di progettazione, dovrebbero essere distribuite in una decina di classi: è comune che l’architetto software decida di raggruppare tutte le istruzioni SQL in una sola classe che non avrà una coesione molto alta ma potrà lavorare agevolmente sulle istruzioni SQL in una sola posizione.

Vantaggi:

- High Cohesion sostiene maggiore chiarezza e facilità di comprensione del progetto.

- High Cohesion sostiene spesso Low Coupling. - La manutenzione e i miglioramenti risultano semplificati.

- Si ha un maggiore riuso di funzionalità a grana fine e altamente correlate, poiché una classe coesa può essere usata per uno scopo molto specifico.

**single responsibility principle**

– mantenere minimo il numero di interfacce implementate da una classe: piu piccolo l’insieme delle interfacce realizzate, piu alta la probabilita che la classe si impegni a soddisfare un’unica responsabilita

– mantenere l’insieme di operazioni definite in una classe coerente con l’insieme di responsabilita da essa assunte

NB: non è specificatamente un GRASP, ma ne è conseguenza diretta.

**Controller**

Problema:

Qual è il primo oggetto oltre lo strato User Interface (UI) che riceve e coordina un’**operazione di sistema**?

Le operazioni di sistema vengono inizialmente esaminate durante l’analisi dei sequence diagram e sono gli eventi di input principali nel sistema. Per esempio, quando uno scrittore che utilizza un word processor preme il pulsante “controllo ortografico”, sta generando un evento di sistema che indica “esegui un controllo ortografico”. Un **controller** è il primo oggetto oltre lo strato UI che è responsabile di ricevere o gestire un messaggio di un’operazione di sistema.

Soluzione:

Assegna la responsabilità a una classe che soddisfa una delle seguenti condizioni:

- Rappresenta un dispositivo all’interno del quale viene eseguito il software, un punto di accesso al software o un sottosistema principale.

- Rappresenta uno scenario di un caso d’uso all’interno del quale si verifica l’evento di sistema, spesso chiamato <UseCaseName>Handler, <UseCaseName>Coordinator o <UseCaseName>Session. In tal caso, si utilizzi la stessa classe controller per tutti gli eventi di sistema nello stesso scenario di caso d’uso. Informalmente, una **sessione** è un’istanza di una conversazione con un attore e spesso corrisponde all’esecuzione di un caso d’uso.

Discussione:

Controller è semplicemente un pattern di delega: in conformità al fatto che lo strato UI non deve contenere logica applicativa, gli oggetti dello strato UI devono delegare le richieste di lavoro a oggetti di un altro strato. Il pattern Controller riassume le scelte fatte comunemente dagli sviluppatori O.O. quando questo “altro strato” è lo strato del dominio, in merito all’oggetto di dominio delegato che riceve le richieste di lavoro e che, appunto, viene chiamato *oggetto controller*.

Gli eventi di sistema sono eventi di input esterni ricevuti da un sistema. Di solito, sono catturati da una GUI utilizzata da una persona. Altri eventi di input comprendono i messaggi esterni, i messaggi remoti sincroni o asincroni, e i segnali provenienti da sensori. In tutti i casi, si deve scegliere un gestore per questi eventi. Il pattern Controller fornisce una guida sulle scelte opportune e generalmente accettate.

Linea guida: normalmente un controller deve delegare ad altri oggetti il lavoro da eseguire durante l’operazione di sistema; infatti, coordina e controlla le attività ma non esegue di per sé molto lavoro.

Un difetto comune nella progettazione dei controller deriva in effetti da un’eccessiva assegnazione di responsabilità. Un controller in questo caso soffre di una coesione bassa, violando il principio High Cohesion. Spesso si tende a usare la stessa classe controller per tutti gli eventi di sistema di un unico caso d’uso, in modo che il controller possegga la responsabilità aggiuntiva di conservare le informazioni sullo stato del caso d’uso. Per casi d’uso diversi possono essere usati controller differenti.

Gli oggetti controller appartengono spesso allo strato del dominio ma, nei sistemi più complessi, possono appartenere a uno strato Application separato, collocato tra lo strato UI e lo strato del dominio.

Oltre a quello del pattern GRASP Controller, al termine “controller” sono stati assegnati anche altri significati. Di seguito ne sono discussi alcuni.

Nel pattern *Boundary-Control-Entity* (BCE) sono definiti i concetti di classi limite, controllo ed entità. Gli **oggetti limite** (*boundary*) sono astrazioni delle interfacce, gli **oggetti entità** (*entity*) sono gli oggetti software del dominio, indipendenti dalla specifica applicazione e normalmente persistenti, mentre gli **oggetti controllo** (*control*) sono gestori dei casi d’uso, come descritto nel pattern Controller. Nel pattern *Model-View-Controller* (MVC), un’applicazione interattiva è divisa in tre tipi di componenti: il **modello**, che contiene i dati e le funzionalità di base dell’applicazione, le **viste**, che mostrano informazioni agli utenti, e il **controller**, che gestiscono le richieste degli utenti. Viste e controller formano l’interfaccia utente.

Le nozioni di controller MVC e di controller GRASP sono distinte come segue:

- Il *controller MVC* fa parte della UI e gestisce l’interazione con l’utente; la sua implementazione dipende in larga misura dalla tecnologia UI e dalla piattaforma utilizzata.

- Il *controller GRASP* fa parte dello strato del dominio e coordina la gestione delle richieste delle operazioni di sistema. Inoltre, non dipende dalla tecnologia UI utilizzata.

Notiamo che la soluzione proposta dal pattern Controller fa riferimento a due categorie di controller: i **facade controller** e i **controller di caso d’uso**.

Un facade controller rappresenta il punto di accesso principale per le chiamate dei servizi dallo strato UI agli strati sottostanti, e potrebbe essere un’astrazione dell’unità fisica complessiva, una classe che rappresenta l’intero sistema software, oppure una classe che rappresenta il punto di accesso all’applicazione. I facade controller sono adatti quando non ci sono troppi eventi di sistema o quando l’interfaccia utente non può reindirizzare i messaggi per gli eventi di sistema a più controller alternativi, come in un sistema per l’elaborazione di messaggi.

Se invece si sceglie un controller di caso d’uso, si avrà un controller diverso per ciascun caso d’uso. Questo tipo di controller non è un oggetto di dominio, bensì un costrutto artificiale per supportare il sistema (una Pure Fabrication, in termini di pattern GRASP). I controller di caso d’uso sono adatti quando la collocazione delle responsabilità in un facade controller porta a progetti con coesione bassa o accoppiamento alto, e lo stesso controller diventa “gonfio” di eccessive responsabilità. Un controller di caso d’uso, dunque, rappresenta una buona scelta quando ci sono molti eventi di sistema in processi diversi: infatti, suddivide la loro gestione in classi separate più gestibili, oltre a fornire una base per conoscere e ragionare sullo stato dello scenario attualmente in esecuzione.

D’altronde, un controller di caso d’uso implementa l’interfaccia di sistema di un solo caso d’uso, mentre un facade controller implementa le interfacce di sistema di tutti i casi d’uso. Una decisione intermedia è scegliere un controller diverso per ciascun attore del sistema software, che gestisce tutti i casi d’uso di quello stesso attore.

Controindicazioni:

Se progettata in modo mediocre, una classe controller può avere una coesione bassa, risultando così un controller gonfio; le sue caratteristiche sono le seguenti:

- È unico nel sistema e riceve tutti gli eventi di sistema, che in genere sono numerosi.

- Svolge molti dei compiti necessari per soddisfare l’evento di sistema, senza delegare il lavoro. Ciò solitamente coinvolge una violazione di Information Expert e High Cohesion.

- Ha numerosi attributi e conserva informazioni significative sul sistema o sul dominio che avrebbero dovuto essere distribuite ad altri oggetti.

Due possibili rimedi sono:

1) Progettare il controller in modo tale che deleghi la soddisfazione della responsabilità di ciascuna operazione di sistema agli altri oggetti.

2) Aggiungere più controller.

Vantaggi:

- **Maggiore potenziale di riuso e interfacce inseribili**: l’applicazione di Controller assicura che la logica applicativa non sia gestita nello strato dell’interfaccia o della presentazione. Ciò favorisce il riuso della logica applicativa in altre applicazioni future e, in generale, in interfacce diverse.

- **Opportunità di ragionare sullo stato del caso d’uso**: a volte è necessario assicurarsi che le operazioni di sistema si susseguano in una sequenza legale, o anche ragionare sullo stato corrente dell’attività nel caso d’uso in corso di esecuzione. Per esempio, potrebbe essere necessario garantire che un’operazione o2 non venga eseguita prima della terminazione dell’operazione o1. Occorre catturare da qualche parte queste informazioni sullo stato del caso d’uso o della sessione; il controller è effettivamente una scelta ragionevole.

**Legge di Demetra**

Nota anche come **Principle of Least Knowledge** (**principio della conoscenza minima**), non è propriamente un pattern GRASP, ma è comunque una linea guida molto valida per lo sviluppo del software orientato agli oggetti. Nella sua forma più generale, può essere descritta nei seguenti termini:

- Ogni unità di programma dovrebbe conoscere solo poche altre unità di programma strettamente correlate.

- Ogni unità di programma dovrebbe interagire solo con le unità che conosce direttamente. Questi due principi possono essere riassunti col motto “non parlate con gli sconosciuti”.

In modo più formale, la legge di Demetra per le funzioni richiede che ogni metodo di un oggetto O possa invocare solo i metodi dei seguenti tipi di oggetti:

- I propri

- Dei suoi parametri

- Di ogni oggetto che crea

- Dei suoi componenti diretti

Viceversa, un oggetto dovrebbe evitare di invocare metodi di un oggetto ritornato da un altro metodo.

Il vantaggio nel seguire la legge di Demetra consiste nel fatto che il software tende a essere più mantenibile e adattabile. Visto che gli oggetti x sono meno dipendenti dalla struttura interna degli altri oggetti, questi ultimi possono essere modificati senza dover ristrutturare gli oggetti x. Ciò comporta anche la diminuzione della probabilità della presenza di bug nel software distribuito. Uno svantaggio della legge è che richiede la scrittura di una grande quantità di metodi wrapper per propagare le chiamate a metodo. Ciò può aumentare il tempo di sviluppo e la quantità di codice necessario e peggiora le performance.

Esempio:

*public class Car {*

*private Driver d;*

*private Vector<Passengers> p;*

*public Car(Context context) {*

*this.d = context.driver;*

*this.p = context.getPassengers();*

*}*

*}*

Questo esempio viola la legge di Demetra: infatti, qui Car deve essere un Information Expert di Context, ovvero deve necessariamente sapere che Context ha un attributo driver e un metodo getPassengers() per inizializzare i suoi attributi. Si verifica perciò un forte coupling tra Car e Context, che rappresentano due concetti semanticamente molto distanti tra loro. Una soluzione più appropriata che rispetta la legge di Demetra è la seguente:

*public class Car {*

*private Driver d;*

*private Vector<Passengers> p;*

*public Car(Driver driver) {*

*this(driver, null);*

*}*

*public Car(Driver driver, Vector<Passengers> listOfPassengers) {*

*this.d = driver;*

*this.p = listOfPassengers; }}*

# Lezione22 – Analisi object oriented

Nella progettazione orientata agli oggetti, per definire una prima versione di class diagram, bisogna anzitutto individuare le classi di analisi. Ma esistono tecniche a supporto di questa attività? Chiaramente sì: RUP, ad esempio, suggerisce agli analisti di distinguere le classi di analisi in tre macrocategorie: **boundary**, **control** ed **entity** (che vengono individuate a partire dallo use case diagram), seguendo così il modello **BCE**.

Ma perché si ha bisogno di modellare aspetti non previsti dal linguaggio come le classi boundary, control ed entity?

- Il dominio di applicazione e il committente hanno bisogno di esprimere concetti generali che non sono nativamente inclusi nel linguaggio di modellazione (i.e. UML).

- Il modellista ha bisogno di rappresentare aspetti legati allo specifico processo di sviluppo adottato, dando differenti interpretazioni a uno stesso elemento del linguaggio in base alle fasi del processo.

- Il modellista vuole raffinare i modelli esistenti in funzione del contesto risolvendo eventuali ambiguità, per una generazione del codice più efficiente ed efficace.

In particolare, per modellare aspetti non direttamente previsti da UML, si utilizzano gli **stereotipi**, che sono il principale meccanismo per la personalizzazione di UML. Uno stereotipo estende il significato di un elemento già esistente del linguaggio (detto *base class*), dando luogo a un nuovo elemento del linguaggio. Attraverso la definizione di uno stereotipo possono essere introdotti:

- Ulteriori interpretazioni per un elemento

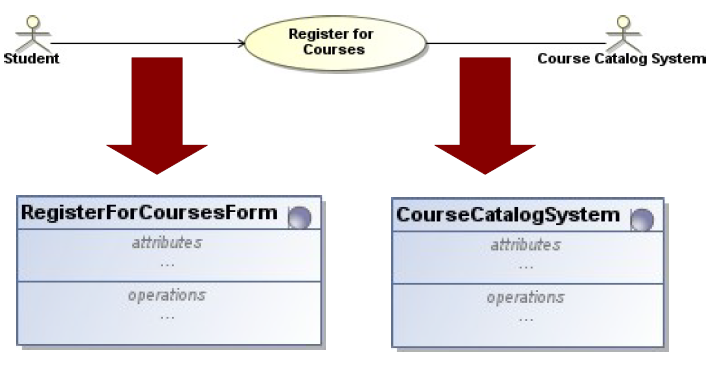
- Ulteriori caratteristiche di un elemento

- Ulteriori relazioni con altri elementi

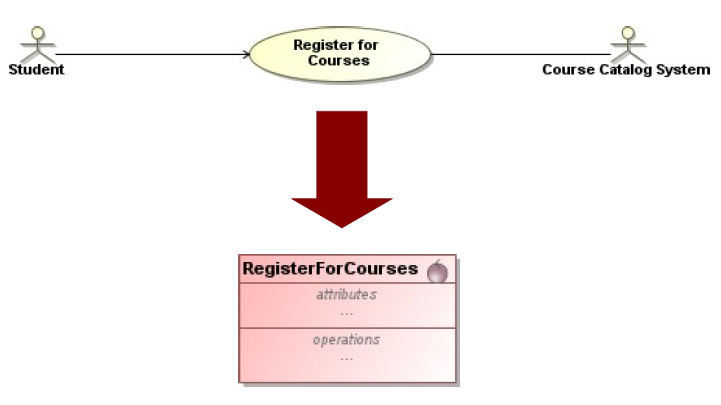
Per applicare il modello BCE, è dunque opportuno definire e marcare le classi di analisi utilizzando gli stereotipi *<<boundary>>*, *<<control>>*, *<<entity>>*.

Analizziamo brevemente queste tre macrocategorie di classi del modello BCE:

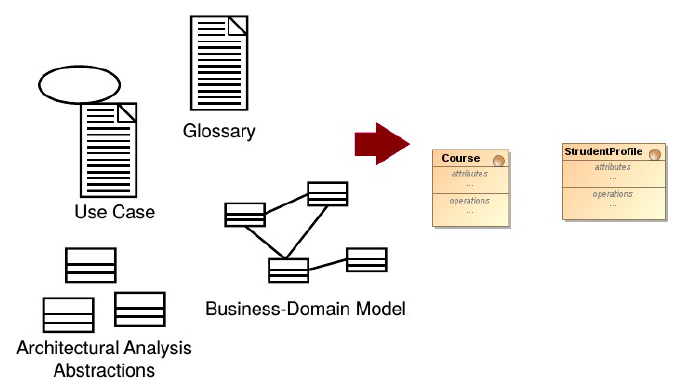
- **Boundary**: mediano l’interazione tra il sistema e l’ambiente, regolano l’interazione con gli attori e rappresentano gli elementi al confine del sistema. Si hanno tante classi boundary quante sono le coppie (attore, caso d’uso) all’interno dello use case diagram.



- **Control**: coordinano il comportamento durante l’esecuzione di un caso d’uso del sistema e rappresentano comportamenti dipendenti dall’interazione con il sistema, pur essendo indipendenti dal modo di attivazione dell’interazione. Si hanno tante classi control quanti sono i casi d’uso all’interno dello use case diagram.



- **Entity**: rappresentano le astrazioni chiave del sistema, modellano il comportamento di un’entità di dominio incapsulando un insieme coeso di dati e sono indipendenti dall’ambiente di esecuzione. In generale, non comprendono gli attori.



In definitiva, il modello BCE dà luogo a un diagramma delle classi di analisi che, solo con raffinamenti successivi, porta alla produzione di un diagramma delle classi di progettazione.

**Diagramma VOPC**

VOPC è l’abbreviazione di **View Of Participating Classes** e descrive le classi che costituiscono un singolo caso d’uso. Generalmente viene utilizzato dopo aver descritto il flusso degli eventi che caratterizzano il relativo caso d’uso.

# Lezione29 – Pattern GoF

**Cos’è un design pattern**

È una descrizione di un problema ricorrente nella progettazione O.O. In particolare, a ogni problema vengono associati:

- Un nome

- Una soluzione che può essere applicata in differenti circostanze anche eterogenee tra loro

- Una discussione sugli effetti e sulle variazioni che conseguono l’applicazione della soluzione

Nella pratica, l’uso e la composizione dei design pattern supportano i modellisti / architetti del software verso la definizione di soluzioni dove sia mitigata l’influenza di fattori umani legati a esperienze personali. Infatti, il loro punto focale è **fomalizzare e strutturare problemi ricorrenti** e il loro scopo non è fornire nuovi spunti alla progettazione, bensì supportare l’applicazione di **tecniche consolidate**.

**Catalogo GoF**

I pattern GoF sono organizzati in un catalogo secondo due criteri di classificazione:

- **Scopo**: criterio di classificazione che definisce il dominio di applicazione dei pattern.

1) **Creational**: pattern che riguardano il processo di creazione di oggetti e rendono il sistema indipendente dalle modalità con cui l’istanziazione avviene.

2) **Structural**: pattern che riguardano aspetti di composizione di classi e oggetti per formare strutture complesse.

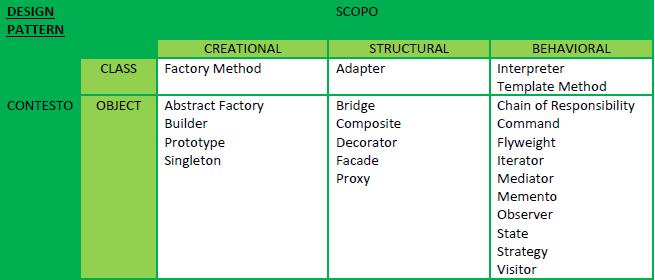
3) **Behavioral**: pattern che riguardano come classi / oggetti interagiscono per raggiungere determinati obiettivi assegnati.

- **Contesto**: criterio di classificazione che definisce la tipologia di elementi cui il pattern può essere applicato.

1) **Class**: pattern che considerano le relazioni tra classi e loro sottoclassi (struttura statica).

2) **Object**: pattern che considerano le relazioni tra oggetti modificabili a runtime (struttura dinamica).

Tabella riassuntiva che rappresenta il catalogo GoF con la classificazione di tutti e 23 i design pattern:



- **Pattern Creational && Class**: delegano parte del processo di creazione di un oggetto a sottoclassi.

- **Pattern Creational && Object**: delegano parte del processo di creazione di un oggetto ad altri oggetti.

- **Pattern Structural && Class**: utilizzano l’erediterietà per comporre classi o implementazioni.

- **Pattern Structural && Object**: descrivono modi per raggruppare oggetti e realizzare così nuove funzionalità.

- **Pattern Behavioral && Class**: utilizzano l’ereditarietà per descrivere algoritmi e flusso di controllo.

- **Pattern Behavioral && Object**: descivono come gruppi di oggetti cooperano per eseguire un compito che un singolo oggetto non potrebbe portare a termine da solo.

**Pattern creazionali**

Forniscono un’astrazione del processo di istanziazione degli oggetti e rendono il sistema indipendente da tale modalità. Quelli basati su classi utilizzano l’ereditarietà per scegliere la particolare classe da istanziare; quelli basati su oggetti delegano l’istanziazione ad un altro oggetto. Rendono il sistema maggiormente flessibile poiché conosce soltanto le interfacce degli oggetti definite mediante classi astratte.

**Factory Method**

Come già accennato mentre descrivevamo il pattern GRASP Creator, esistono dei casi particolari in cui non è facile assegnare la responsabilità di istanziare nuovi oggetti a una classe di dominio già esistente, poiché si rischierebbe di ottenere una soluzione poco elegante: la classe di dominio, infatti, potrebbe avere già diverse responsabilità riguardanti la logica applicativa, e attribuirle anche una responsabilità creazionale sarebbe sintomo di una coesione bassa e, quindi, di una violazione dei principi GRASP High Coehesion nonché Low Coupling. Per esempio, questa situazione si può avere nel momento in cui l’oggetto da istanziare apparterrà a un insieme di classi con caratteristiche simili e la sua creazione va fatta dipendentemente da una proprietà esterna che, appunto, determinerà il tipo esatto di tale oggetto.

Una soluzione consolidata a questo problema consiste nell’applicare il design pattern creazionale Factory Method, in cui viene definito un oggetto “factory” (“fabbrica”) che avrà la responsabilità di creare delle istanze.

- Nome: Factory Method

- Sinonimi: Simple Factory, Concrete Factory, Virtual Constructor

- Problema: chi deve essere responsabile della creazione di oggetti quando ci sono delle considerazioni speciali, come una logica di creazione complessa, quando si desidera separare le responsabilità di creazione per una coesione migliore?

- Soluzione: crea un oggetto Factory che gestisca la situazione.

- Motivazione: si ha bisogno di determinare l’esatto tipo dell’oggetto da istanziare solo a run-time.

Non si tratta di un pattern GoF vero e proprio, bensì di una semplificazione dell’*Abstract Factory* e spesso viene descritto come una sua variante.

Gli oggetti Factory presentano diversi vantaggi:

- Separano le responsabilità delle creazioni complesse in oggetti di supporto coesi.

- Nascondono la logica di creazione potenzialmente complessa.

- Consentono l’introduzione di strategie per la gestione della memoria che possono migliorare le prestazioni, come il caching o il riciclaggio degli oggetti.

Applicabilità:

Nella pratica, il pattern Factory Method è molto utile quando:

- Una classe non è in grado di sapere a priori le classi di oggetti che deve creare e si ha dunque bisogno di determinare l’esatto tipo degli oggetti da istanziare solo a runtime.

- La creazione di un oggetto richiede l’accesso a informazioni o risorse che non dovrebbero essere contenute nelle classi associate / che aggregano / che compongono tale oggetto.

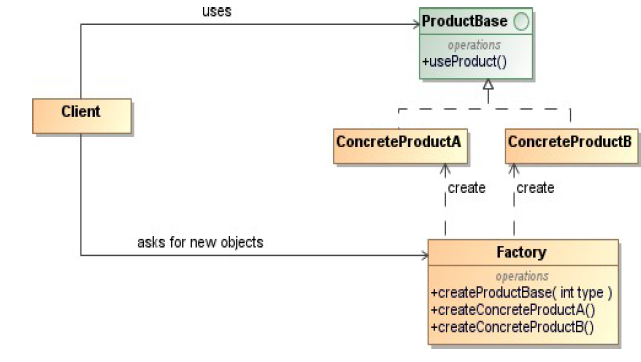
- Il sistema deve essere indipendente dalle modalità di creazione, composizione e rappresentazione dei suoi oggetti.

- Si vuole una libreria (i.e. un insieme di classi) che esponga soltanto l’interfaccia e non la sua implementazione.

- La gestione del ciclo di vita degli oggetti deve essere centralizzata in modo da assicurare un comportamento coerente all’interno dell’applicazione.

- Tutte le classi delegano la responsabilità di creazione.

Struttura:



Partecipanti:

- **Factory**: dichiara e implementa i meccanismi di creazione per un insieme di oggetti simili (ConcreteProductA, ConcreteProductB) che condividono la stessa interfaccia / classe astratta.

- **ProductBase**: dichiara un’interfaccia (o una classe astratta) per una tipologia di oggetti che saranno utilizzati dal Client.

- **ConcreteProductA**, **ConcreteProductB**: definiscono gli oggetti che dovranno essere creati dalla corrispondente factory e implementano / estendono ProductBase.

- **Client**: crea istanze di ConcreteProduct non direttamente, bensì attraverso la Factory, e utilizza soltanto l’interfaccia / la classe astratta ProductBase.

Collaborazioni:

- Durante l’esecuzione è preferibile riferire un’unica istanza per la classe Factory.

- La Factory gestisce la creazione di oggetti simili con un’implementazione specifica.

Conseguenze:

Un grosso vantaggio offerto da questo pattern è l’isolamento delle classi concrete (ConcreteProductA, ConcreteProductB) dal Client, il che migliora la mantenibilità del sistema. In particolare, Factory Method consente sia di cambiare in modo semplice l’implementazione e i comportamenti esposti da una o più classi ConcreteProduct, sia di aggiungere in modo semplice nuove classi concrete ConcreteProduct: in entrambi i casi, l’unica classe che risente di tali modifiche è la Factory, mentre il Client può rimanere intatto (il che rappresenta un grosso vantaggio nel momento in cui i Client che utilizzano il sistema sono molteplici).

Implementazione:

Nella formulazione originale del pattern si propone un unico metodo all’interno della Factory che restituisce un solo tipo di ConcreteProduct, il quale viene stabilito tramite un parametro passato in ingresso all’operazione. Tuttavia, questo aspetto è sconsigliato in quanto tende a essere poco sicuro, e si preferisce dunque la versione proposta nel diagramma mostrato precedentemente, in cui la Factory contiene un metodo diverso per ogni tipo differente che potranno avere gli oggetti da istanziare.

Per il codice di esempio, vedere il materiale allegato alla lezione.

**Abstract Factory**

Generalizziamo il problema descritto precedentemente e supponiamo di voler fornire un’interfaccia per la creazione di intere **famiglie di oggetti** correlati senza specificare quali siano le loro classi concrete.

- Nome: Abstract Factory

- Sinonimi: Kit

- Problema: come creare famiglie di classi correlate che implementano un’interfaccia comune?

- Soluzione: definisci una factory astratta tramite un’interfaccia o una classe astratta; definisci una classe factory concreta per ciascuna famiglia di elementi da creare; opzionalmente, definisci una vera classe astratta che implementi la factory astratta e fornisca servizi comuni alle factory concrete che la estendono.

In tal modo, non c’è bisogno che la classe Client specifichi esplicitamente i nomi delle classi concrete all’interno del proprio codice (come nel pattern Factory Method, è sufficiente che il Client sia associato a un’interfaccia o a una classe astratta). Ciò porta il sistema a essere indipendente dall’implementazione degli oggetti concreti.

Applicabilità:

Nella pratica, questo pattern è utile quando:

- Si vuole un sistema indipendente da come gli oggetti vengono creati, composti e rappresentati.

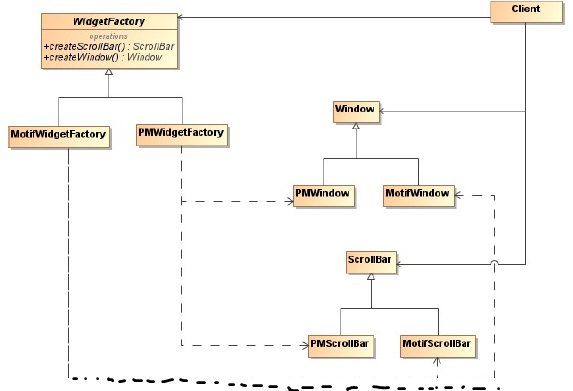
- Si vuole permettere la configurazione del sistema come scelta tra diverse famiglie di oggetti alternative tra loro.

- Si vuole che gli oggetti che sono organizzati in famiglie siano vincolati a essere utilizzati con altri oggetti appartenenti alla medesima famiglia.

- Si vuole che ogni oggetto esponga un insieme condiviso di operazioni indipendentemente dalla famiglia.

- Si vuole fornire una libreria (i.e. un insieme di classi) mostrando solo le interfacce e nascondendo le implementazioni.

Struttura:



Partecipanti:

- **WidgetFactory**: dichiara un’interfaccia (o una classe astratta) per le operazioni di creazione degli oggetti.

- **MotifWidgetFactory**, **PMWidgetFactory**: implementano la creazione degli oggetti concreti.

- **Window**, **ScrollBar**: dichiarano un’interfaccia (o una classe astratta) per una tipologia di oggetti (che, appunto, può essere Window o ScrollBar).

- **MotifWindow**, **MotifScrollBar**, **PMWindow**, **PMScrollBar**: definiscono un oggetto che dovrà essere creato dalla corrispondente factory concreta, e implementano le interfacce / classi astratte Window e ScrollBar.

- **Client**: utilizza soltanto le interfacce dichiarate dalle classi WidgetFactory, Window e ScrollBar.

Collaborazioni:

- In generale si crea una sola istanza di factory concreta a runtime. Questa istanza gestisce la creazione di una sola famiglia di oggetti con un’implementazione specifica. Per creare oggetti di un’altra famiglia bisogna istanziare un’altra factory.

- La factory astratta delega la creazione degli oggetti alle sue sottoclassi factory concrete.

Conseguenze:

Il design pattern Abstract Factory:

- **Isola le classi concrete**: secondo il principio dell’incapsulamento dell’implementazione, la factory viene utilizzata solamente attraverso la sua interfaccia, per cui nei Client non c’è traccia del codice per istanziare gli oggetti. Inoltre, i Client usano gli oggetti concreti attraverso la loro interfaccia comune, in modo che anche il codice successivo all’istanziazione sia indipendente dall’esatta classe che effettivamente implementa il metodo concreto.

- **Consente di cambiare in modo semplice la famiglia di prodotti utilizzata**: la factory viene istanziata una sola volta nel codice, quindi è sufficiente cambiare il tipo di factory istanziato in quel punto del sorgente per utilizzare una diversa famiglia di oggetti. La coerenza col resto del codice è assicurata dall’utilizzo delle interfacce (o classi) astratte e non delle classi concrete secondo il principio di *programmazione verso l’interfaccia e non verso l’implementazione*.

- **Promuove la coerenza nell’utilizzo degli oggetti**: se gli oggetti di una famiglia sono stati esplicitamente progettati per lavorare insieme, la factory astratta permette di rispettare questo vincolo.

- **È tale che è difficile aggiungere supporto per nuove tipologie di oggetti**: dato che la factory astratta definisce tutte le varie tipologie di prodotti che è possibile istanziare, aggiungere un nuovo tipo significa modificare l’interfaccia della factory. La modifica si ripercuote a cascata nelle factory concrete e in tutte le sottoclassi, rendendo laboriosa l’operazione.

Implementazione:

**Creazione dei prodotti**: ogni prodotto viene creato mediante un factory method; ogni famiglia di prodotti è istanziata attraverso una specifica sottoclasse.

Nella formulazione originale del pattern si propone un unico metodo all’interno della factory astratta che restituisce un solo tipo di oggetto. Tuttavia, questa versione è fortemente sconsigliata da un punto di vista logico e non è adatta a Java; il suo uso resta dunque confinato a particolari linguaggi.

Per il codice di esempio, vedere il materiale allegato alla lezione.

**Singleton**

La classe Factory solleva un altro problema nella progettazione: chi crea tale classe, e come vi si accede? Come già accennato, nei pattern Factory Method e Abstract Factory è necessaria una sola istanza della Factory. Inoltre, i metodi di questa classe devono poter essere richiamati da vari punti del codice. C’è dunque un problema di visibilità: come ottenere la visibilità verso questa singola istanza di Factory? Una soluzione consiste nel passare l’istanza Factory come parametro ovunque sia necessario; un’altra soluzione, invece, consiste nell’inizializzare gli oggetti che devono comunicare con la Factory con un suo riferimento permanente. Ciò è possibile ma scomodo; un’alternativa è offerta dal design pattern creazionale Singleton.

- Nome: Singleton

- Problema: è richiesta esattamente una sola istanza di una classe, ovvero un “singleton”. Gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto.

- Soluzione: definisci un metodo statico getInstance() della classe che restituisce l’oggetto singleton.

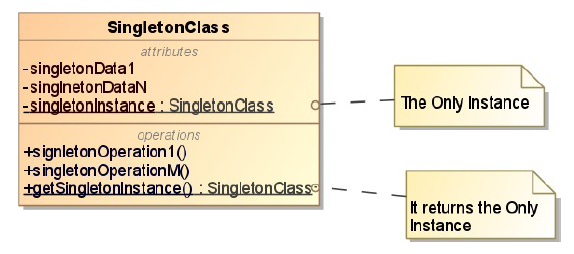
Applicabilità:

Nella pratica, questo pattern è utile quando:

- Deve esistere esattamente un’istanza di una classe e tale istanza deve essere accessibile ai Client attraverso un punto di accesso noto a tutti gli utilizzatori.

- L’unica istanza deve poter essere estesa attraverso la definizione di sottoclassi e i Client devono essere in grado di utilizzare le istanze estese senza dover modificare il proprio codice.

Struttura:



Partecipanti:

- **SingletonClass**: definisce un’operazione getSingletonInstance(), che è un’operazione statica che consente ai Client di accedere all’unica istanza esistente della classe. SingletonClass può essere responsabile della creazione della sua unica istanza.

Collaborazioni:

I Client possono accedere all’istanza di un singleton soltanto attraverso l’operazione getSingletonInstance().

Per garantire che una classe sia un singleton, il metodo getSingletonInstance deve essere l’unico modo fornito dalla classe per accedere al singleton. Per impedire la creazione diretta di oggetti dalla classe, in Java è possibile dichiarare privato il costruttore del singleton.

Conseguenze:

- accesso controllato a un’unica istanza

- riduzione dello spazio dei nomi ovvero non è necessario definire variabili globali

- permette il raffinamento di operazioni e rappresentazione ovvero è possibile definire delle sottoclassi che costituiscono l’unica istanza

- permette di gestire un numero variabile di istanze

- maggiore flessibilità rispetto a operazioni di classe

Implementazione:

**-** assicurare l’esistenza di un’unica istanza:

- costruttore privato

- variabile di classe privata

- metodo di classe che restituisce la variabile di classe

- definizione di sottoclassi di Singleton:

- costruttore protetto

- metodo set oppure utilizzo di meccanismi globali per ottenere l’istanza della sottoclasse

Un’altra domanda comune sull’implementazione di Singleton è: perché non far diventare tutti i servizi dei metodi statici della classe, anziché utilizzare un unico oggetto con dei metodi d’istanza? Un’istanza e un metodo s’istanza sono solitamente preferiti per i seguenti motivi:

- Consentono la ridefinizione nelle sottoclassi e il raffinamento della classe singleton in sottoclassi (in tal caso, è necessario dichiarare il costruttore protetto e non privato); i metodi statici non sono polimorfi e non consentono la ridefinizione nelle sottoclassi.

- La maggior parte dei meccanismi di comunicazione remota orientati agli oggetti supporta l’accesso remoto solo a metodi d’istanza e non a metodi statici.

- Una classe non è sempre un singleton in tutti i contesti applicativi. Per esempio, può esserlo in un’applicazione X ma non in un’applicazione Y.

Per il codice di esempio, vedere il materiale allegato alla lezione.

Attenzione: approfondimenti su PDF (sia riassunto che slide).

# Lezione31 – Altri GoF

**Pattern strutturali**

Descrivono le modalità di composizione di classi e oggetti per formare strutture complesse. Quelli basati su classi utilizzano l’ereditarietà per comporre interfacce o implementazioni; quelli basati su oggetti modellano le modalità di composizione di oggetti per realizzare nuove funzionalità e hanno maggiore flessibilità poiché è possibile cambiare la composizione durante l’esecuzione.

**Adapter**

Supponiamo che il sistema che si sta sviluppando debba supportare diversi tipi di servizi esterni prodotti da terze parti, e che ciascuno di questi servizi abbia una propria API che non può essere modificata. Supponiamo inoltre che l’interfaccia di tali servizi non sia perfettamente compatibile con quella richiesta dal sistema e risulti dunque non riusabile a tutti gli effetti. (e.g. perché non rispecchia esattamente i requisiti o la segnatura richiesta). In casi come questo, invece di riscrivere i servizi da capo (il che risulterebbe oneroso), può essere comodo scrivere un adapter (“adattatore”) che faccia da tramite: si tratta di una soluzione proposta dall’omonimo design pattern strutturale Adapter.

- Nome: Adapter

- Sinonimi: Wrapper

- Problema: come gestire interfacce incompatibili, o fornire un’interfaccia stabile a componenti simili ma con interfacce diverse?

- Soluzione: converti l’interfaccia originale di un componente in un’altra interfaccia, attraverso un oggetto adapter intermedio.

In generale, lo scenario tipico è il seguente: l’adapter riceve dai client delle richieste che, appunto, sono nel “formato del client”. Dopodiché adatta (trasforma) queste richieste nel “formato del server”, per poi inviarle effettivamente al server, il quale fornisce una risposta nel “formato del server”. Poi, allo stesso modo, adatta la risposta ricevuta nel “formato del client” e, infine, la restituisce al client.

Applicabilità:

Nella pratica, questo pattern è utile quando:

- Si vuole utilizzare una classe esistente ma la sua interfaccia non è compatibile con quella che serve, perché magari si vuole fruire di questa interfaccia e dei relativi servizi in una modalità diversa da quella prevista dall’oggetto server.

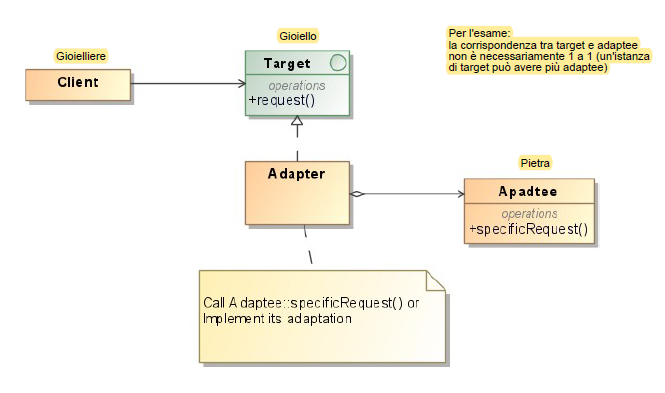
- Si vuole realizzare una classe riusabile che coopera con altre classi, anche se scorrelate e con un’interfaccia eventualmente incompatibile.

- Si hanno più oggetti server che offrono servizi simili; in particolare, questi oggetti hanno interfacce simili ma diverse tra loro, e si vuole fruire dei servizi offerti da uno di loro.

- Si vuole che l’invocazione di un metodo di un oggetto da parte dei Client avvenga solo in maniera indiretta: il metodo “target” viene incapsulato all’interno dell’oggetto, mentre uno o più metodi pubblici fanno da tramite coi Client. Questo consente alla classe di subire modifiche future mantenendo la retrocompatibilità, oppure di implementare in un unico punto una funzionalità alla quale i Client accedono tramite metodi più “comodi”.

Il pattern Adapter può essere basato su classi, utilizzando l’ereditarietà multipla per adattare interfacce diverse con il meccanismo dell’ereditarietà, oppure su oggetti con l’aggregazione, ovvero includendo l’oggetto sorgente nell’implementazione dell’adapter. Noi analizzeremo quest’ultima variante.

Struttura:



Partecipanti:

- **Target**: definisce l’interfaccia di dominio con il client.

- **Client**: coopera con oggetti conformi all’interfaccia target.

- **Adaptee**: definisce l’interfaccia esistente da adattare.

- **Adapter**: realizza il meccanismo di adattamento.

Collaborazioni:

I Client invocano le operazioni su un’istanza di Adapter la quale, a sua volta, gestisce opportunamente le invocazioni verso le istanze di Adaptee.

Notiamo che la classe Adapter aggrega e non estende Adaptee sia per prevenire il fragile base class problem, sia perché l’Adapter potrebbe avere la necessità di utilizzare una o più istanze di Adaptee (in effetti, con una generalizzazione, non sarebbe possibile coinvolgere più istanze diverse di Adaptee).

Nel momento in cui si vuole adattare anche una classe Adaptee2, conviene creare un nuovo Adapter2 che aggrega Adaptee2: aggiungere nuovi costruttori alla vecchia classe Adapter è una soluzione legittima che però potrebbe portare all’assegnazione di troppe responsabilità a un unico Adapter, inficiando sul Low Coupling.

Conseguenze:

- Il pattern adatta Adaptee a Target attraverso la classe concreta Adapter; quest’ultima non è indicata nel momento in cui si vuole gestire l’adattamento di una classe e di tutte le sue sottoclassi.

- L’uso del pattern non è sempre trasparente ai Client.

NB: Affinché la coesione all’interno delle classi della logica applicativa rimanga sufficientemente elevata, è buona norma delegare la responsabilità di creazione degli oggetti di Adapter a una classe Factory apposita, combinando così i design pattern Adapter e Factory (e, volendo, anche Singleton).

Per il codice di esempio, vedere il materiale allegato alla lezione.

**Decorator**

Quando venne introdotta la programmazione orientata agli oggetti, l’ereditarietà era il modello principale utilizzato per estendere la funzionalità di un oggetto. Non sempre però è l’approccio più giusto: infatti, è stato dimostrato che estendere gli oggetti usando l’ereditarietà si traduce spesso in una gerarchia di classi che esplode (**fenomeno dell’Exploding class hierarchy**). Ciò avviene in particolar modo nel momento in cui si hanno a disposizione molte funzionalità che possono essere combinate tra loro in qualunque modo, dato che il numero di comportamenti possibili degli oggetti varia con legge combinatoriale rispetto al numero di funzionalità disponibili. Il design patter strutturale Decorator fornisce un’alternativa flessibile all’erediterietà per estendere la funzionalità degli oggetti.

Nota: nasce per le GUI, ma può essere usato anche per le entity.

- Nome: Decorator

- Sinonimi: Wrapper

- Problema: come gestire dinamicamente le funzionalità e/o le caratteristiche di un oggetto?

- Soluzione: introduci delle classi decorator, che abbiano un’interfaccia conforme all’oggetto, in modo tale che possano “avvolgere” l’oggetto stesso, arricchendolo di ulteriori funzionalità / caratteristiche.

Tale pattern consente di arricchire dinamicamente (a runtime) un oggetto con nuove funzionalità in un modo trasparente al Client, semplicemente collegando i decoratori all’oggetto. Per realizzare tale meccanismo, al costruttore del decoratore si passa come parametro l’oggetto originale (o un differente decoratore). In questo modo, più decoratori possono essere concatenati l’uno all’altro, aggiungendo in modo incrementale funzionalità alla classe concreta, la quale è rappresentata dall’ultimo anello della catena.

Applicabilità:

Nella pratica, questo pattern è utile quando:

- Si vuole poter aggiungere responsabilità a singoli oggetti dinamicamente e in modo trasparente, senza coinvolgere altri oggetti.

- Si vuole poter togliere dinamicamente responsabilità agli oggetti.

- Si vuole che le implementazioni concrete degli oggetti siano disaccoppiate da responsabilità e comportamenti.

- L’estensione diretta attraverso la definizione di sottoclassi non è praticabile, per esempio a causa dell’esplosione del numero di sottoclassi necessarie.

Un caso noto del pattern Decorator è quello di alcune classi core di Java relative alle opereazioni di I/O (che approfondiremo successivamente):

- java.io.BufferedReader;

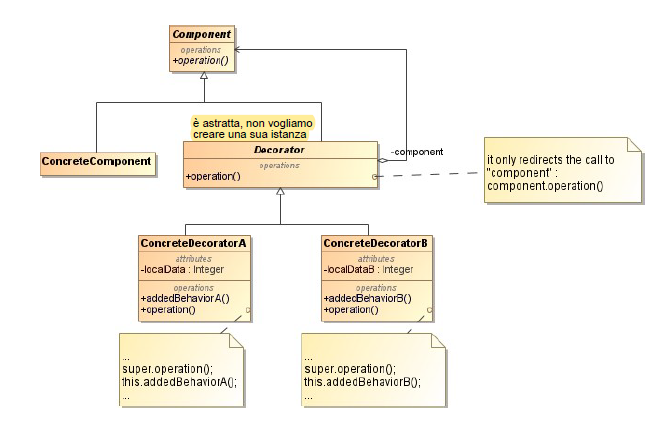
- java.io.BufferedWriter;

- java.io.FileReader;

- java.io.Reader;

E così via.

Struttura:



Partecipanti:

- **Component**: definisce l’interfaccia comune per gli oggetti ai quali possono essere aggiunte responsabilità dinamicamente.

- **ConcreteComponent**: definisce un oggetto al quale possono essere aggiunte responsabilità dinamicamente.

- **Decorator**: mantiene un riferimento a un oggetto di Component e definisce un’interfaccia conforme all’interfaccia di Component.

- **ConcreteDecoratorA**, **ConcreteDecoratorB**: hanno la sola responsabilità di aggiungere funzionalità ai ConcreteComponent.

Collaborazioni:

Un Decorator trasferisce le richieste al suo oggetto Component che può svolgere le eventuali operazioni precedenti e successive alla spedizione delle richieste. in questo modo si ottiene una maggiore flessibilità, con tanti piccoli oggetti al posto di uno molto complicato, andando a modificare solo il contorno e non la sostanza del Component.

Conseguenze:

Come già accennato, il pattern Decorator, rispetto all’approccio basato sull’ereditarietà, presenta diversi vantaggi:

- Un Decorator agisce a runtime a differenza dell’ereditarietà che estende con le sottoclassi il comportamento della classe Component in fase di compilazione.

- Un Decorator può operare su qualsiasi implementazione di una determinata interfaccia, eliminando la necessità di creare sottoclassi di un’intera gerarchia di classi.

- L’aggiunta di funzionalità al Component per mezzo di una sottoclasse interessa tutte le istanze di Component; il pattern Decorator, invece, è in grado di fornire nuovi comportamenti per i singoli oggetti.

- L’uso del Decorator porta a codice pulito e testabile. D’altra parte, i servizi creati con l’ereditarietà non possono essere testati separatamente dalla classe parent perché non esiste un meccanismo per sostituire una classe parent con uno stub.

|  |
| --- |
| *Uno stub consiste in una porzione di codice dummy che sostituisce una funzionalità software vera e propria che potrebbe non essere ancora disponibile. È molto utile nel momento in cui si vuole testare solo le parti restanti del software.* |

Tuttavia, Decorator presenta anche degli svantaggi che potrebbero scoraggiare l’utilizzo del pattern in situazioni per cui non è stato progettato:

- Tutti i metodi dell’interfaccia decorator devono essere implementati nella classe decorator concreta. In effetti, i metodi che non forniscono alcun comportamento aggiuntivo devono essere implementati come metodi di inoltro per mantenere il comportamento esistente. Al contrario, l’ereditarietà richiede solo sottoclassi per implementare metodi che modificano o estendono il comportamento della classe base.

- L’uso dei decorator può complicare il processo di creazione dell’istanza del componente perché, oltre istanziare il componente stesso, è necessario avvolgerlo in un certo numero di decorator.

Per il codice di esempio, vedere il materiale allegato alla lezione.

**Facade**

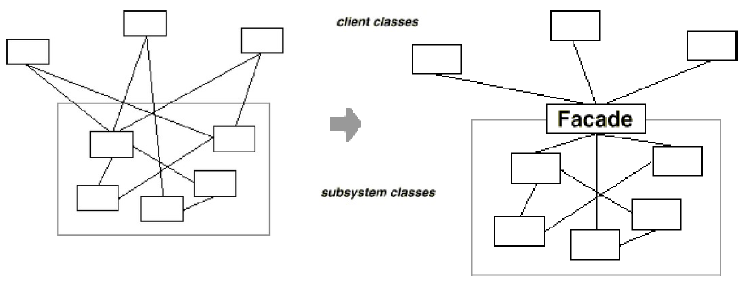
Supponiamo che l’architetto del software, nel progettare un particolare componente software del sistema, non sia sicuro di quale sia la sua implementazione migliore, e che quindi voglia sperimentare diverse soluzioni alternative. Potrebbe sorgere il problema per cui ogni modifica di tale componente software comporti una conseguente modifica nei Client. Per mitigare questo problema e, più in generale, ogni volta che si vuole disaccoppiare per quanto possibile i Client da una porzione del sistema (=da un sottosistema), è possibile ricorrere al design pattern strutturale Facade.

- Nome: Facade

- Problema: è richiesta un’interfaccia comune, unificata e semplificata per un insieme disparato di implementazioni o interfacce, come per definire un sottosistema. Può verificarsi un accoppiamento indesiderato a molti oggetti del sottosistema, o comunque l’implementazione del sottosistema può cambiare (mantenendo comunque invariate le funzionalità offerte). Che cosa fare?

- Soluzione: definisci un punto di contatto singolo con il sottosistema, ovvero un oggetto facade (facciata) che copre il sottosistema. Questo oggetto facade presenta un’interfaccia singola e unificata ed è responsabile della collaborazione coi componenti del sottosistema.

Una Facade è un oggetto “front-end” che rappresenta il punto di entrata singolo ai servizi di un sottosistema; l’implementazione e gli altri componenti del sottosistema sono privati e non possono essere visti dai componenti esterni. Talvolta, il sottosistema potrebbe contenere decine o centinaia di classi, o addirittura una soluzione non orientata agli oggetti; tuttavia, con l’utilizzo della Facade, i Client sono in grado di vedere l’unico punto di accesso pubblico e nient’altro.



Applicabilità:

Nella pratica, questo pattern è utile quando:

- Si vuole fornire un’interfaccia semplificata a un sistema complesso e/o che evolve nel tempo.

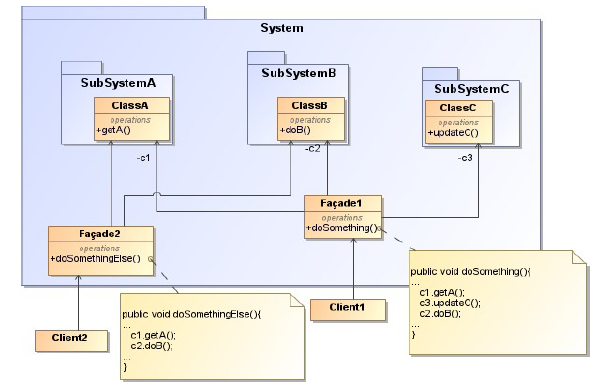
- Si vuole aumentare il grado di riuso di un sottosistema.

- Si vuole stratificare un sistema identificando i punti di accesso a ogni sottosistema.

- Ci sono molte dipendenze tra l’implementazione del sistema e i suoi Client.

- Un Client, per realizzare una singola operazione logica, deve accedere a più classi del sottosistema molto differenti tra loro.

Struttura:



Partecipanti:

- **System**: definisce un sistema del quale si vuole nascondere i dettagli implementativi all’esterno.

- **SubSystemA**, **SubSystemB**, **SubSystemC**: definiscono eventuali sottomoduli del sistema.

- **Facade1**, **Facade2**: definiscono l’interfaccia comune per l’accesso alle funzionalità del sistema; possono implementare logiche composizionali per esportare funzionalità del sistema.

- **Client1**, **Client2**: sono gli utilizzatori delle funzionalità del sistema.

Collaborazioni:

I Client comunicano con il sistema attraverso l’interfaccia comune esportata (la Facade) e non hanno in alcun modo accesso agli oggetti dei sottosistemi.

Conseguenze:

L’utilizzo del pattern Facade:

- Riduce il numero di oggetti che un Client deve gestire direttamente.

- Rende il sistema più riusabile.

- Riduce il grado di accoppiamento tra un sistema e i suoi Client, mitigando la ripercussione delle modifiche sul sistema anche sui Client.

La strutturazione in livelli consente di progettare sistemi indipendenti, con un basso tasso di dipendenze circolari.

Per il codice di esempio, vedere il materiale allegato alla lezione.

**Pattern comportamentali**

Definiscono i modi di comunicazione tra le classi e gli oggetti coinvolti nel pattern.

**Observer**

Supponiamo di avere a disposizione delle classi view che contengono un foglio elettronico, un istogramma e un diagramma a torta, e supponiamo di desiderare che tali classi view si aggiornino ogni volta che cambiano i dati che esse rappresentano (possiamo ipotizzare che i dati siano raccolti all’interno di una classe entity). Una prima possibile soluzione (di tipo *pull*) è la seguente: quando i dati cambiano, la GUI chiede alla classe entity (direttamente o tramite il controller) i dati aggiornati per poter visualizzarli correttamente. Il problema è che la GUI dovrebbe sapere quando i dati variano, il che richiederebbe la duplicazione di una parte significativa della logica applicativa anche nello strato user interface. Una seconda soluzione (di tipo *polling*) è la seguente: la GUI chiede periodicamente alla entity il valore dei dati e, quando questi cambiano, si aggiorna di conseguenza. Spesso, però, le soluzioni basate sul polling sono poco efficienti, soprattutto se i dati da visualizzare sono molti. Perché allora non usare una soluzione di tipo *push*? Quando la entity modifica il valore dei suoi dati, la sua istanza invia un messaggio alle view, chiedendo loro di aggiornare la visualizzazione di questi dati. Tuttavia, il **principio di Separazione Modello-Vista** (secondo cui gli oggetti “modello” non devono conoscere direttamente gli oggetti “vista”, e viceversa) sembra scoraggiare una soluzione di tipo push. In realtà, non proibisce una conoscenza o collaborazione *indiretta* tra gli oggetti “modello” e gli oggetti “vista”.

Questa è una delle idee alla base del design pattern comportamentale Observer, che può essere utilizzato per risolvere il problema di progettazione sopra citato.

- Nome: Observer

- Sinonimi: Dependents, Publish-Subscribe, Delegation Event Model

|  |
| --- |
| *l nome Delegation Event Model è dovuto al fatto che il publisher delega la gestione degli eventi (i.e. cambi di stato) ai subscriber tramite la notificazione.* |

- Problema: diversi tipi di oggetti ***subscriber*** (abbonati) sono interessati ai cambiamenti di stato o agli eventi di un oggetto ***publisher*** (editore). Ciascun subscriber vuole reagire in un suo modo proprio quando il publisher genera un evento. Inotre, il publisher vuole mantenere un accoppiamento basso verso i subscriber. Che cosa fare?

- Soluzione: definisci un’interfaccia subscriber o listener (ascoltatore). Gli utenti subscriber implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.

Questo pattern permette la cooperazione tra l’entity e le view mantenendo tutte queste classi *indipendentemente riusabili*. Infatti, l’entity (=il publisher) può comunicare con le view (=i subscriber) soltanto mediante l’interfaccia subscriber (o listener).

Applicabilità:

Nella pratica, questo pattern è utile quando:

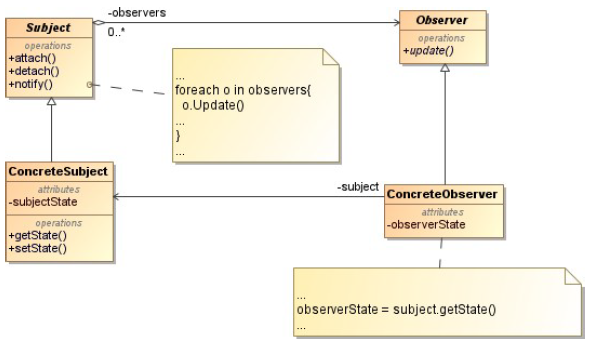
- Un’unica entità presenta di fatto due o più aspetti dipendenti tra loro. Incapsulando questi aspetti in classi separate, è possibile riusarli in modo indipendente.

- Si vuole gestire il fatto che una modifica a un oggetto richiede modifiche anche agli oggetti che dipendono da esso, anche se non si conosce il numero degli oggetti dipendenti.

- L’oggetto modificato ha bisogno di notificare gli altri oggetti senza conoscerne l’identità precisa. In altre parole, si vuole mantenere un alto livello di disaccoppiamento.

In generale, si tratta di un pattern molto utilizzato: viene impiegato in molte librerie, nei tool per le GUI, nel pattern MVC (che analizzeremo successivamente) e nei sistemi di messaggistica e realtime.

Struttura:



Partecipanti e collaborazioni:

- **Subject**: è una classe astratta che fornisce interfacce per registrare o rimuovere dinamicamente gli observer (=subscriber), e implementa le seguenti funzioni:

- attach(observer): aggiunge un ConcreteObserver alla lista delle classi da notificare.

- detach(observer): rimuove un ConcreteObserver dalla lista delle classi da notificare.

- notify(): notifica un cambiamento alle classi ConcreteObserver.

Il metodo notify() viene implementato con un loop su tutti i ConcreteObserver, dove ciascuno di essi chiama conseguentemente la funzione update(); notify() viene eseguita dal ConcreteSubject (=publisher) per notificare un cambio del suo stato (ovvero una variazione del valore assunto dall’attributo subjectState). Poiché ConcreteSubject deve essere l’unica classe ad avere la responsabilità di notificare un cambio di stato tramite il metodo notify(), è buona norma rendere non visibile tale metodo, settandolo come privato all’interno della classe Subject.

Notiamo inoltre che i metodi attach(), detach() e notify() possono essere tutti implementati all’interno della classe parent Subject, poiché il loro funzionamento non dipende dalla classe ConcreteSubject specifica. Ciononostante, manteniamo la dichiarazione di Subject come classe astratta per indicare che non vogliamo istanziare alcun oggetto di Subject, bensì solo oggetti di ConcreteSubject.

- **ConcreteSubject**: contiene l’attributo subjectState che ne descrive lo stato. Implementa anche i seguenti metodi aggiuntivi:

- getState(): restituisce lo stato del publisher.

- setState(): setta lo stato del publisher.

L’utilità di queste due funzioni diventa evidente quando si presenta il seguente scenario: un’istanza di ConcreteObserver (un subscriber) cambia stato (per esempio a seguito dell’invocazione di un metodo apposito); affinché tutti gli altri subscriber rimangano coerenti con tale modifica, devono essere notificati. Ciò avviene perché, contestualmente al cambio di stato, l’istanza modificata invoca il metodo setState() del ConcreteSubject, il quale a sua volta, oltre a settare il valore dell’attributo subjectState, chiama il metodo notify(), che porta dunque alla modifica di tutte le istanze ConcreteObserver della lista. Tali istanze, per cambiare stato in modo corretto e consistente, inviano un messaggio a ConcreteSubject con una getState(), che restituisce loro lo stato a cui bisogna omologarsi. Per realizzare questo meccanismo, è necessaria dunque una relazione di associazione che va dalla classe ConcreteObserver alla classe ConcreteSubject.

- **Observer**: definisce un’interfaccia per tutti i subscriber per ricevere le notifiche dal publisher. È utilizzata come classe astratta per implementare i ConcreteObserver. Contiene l’operazione astratta update(), che dovrà essere realizzata dai ConcreteObserver.

- **ConcreteObserver**: mantiene un riferimento alla classe ConcreteSubject per riceverne lo stato ogni volta che avviene una notifica; ha anche l’attributo observerState che tiene traccia del proprio stato corrente.

NB: Il pattern così definito non richiede che almeno un oggetto subscriber sia registrato col publisher: la lista di oggetti di ConcreteObserver può anche essere vuota.

In definitiva, si tratta di un pattern molto flessibile; infatti:

- È in grado di supportare diversi tipi di eventi all’interno dell’applicazione: basta definire un’interfaccia listener (=una classe astratta Observer) per ogni tipo di evento.

- Per ciascun tipo di evento (=per ogni classe astratta Observer) possono esserci diversi publisher (=ConcreteSubject).

- Ogni publisher può essere l’editore di più tipi di eventi.

- Ogni publisher può avere zero o più subscriber.

- Ogni subscriber può essere registrato a zero o più publisher, anche con riferimento a diversi tipi di eventi.

- Le relazioni tra publisher e subscriber possono cambiare dinamicamente.

Conseguenze:

- È possibile variare publisher e subscriber(soggetti e osservatori) in modo indipendente; inoltre, è possibile riusare i publisher senza riusare i loro osservatori e viceversa.

- Si ha un accoppiamento astratto e minimale fra Subject e Observer: i Subject conoscono una lista di Observer conformi a una specifica interfaccia, mentre non conoscono affatto alcuna classe ConcreteObserver.

- Le comunicazioni “broadcast” sono facilitate: le notifiche non sono inoltrate a un solo specifico destinatario, bensì a tutti gli Observer interessati che si sono registrati come subscriber; il Subject non si occupa di quanti Observer sono presenti ma la sua unica responsabilità è inoltrare la notifica a tutti loro; inoltre, è possibile aggiungere o togliere dinamicamente osservatori in qualunque momento; infine, ogni ConcreteObserver può decidere in modo del tutto indipendenete se e come reagire a ogni notifica ricevuta.

- Gli Observer sono tra loro indipendenti e possono ignorare gli effetti di una richiesta di modifica sul Subject. Vi è quindi necessità di protocolli aggiuntivi che aiutino gli Observer a capire/gestire il cambiamento nel Subject.

Per il codice di esempio, vedere il materiale allegato alla lezione.

**NB**: negli appunti vi è anche il pattern State (pag. 53), non presente nelle slide!

# Lezione34-35 – MVC, MVP

Abbiamo analizzato il BCE, che è un pattern di analisi per i sistemi object-oriented. Sappiamo che il suo scopo principale è identificare le macro-classi del sistema, che derivano dall’osservazione del dominio e dai modi in cui il sistema è utilizzato dagli attori (ovvero dai casi d’uso).

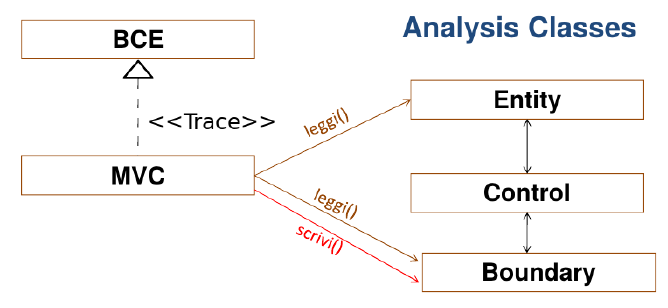
Ora si pone una questione: come si fa a raffinare un modello di analisi BCE per ottenere un diagramma delle classi che sia più fedele a quella che sarà l’implementazione vera e propria? In altre parole: come si introducono e gestiscono aspetti nel dominio della soluzione che tengano conto dell’ingegnerizzazione del sistema (e.g. riuso, ottimizzazione delle soluzioni, mantenibilità, scalabilità)?

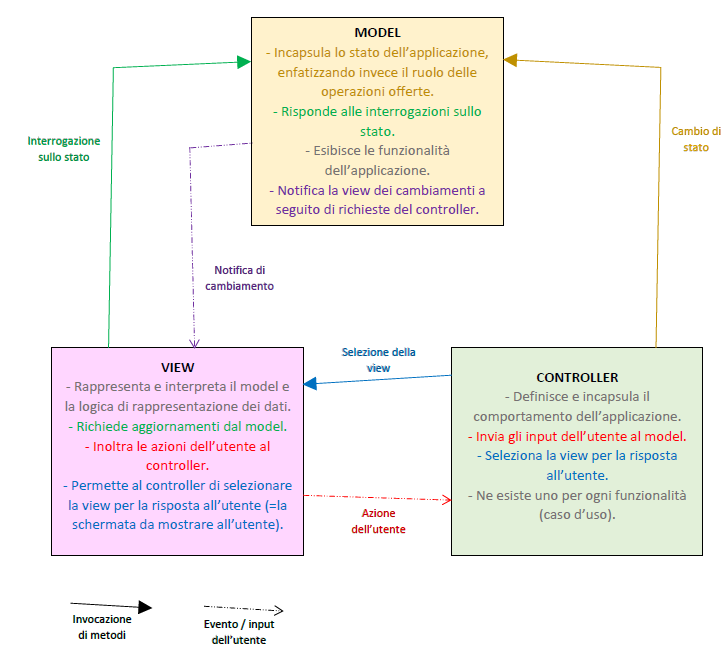
Risposta: il pattern di analisi BCE può essere raffinato secondo diverse varianti, tra cui le più importanti sono:

- **Model-View-Controller** (**MVC**)

- **Model-View-Presenter** (**MVP**)

**MVC**





View:

È responsabile per:

- La logica di presentazione dei dati

- La costruzione e la gestione dell’interfaccia grafica (GUI)

- L’acquisizione dei dati dell’applicazione

- L’accesso ai dati in sola lettura direttamente al model

Inoltre, se non ne è responsabile il controller, può gestire la conversione dei dati dal formato esterno (=come appaiono all’utente) al formato interno (=come sono memorizzati nel model) e viceversa. Es: “12 ottobre 1942” → {1942; 10; 12}

La view ha anche il compito di mantenere aggiornati i dati presentati. Per far ciò, esistono due possibili modalità:

- **Push model**: consiste nell’applicazione del pattern GoF Observer, in cui le view possono registrarsi come osservatori del model e ricevono aggiornamenti dal model “in tempo reale”.

- **Pull model**: viene utilizzato nel caso in cui le view intendono richiedere gli aggiornamenti solo in determinati istanti di tempo e non in tempo reale.

Controller:

È suddiviso in due parti:

**Controller grafico**

- Coordina l’interazione tra view e model per la realizzazione di una funzionalità.

- Realizza il mapping dell’input dell’utente sui processi eseguiti dal model e sulle risposte dell’applicazione.

- Se non ne è responsabile la view, converte i dati dal formato esterno al formato interno e viceversa.

- Crea e seleziona le istanze di view richieste.

**Controller applicativo**

- Implementa la logica di controllo dell’applicazione.

- Raffina il concetto di Controller in BCE.

- Ha la responsabilità di *gestire* un’azione dell’utente sulla view in una o più azioni eseguite sulle istanze nel model (tramite l’invocazione di appositi metodi).

- Per quanto possibile, dovrebbe avere un comportamento stateless.

Esistono due modi per organizzare il controller:

- Controller grafico e controller applicativo aggregati in un unico “bundle”, dove il controller grafico implementa direttamente la logica di controllo. In tal caso, la Boundary del modello BCE si traduce semplicemente in una view nel modello MVC.

- Controller grafico e controller applicativo disaccoppiati, dove il controller grafico invoca operazioni sul controller applicativo. In quest’altro caso, la Boundary del modello BCE si traduce nell’unione tra view e controller grafico nel modello MVC.

**NB**: quest’ultima soluzione è la preferita per il corso (per l’esame, argomentare il perché) -> se la view evolve, l’impatto dei cambiamenti si ripercuotono solo sul controller grafico e non sul modo in cui il caso d’uso è realizzato (i.e. controller applicativo).

Model:

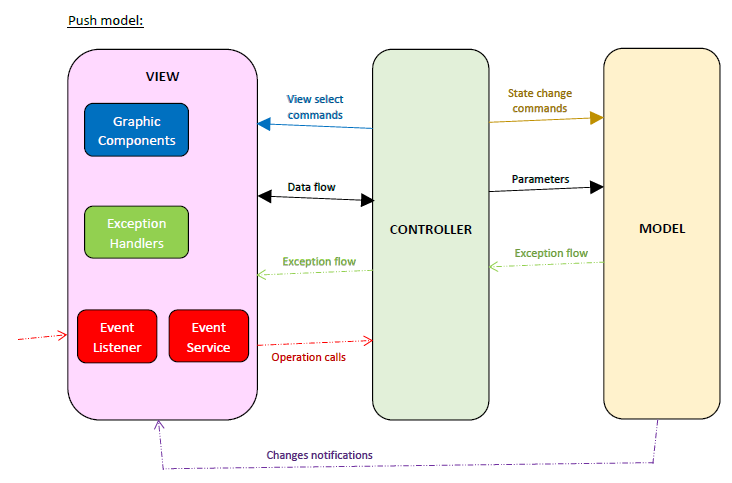
Costituisce la rete delle entità partecipanti alla logica dell’applicazione (ovvero la traduzione in software del dominio reale e del dominio di business del sistema). Per ogni entità:

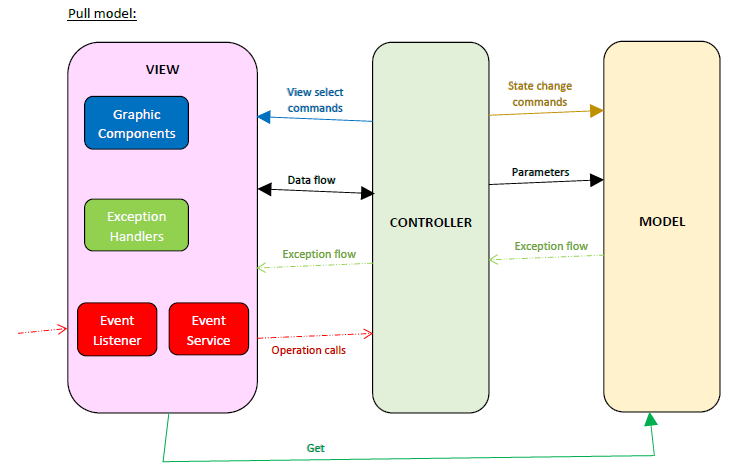
- Descrive i comportamenti esposti in forma di operazioni.

- Definisce le regole di business per l’interazione coi dati.

- Realizza e gestisce le relazioni con le altre entità.

Vediamo ora graficamente il funzionamento del push model e del pull model per quanto riguarda l’aggiornamento delle view.





Bean:

Supponiamo di avere all’interno del model del sistema una classe Date (una key abstraction) con tre attributi: giorno, mese e anno. Supponiamo inoltre di voler modificare solo internamente al sistema (ovvero solo nel model) la rappresentazione della data, portandola a un semplice numero intero. Sicuramente il model risente di questo cambiamento perché la data non è più una key abstraction del sistema. Tuttavia, per come abbiamo definito il modello MVC, la modifica si riversa anche all’interno della view che, quando serve, deve visualizzare la data. Ciò si verifica a causa di un alto accoppiamento tra la view e il model, ovvero tra il modo in cui i dati vengono rappresentati e il modo in cui vengono memorizzati (notiamo infatti che all’interno degli schemi precedenti si ha sempre qualche interazione tra le due macro-classi). Il problema viene accentuato dal fatto che spesso, al momento dello sviluppo, non si riesce a prevedere in che modo e con quale tecnologia gli utenti interagiranno col sistema (da desktop, da browser, su dispositivi mobili). Appare quindi chiaro il bisogno di un’architettura che permetta la separazione netta tra la view e il model.

Per questo motivo, introduciamo un particolare tipo di classe (**bean**) che si interpone principalmente tra le classi view e il model. Le sue principali responsabilità sono le seguenti:

- Disaccoppiamento tra le Boundary e le Entity

- Controllo della validazione sintattica sull’inserimento dei dati in input

- Eventuale gestione dei riferimenti per i dati in visualizzazione (che, come abbiamo già visto, può essere una gestione push-mode o pull-mode).

In genere, una classe bean presenta le seguenti caratteristiche:

- Realizza un POJO (Plain Old Java Object).

|  |
| --- |
| *Un POJO è un oggetto Java ordinario (che quindi, non ha caratteristiche speciali). In particolare:*  *- Non estende classi prespecificate (e.g. “extends javax.servlet.http.HttpServlet”).*  *- Non implementa classi prespecificate (e.g. “implements javax.ejb.EntityBean”).*  *- Non contiene delle annotazioni prespecificate (e.g. “@javax.persistence.Entity public class”).* |

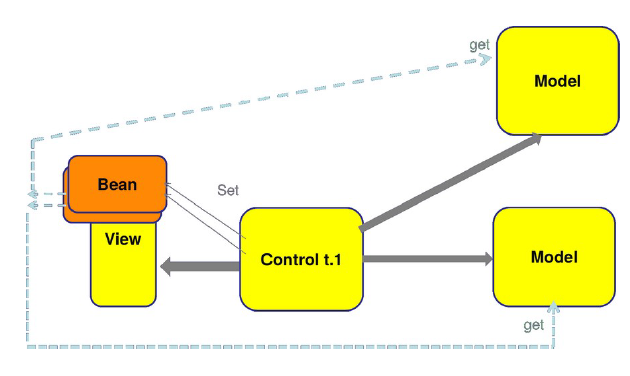
- I suoi attributi (privati) sono destinati a contenere i dati di I/O.

- I suoi metodi pubblici sono principalmente di tipo getter e setter.

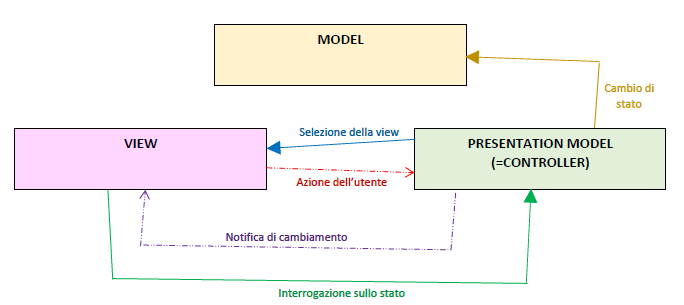
- Accesso agli attributi solo attraverso IO setter e getter.

- Può contenere dei metodi privati per il controllo sintattico dei dati.

La seguente immagine mostra il pull-model con beans:



**MVP**



View:

A differenza della view nel modello MVC: - Non può acquisire i dati da presentare direttamente dal model. - Deve sempre interagire col controller, anche per invocare i metodi getter e setter propri del model.

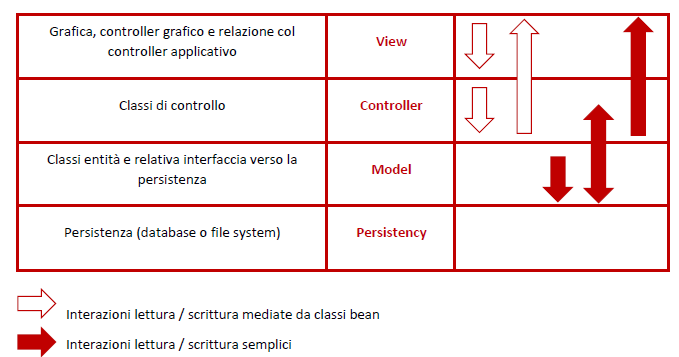
Per il resto, le sue responsabilità sono pari a quelle della view di MVC. In particolare, anche nel modello MVP i dati scambiati tra la parte view e la parte presentation model vanno intesi come classi bean di appoggio (altrimenti si ricreerebbe un accoppiamento indesiderato tra la view e il model).

Il presentation model e il model sono del tutto analoghi ai rispettivi controller e model del pattern MVC.

**MVC in architetture stand-alone**

Tutti e tre gli strati (view, controller, model) risiedono sulla macchina dell’utente (lato client).

Struttura tipica:



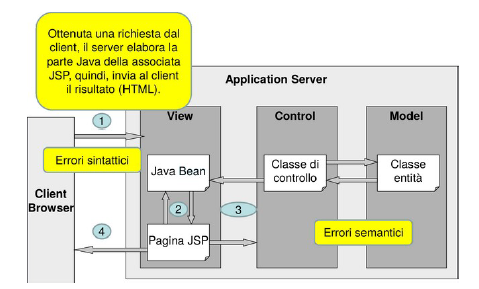
**NB**: per approfondimenti consultare le slide! (da pag.24 a 26)

**MVC in architetture web**

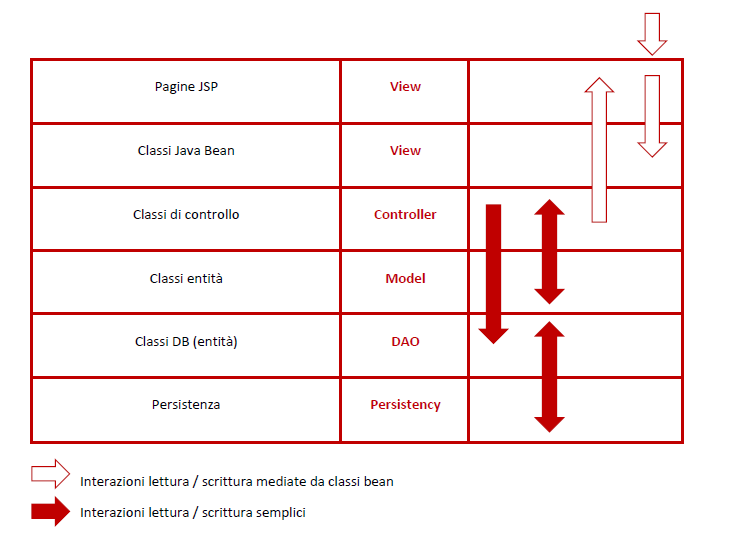
Le applicazioni in un contesto web sono intese come nativamente distribuite e che forniscono funzionalità accessibili da remoto utilizzando **thin-clients** (che sono client a cui vengono fornite meno funzionalità possibili, ponendo tutto il resto sul lato server). La view non è istanziata sulle macchine “utente”, ed è necessario introdurre delle classi di appoggio per la gestione di I/O (delle classi bean, appunto). Nella pratica, di norma, la view è realizzata con la tecnologia **JSP** (**Java Server Pages**), che è una tecnologia di programmazione web in Java che fornisce contenuti dinamici in formato HTML / XML.

D’altra parte, il controller e il model risiedono server-side; in particolare, al contrario delle architetture stand-alone, il controller non è responsabile dell’identificazione della view da presentare all’utente come risposta della sua interazione. Per questo motivo, si introducono le **servlet**, che costituiscono uno strumento server-side che ha la responsabilità di istanziare il controller applicativo e, se non se ne occupa la tecnologia JSP stessa, ha anche il compito di selezionare la view successiva da mostrare all’utente.

In pratica, una servlet è un oggetto scritto in Java che opera all’interno di un server web (e.g. Tomcat) e che processa le richieste HTTP provenienti dal client; viene spesso usata per la generazione di pagine web dinamiche a seconda dei parametri di richiesta inviati dal client browser dell’utente verso il server.



Struttura tipica:



# Lezione38-39 – Persistenza

**Input / output** Con input / output (I/O) si intendono tutte le interfacce informatiche messe a disposizione da un sistema operativo ai programmi per effettuare uno scambio di dati o segnali. In particolare, tali dispositivi (o dimensioni):

- Sono **eterogenei**: si hanno sia molteplici dispositivi di input (e.g. mouse, tastiera, scanner, microfono), sia molteplici dispositivi di output (e.g. monitor, stampante, casse audio).

- Possono dunque supportare **più formati di dato** (e.g. testo, binario, audio, video).

- Possono essere caratterizzati da **più modalità di accesso** (e.g. accesso sequenziale, accesso casuale, accesso bufferizzato).

- Possono avere **più modalità di interazione** (e.g. bit, byte, word, linee, blob).

Java gestisce la varietà e la complessità delle dimensioni di I/O attraverso una vasta gamma di librerie e classi incluse nella piattaforma. Ogni classe incapsula le operazioni necessarie per gestire al meglio:

- Un tipo di dato

- Una tipologia di dispositivi

- Una modalità di accesso

Per trattare tutte le dimensioni della gestione di I/O, è possibile comporre tra loro oggetti afferenti a classi diverse. È possibile inoltre estendere e aumentare il comportamento di queste librerie della piattaforma Java specializzando a piacimento le loro classi.

L’intera gestione dell’I/O in Java è basata sul concetto di **stream** (flusso), che è un canale di comunicazione (un “tubo”) che modella un flusso di informazione tra un programma / dispositivo di input (capace di ricevere dati) e un programma / dispositivo di output (capace di produrre dati). Uno stream è astratto rispetto ai dettagli con i quali vengono effettivamente gestiti i dati dai dispositivi di I/O veri e propri. In generale lo stream è un canale **unidirezionale** e **sequenziale**: permette di leggere solo un byte per volta e, per arrivare a leggere un certo byte B, è necessario prima leggere tutti i byte precedenti all’interno del “tubo”. L’unico elemento che può prescindere dalla sequenzialità dello stream è il file, che è in grado in modellare flussi ad accesso casuale.

Il funzionamento dello stream è molto semplice:

- Un programma, per ricevere in ingresso dei dati, apre uno stream su una sorgente di informazioni e ne legge *sequenzialmente* i dati.

- Un programma può anche inviare informazioni a un certo destinatario aprendo uno stream verso di esso e scrivendo *sequenzialmente* i dati in uscita.

Il processo di lettura di informazioni da uno stream segue sempre il seguente schema:

*open(stream);*

*while(there is more information)*

*read(information);*

*close(stream);*

Similmente, il processo di scrittura da stream segue sempre il seguente schema:

*open(stream);*

*while(there is more information)*

*write(information);*

*close(stream);*

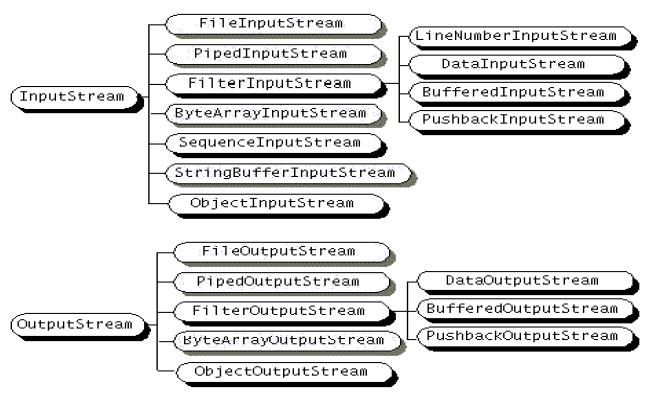
**Package java.io**

È un package dove è localizzata la libreria standard della piattaforma Java per la gestione degli stream. Qui è definito ogni specifico stream sottoforma di classi e interfacce. Perciò, un programma che ha bisogno di utilizzare delle operazioni di I/O avrà nell’intestazione l’*import* di alcune classi presenti in questo package (per importarle tutte -> import java.io.\*).

Le classi / interfacce in java.io sono *logicamente* suddivise in base alla funzionalità che compiono (acquisire informazioni in **input** vs produrre informazioni in **output**) e a come gestiscono i dati (utilizzo di uno stream **byte oriented** vs utilizzo di uno stream **char oriented**).

- **Stream byte oriented**: processa i dati byte per byte (per cui l’unità atomica di memorizzazione è il byte), ed è adatto per trattare dati grezzi (e.g. bit di un’immagine digitale o di un segnale sonoro digitalizzato). Un flusso di byte in ingresso viene gestito da una classe che specializza la classe astratta **InputStream**, mentre un flusso di byte in uscita viene gestito da una classe che specializza la classe astratta **OutputStream**.

- **Stream chat oriented**: processa i dati carattere per carattere (in particolare, in Java, i caratteri vengono memorizzati utilizzando le convenzioni UNICODE, secondo cui ciascun carattere occupa 16 bit di memoria), ed è adatto per trattare file di testo. Prevede dunque i concetti di linea (‘\n’), di tabulazione (‘\t’) e così via. Un flusso di caratteri in ingresso viene gestito da una classe che specializza la classe astratta **Reader**, mentre un flusso di caratteri in uscita viene gestito da una classe che specializza la classe astratta **Writer**.



java.io.InputStream:

Canali tipici di input:

- Un array di byte (classe ByteArrayInputStream)

- Un oggetto String (classe StringBufferInputStream)

- Un file (classe FileInputStream)

- Una pipe che realizza una comunicazione tra processi / thread (classe PipedInputStream)

- Una sequenza di dati proveniente da altri stream raggruppati insieme in un unico stream (classe SequenceInputStream)

- Altre sorgenti, come le connessioni a Internet

In aggiunta, la classe astratta FilterInputStream fornisce utili modalità di input per dati particolari come i tipi primitivi, i meccanismi di bufferizzazione e così via.

Metodi:

- read(): legge un byte dallo stream di input.

- read(byte[] array): legge dei byte dallo stream di input e li memorizza nell’array specificato.

– available(): restituisce il numero di byte disponibili nello stream di input.

- mark(): marca la posizione all’interno dello stream di input prima della quale i dati sono stati già letti.

- reset(): restituisce il controllo al punto dello stream che è stato marcato.

- markSupported(): verifica se mark() e reset() sono due metodi supportati all’interno dello stream.

- skip(): salta e scarta il numero specificato di byte dallo stream di input.

- close(): chiude lo stream di input.

java.io.OutputStream:

Canali tipici di output:

- Un array di byte (classe ByteArrayOutputStream)

- Un file (classe FileOutputStream)

- Una pipe che realizza una comunicazione tra processi / thread (classe PipedOutputStream)

- Un qualunque oggetto (classe ObjectOutputStream)

In aggiunta, la classe astratta FilterOutputStream fornisce utili modalità di output per dati particolari come i tipi primitivi, i meccanismi di bufferizzazione e così via.

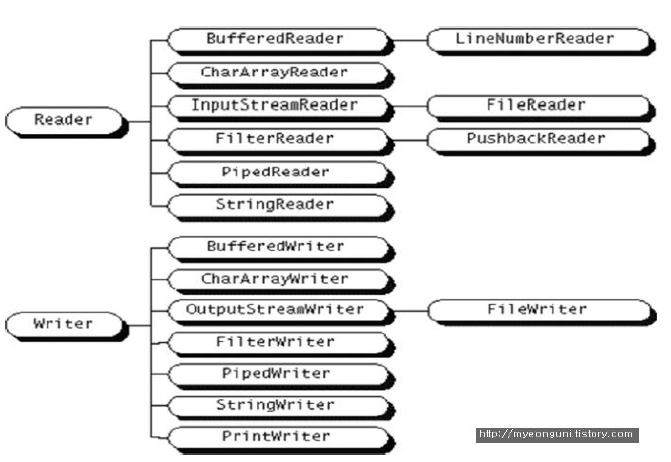
Metodi:

- write(): scrive dei byte sullo stream di output.

- write(byte[] array): scrive i byte dall’array specificato sullo stream di output.

- flush(): forza la scrittura di tutti i dati presenti nello stream di output verso la destinazione.

- close(): chiude lo stream di output.



java.io.Reader:

Canali tipici di input:

- Un array di caratteri (classe CharArrayReader)

- Un oggetto String (classe StringReader)

- Un file (classe FileReader)

- Una pipe che realizza una comunicazione tra processi / thread (classe PipedReader)

Metodi:

- ready(): verifica se lo stream di input è pronto per essere letto.

- read(char[] array): legge dei caratteri dallo stream di input e li memorizza nell’array specificato.

- read(char[] array, int start, int length): legge un numero di caratteri pari a *length* dallo stream di input e li memorizza nell’array specificato a partire dalla posizione indicata da *start*.

- mark(): marca la posizione all’interno dello stream di input prima della quale i dati sono stati già letti.

- reset(): restituisce il controllo al punto dello stream che è stato marcato.

- skip(): salta e scarta il numero specificato di byte dallo stream di input.

- close(): chiude lo stream di input.

java.io.Writer:

Canali tipici di output:

- Un array di caratteri (classe CharArrayWriter)

- Un oggetto String (classe StringWriter)

- Un file (classe FileWriter)

- Una pipe che realizza una comunicazione tra processi / thread (classe PipedWriter)

Metodi:

- write(char[] array): scrive i caratteri dall’array specificato sullo stream di output.

- wriet(String data): scrive la stringa specificata sullo stream di output.

- append(char c): inserisce il carattere c all’interno dello stream di output.

- flush(): forza la scrittura di tutti i dati presenti nello stream di output verso la destinazione.

- close(): chiude lo stream di output.

Flussi di byte da / verso file:

Per poter scrivere su un file di byte è necessario far cooperare due elementi:

- Un oggetto che gestisca lo stream (=il canale di comunicazione verso il file) e che sia in grado di inviare byte lungo lo stream; si tratta di un’istanza di una sottoclasse di OutputStream.

- Un oggetto che crei fisicamente un collegamento con il file; si tratta di un’istanza della classe **File**.

Un discorso analogo vale per la lettura da un file di byte.

La classe File fornisce una rappresentazione astratta e indipendente dal sistema dei pathname gerarchici che si riferiscono a file e directory. In generale, non si ha a che fare con una rappresentazione di questo tipo, poiché le interfacce utente e i file system utilizzano pathname che dipendono dal sistema operativo correntemente in uso. Per esempio, su Windows un pathname ha una forma del tipo ***C:\\directory\file.ext***, mentre lo stesso path su Unix sarebbe simile a ***/directory/file.ext***.

In particolare, per gestire la costruzione dei path, la classe File mette a disposizione un campo statico **separatorChar** che varia in funzione del sistema in cui l’applicazione è in esecuzione; File.separatorChar restituisce il primo carattere di un altro campo statico della classe File, **separator**, che restituisce **\\** su Windows e **/** su Unix.

NB: Come già accennato in precedenza, la classe File permette solo di creare un collegamento con il file fisico; per poter avere delle scritture o letture su file, è necessario anche l’appoggio di uno stream tramite una o più classi del package java.io.

Composizione di stream:

Vediamo ora qualche esempio di composizione di stream.

*- BufferedReader in = new BufferedReader (new FileReader (“file.txt”));*

Viene aperto un file in lettura come stream orientato ai caratteri, e poi viene aperto uno stream in lettura orientato ai caratteri realtivo a un buffer, che permette la bufferizzazione dei dati in memoria principale e, quindi, una maggiore efficienza.

*- String s = “this is foo”;*

*DataInputStream in = new DataInputStream (new ByteArrayInputStream (s.getBytes()));*

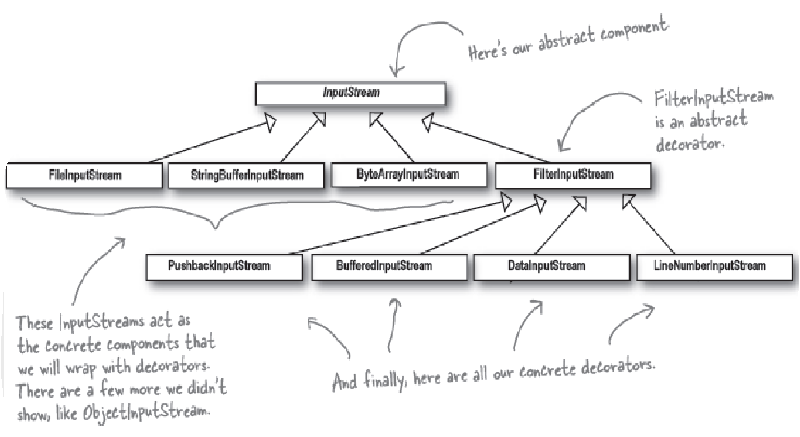
Viene creato uno stream in lettura per array di byte, e poi viene aperto uno stream (DataInputStream) che consente di rappresentare tutti i tipi di dato primitivi di Java, ottenendo così un codice più flessibile.

*- PrintWriter out = new PrintWriter (new BufferedWriter (new FileWriter(“file.txt”)));*

Viene aperto un file in scrittura come stream orientato ai caratteri, viene aperto uno stream in scrittura orientato ai caratteri relativo a un buffer, e infine viene aperto uno stream in scrittura orientato ai caratteri che supporta la stampa di vari tipi basici formattandoli come testo.

NB: esempi di codice nelle slide e tra gli esempi!

Da questi esempi possiamo evincere che l’organizzazione delle classi all’interno del package java.io riflette l’applicazione del design pattern GoF Decorator:



**File ad accesso casuale**

Le classi che abbiamo visto finora e che consentono di operare su stream provenienti da file (e.g. FileInputStream, FileOutputStream, FileReader, FileWriter) forniscono dei metodi per leggere dei dati esclusivamente in maniera sequenziale. Tuttavia, abbiamo precedentemente accennato che i file (tipicamente presenti all’interno del file system locale) sono in grado di modellare flussi ad accesso casuale. Con l’accesso casuale, i dati all’interno di un file possono essere letti e riletti da qualsiasi posizione nella struttura soggiacente lo stream, prescindendo dunque dalla sequenzialità.

La classe Java che supporta questa modalità di accesso ai file è **RandomAccessFile** che, pur rientrando nel package java.io, non fa parte di alcuna delle gerarchie di classi per la gestione di input / output, bensì è una classe a sé stante. Il modo di interazione con RandomAccessFile *ricorda* quello definito per DataInputStream e per DataOutputStream; la differenza risiede nel fatto che RandomAccessFile esporta in più il metodo **seek()** che consente di posizionarsi in un qualsiasi punto del file.

RandomAccessFile rf = **new RandomAccessFile**(**"rtest.dat"**, **"rw"**);

…

rf = **new RandomAccessFile**(**"rtest.dat"**, **"rw"**);

…

rf = **new RandomAccessFile**("rtest.dat", "r");

**Serializzazione / deserializzazione**

Sappiamo già che, nei linguaggi di programmazione object-oriented, il ciclo di vita canonico di un oggetto:

- Inizia con l’allocazione di un opportuno spazio di memoria (i.e. *new* in Java) e con l’invocazione di uno dei costruttori definiti dalla classe di afferenza.

- Termina con la deallocazione esplicita dell’oggetto (tipica di C++), con la deallocazione implicita da parte di un garbage collector (tipica di Java) oppure con la terminazione dell’applicazione che ha creato l’oggetto.

Il ciclo di vita canonico di un oggetto ha generalmente senso; tuttavia, esistono molti scenari in cui risulta utile o desiderabile che:

- Lo stato di un oggetto continui a esistere anche dopo la terminazione dell’esecuzione dell’applicazione che lo ha creato.

- Un’applicazione possa configurare il suo contesto di esecuzione attraverso il ripristino dello stato di un insieme di oggetti.

- Lo stato di un oggetto possa migrare tra un nodo e l’altro di un’applicazione distribuita.

In generale, l’utilizzo di file o DBMS in parte sopperisce a queste necessità, pur tuttavia con i seguenti svantaggi:

- File: richiede la gestione esplicita delle convenzioni di memorizzazione e del formato dei dati utilizzato.

- DBMS: generalmente non è molto adatto a sistemi di dimensioni limitate o non specificatamente “enterprise”; inollre, richiede la gestione del mismatch tra la rappresentazione object-oriented del dominio e la rappresentazione specifica dei dati da adottare nel database.

Per questi motivi, Java propone un modo semplice ed efficiente per garantire la persistenza degli oggetti creati a runtime sfruttando il concetto di stream. In particolare:

- Il salvataggio dello stato interno di un oggetto (il quale viene convertito in uno stream di byte) viene detto **serializzazione**.

- Il ripristino dello stato di un oggetto precedentemente salvato (=conversione da stream di byte a oggetto Java) viene detto **deserializzazione**.

Uno stream di byte creato tramite la serializzazione è *platform-indipendent*. Perciò, è possibile che un oggetto serializzato su una specifica piattaforma venga correttamente deserializzato su una piattaforma differente.

Le classi che supportano la gestione di object serialization e di object deserialization sono rispettivamente:

- **ObjectOutputStream**, che contiene il metodo **writeObject()** per serializzare un oggetto.

- **ObjectInputStream**, che contiene il metodo **readObject()** per deserializzare un oggetto.

NB: 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.

Notiamo inoltre che non tutto può essere effettivamente serializzato in Java. In particolare:

- Java tratta come serializzabili tutte le variabili afferenti ai tipi di dato primitivi.

- Per i tipi di dato complessi (i.e. le classi), i meccanismi di serializzazione sono abilitati se e solo se una classe realizza l’interfaccia **Serializable** del package java.io.

Serializable è un’interfaccia particolare che non definisce alcuna operazione e, quindi, non richiede la definizione di alcun metodo da parte delle classi che la realizzano. In altre parole, si tratta di una **tagging interface**.

Di seguito vengono proposte alcune indicazioni delle JavaDoc di java.io.Serializable in Java 8:

- “Tutti i sottotipi di una classe serializzabile sono a loro volta serializzabili”.

- “Per consentire ai sottotipi di classi non serializzabili di essere serializzati, è possibile assegnare ai sottotipi la responsabilità di salvare e recuperare lo stato dei campi pubblici, protetti e (se accessibili) “package” della super-classe. I sottotipi possono assumere questa responsabilità solo se la classe che estendono ha un costruttore di default accessibile (=pubblico o protetto)”.

- “Durante la deserializzazione, i campi delle classi non serializzabili saranno inizializzati tramite il loro costruttore di default (pubblico o protetto). Il costruttore di default deve essere accessibile alle sottoclassi serializzabili. I campi delle sottoclassi serializzabili verranno recuperate dallo stream”.

- “Le classi che richiedono una gestione speciale durante i processi di serializzazione e di deserializzazione devono implementare metodi speciali con esattamente queste segnature:

- *private void writeObject(java.io.ObjectOutputStream out) throws IOException*

- *private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException*

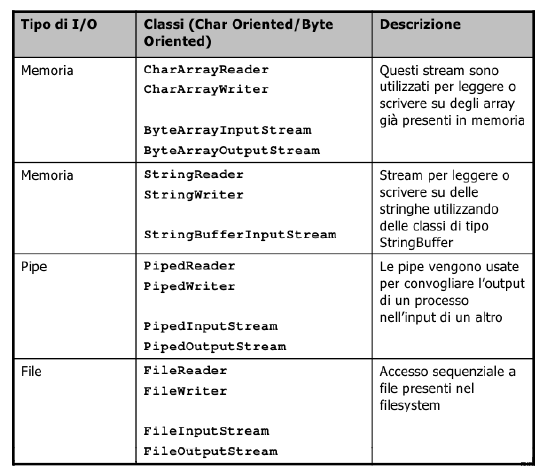
- *private void readObjectNoData() throws ObjectStreamException*”.

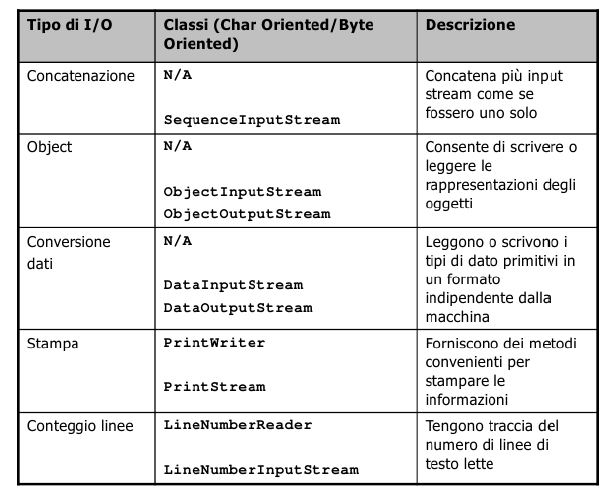
Serializzazione e variabili di classe:

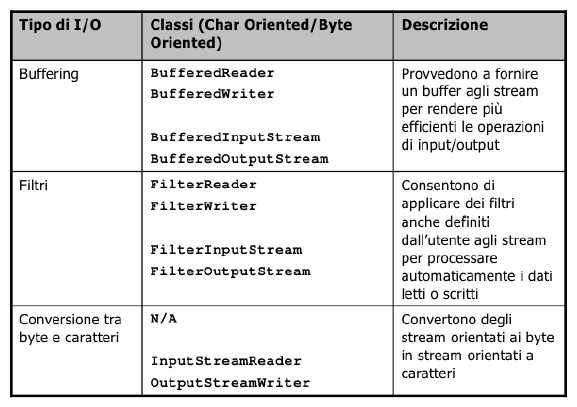
Il serializzatore e il deserializzatore in Java lavorano su oggetti allocati nell’heap di memoria. In particolare, le variabili / gli attributi di classe vengono memorizzati all’interno del run-time constant pool; tuttavia, vengono istanziati sempre e soltanto al momento del caricamento (dinamico) della classe, e vengono inizializzati sempre con il valore (statico) definito nella classe. Di conseguenza, non sono oggetto di salvataggio / caricamento da parte del serializzatore / deserializzatore. Per serializzare / deserializzare il valore corrente, è necessario prevedere dei meccanismi espliciti di salvataggio e ripristino in memoria.

**Breve schema riassuntivo**

Nelle pagine seguenti viene riportata una tabella riassuntiva che descrive brevemente le classi Java che si occupano della gestione di stream I/O.







# Lezione42-43 – Eccezioni

All’interno del software possono presentarsi differenti tipi di errore, ciascuno dei quali si presenta in un momento diverso:

- **Errore di sintassi**: in fase di compilazione

- **Errore di comportamento**: in fase di test della singola unità software

- **Errore di interazione**: in fase di test di integrazione

Nella pratica, quasi mai è possibile essere certi dell’assenza di errori in un sistema software: è molto comune che un sistema in esecuzione possa andare in crush. Per questa ragione, spesso è opportuno progettare e sviluppare delle logiche di controllo e di gestione degli errori parallelamente alla progettazione dei comportamenti desiderati.

Storicamente questo problema è stato gestito mediante la definizione di determinate convenzioni che, per esempio, possono consistere in particolari valori di ritorno (come avviene in C). Comunque sia, i linguaggi di progettazione / programmazione avanzati prevedono costrutti nativi per la definizione e gestione esplicita del problema.

Esempi:

- La UML State Machine ricorre ai cosiddetti **exit point**.

- Java ricorre alla classe **java.lang.Exception**.

Definizione:

**Eccezione** = 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:

Possono considerarsi eccezioni i seguenti eventi:

- Si rompe l’hard disk o l’SSD.

- Si tenta di accedere a un indice fuori da un array.

- La connessione di rete diviene assente.

- La serializzazione / deserializzazione di un oggetto non va a buon fine.

- Un file a cui si vuole accedere è protetto in lettura.

Analizziamo ora un esempio di porzione di codice.

“Scrivere una funzione che apre un file, ne verifica la dimensione e lo copia in memoria dopo averne allocato sufficiente spazio”.

*readFile {*

*open(file);*

*s = size(file);*

*mem = allocate(s);*

*read(file, mem);*

*close(file);*

*}*

Sembra tutto ok ma:

- Se il file non può essere aperto?

- Se la lunghezza del file non può essere determinata?

- Se non vi è abbastanza memoria da allocare?

- Se la lettura fallisce?

- Se il file non può essere chiuso?

Risulta dunque necessario aggiungere dei controlli relativi alla logica di errore all’interno di questo codice.

*readFile {*

*errcode = 0;*

*open(file);*

*if(!failed) {*

*s=size(file);*

*if(s >= 0) {*

*mem = allocate(s);*

*if(isFree(mem)) {*

*read(file, mem);*

*if(failed) errcode = -1;*

*} else errcode = -2;*

*} else errcode = -3;*

*close(file);*

*if(closeFailed && errcode == 0) errcode = -4;*

*else if(errcode != 0) errcode = -5;*

*} else errcode = -6;*

*return errcode;*

*}*

Ora il programma è perfettamente funzionante e tiene conto di tutte le possibili eccezioni che potrebbero sollevarsi. Tuttavia, è diventato illeggibile a causa del mescolamento tra logica di controllo e logica di errore, il che inficia sulla mantenibilità del codice.

È possibile ricorrere a una soluzione altrettanto corretta ma di più facile comprensione e, quindi, più mantenibile? Naturalmente sì, ed è la seguente:

*readFile {*

*try {*

*open(file);*

*s = size(file);*

*mem = allocate(s);*

*read(file, mem);*

*close(file);*

*} catch (failedOpen) { // Da notare come logica di controllo e logica di errore siano separate.*

*…*

*} catch (noSize) {*

*.…*

*} catch (failedCopy) {*

*…*

*} catch (failedClose) {*

*…*

*}*

*}*

**Costrutto try / catch**

È un costrutto Java in cui si tenta di eseguire un determinato blocco di codice e, se si intercetta un’eccezione, si cerca di porvi rimedio.

Di seguito viene riportata la sintassi generale del costrutto.

*try {*

*<IstruzioniLogicaDiControllo>*

*} catch (<TipoEccezione> <Identificatore>) {*

*<IstruzioniLogicaDiErrore>*

*}*

La clausola **try** definisce un blocco di istruzioni della logica di controllo in cui può verificarsi un’eccezione. Essa è seguita da una o più clausole **catch**, che specificano quali eccezioni vengono gestite.

Per aiutarci nell’analisi del funzionamento di questo costrutto, ricorriamo a un semplice esempio:

*int a, b;*

*…*

*try {*

*divisione = operazioni.divisione(a,b);*

*System.out.println(“Il risultato della divisione è: “ + divisione);*

*} catch (ArithmeticException e) {*

*System.out.println(e);*

*System.out.println(“Non puoi effettuare una divisione per zero”);*

*} catch (Exception e) {*

*System.out.println(“Errore generico”);*

*}*

Qui abbiamo due clausole catch: ciascuna di esse corrisponde a un tipo di eccezione sollevata. Al verificarsi di un’eccezione *exc* all’interno della clausola try, la computazione salta direttamente alla prima istruzione della prima clausola catch relativa a un’eccezione *ex* tale che *exc* is-a-kind-of *ex*.

Notiamo infatti che le eccezioni in Java sono degli Object veri e propri e, in quanto tali, possono sfruttare il concetto dell’ereditarietà e possono relazionarsi tra loro mediante generalizzazioni. In particolare, tutte le eccezioni di Java are-a-kind-of java.lang.Exception.

Tornando al nostro esempio, se nella clausola try viene sollevata un’eccezione di tipo ArithmeticException o di un qualunque suo sottotipo, allora verrà eseguita la prima clausola catch. Se invece viene sollevata un’eccezione di un qualunque altro tipo, allora verrà eseguita la seconda clausola catch. Notiamo che, se avessimo invertito la clausola catch relativa ad ArithmeticException con la clausola catch relativa a Exception, il compilatore avrebbe dato un errore dovuto a un “unreachable code”: infatti, la clausola relativa a Exception sarebbe stata la prima in grado di gestire qualunque eccezione, comprese quelle di tipo ArithmeticException. Ne consegue che l’ordine con cui compaiono le clausole catch è importante.

Inoltre, se non avessimo avuto una clausola catch relativa alla classe Exception, sarebbe stato possibile avere un’eccezione non catturabile e, quindi, non gestibile: di fronte a questo scenario, la procedura corrente viene interrotta e l’eccezione viene propagata al chiamante e così via, finché l’eccezione non viene effettivamente gestita, oppure finché la propagazione non giunge al livello più alto possibile, cioè al sistema operativo, il quale tipicamente termina il programma.

Comunque sia, nel momento in cui viene sollevata un’eccezione, sia se essa viene gestita, sia se viene propagata, il flusso della logica di controllo all’interno del costrutto try viene interrotto definitivamente e non viene più ripreso. In particolare:

- Se l’eccezione viene gestita, dopo aver eseguito la clausola catch appropriata, la computazione riprende *dopo* il costrutto try / catch.

- Se invece l’eccezione viene propagata, il controllo torna definitivamente al chiamante (come se fosse stato invocato un comando di return).

Per questo motivo, potrebbe tornare utile il costrutto **try / catch / finally** che, rispetto al costrutto try / catch appena analizzato, presenta in più una clausola **finally**, che è opzionale nel caso in cui si ha almeno una clausola catch. Analizziamo il suo schema di funzionamento nei tre scenari possibili:

- Non viene sollevata alcuna eccezione → le istruzioni nella clausola finally vengono eseguite dopo che si è concluso il blocco try.

- Si verifica un’eccezione catturata dai comparti catch → le istruzioni nella clausola finally vengono eseguite dopo le istruzioni della clausola catch appropriata.

- Si verifica un’eccezione non prevista dai comparti catch → le istruzioni nella clausola finally vengono eseguite prima dell’inoltro dell’eccezione al chiamante del metodo.

In definitiva, se presente, la clausola finally viene sempre eseguita; l’unico caso in cui ciò non avviene è a seguito dell’invocazione dell’istruzione System.exit(int n). Questo comparto serve dunque a garantire una consistenza dello stato delle risorse utilizzate all’interno del blocco try / catch e, per esempio, a tale scopo può contenere delle istruzioni che chiudono dei file o che rilasciano delle risorse.

NB: I costrutti try / catch / finally possono essere annidati a piacere. Ad esempio, se in una clausola catch (o finally) può essere generata un’eccezione, le sue istruzioni possono essere a loro volta racchiuse in un altro blocco try.

**Nota**: da qui in poi è possibile vedere gli esempi nelle cartelle.

**Clausola throws vs comando throw**

Esistono situazioni in cui non si è in grado di stabilire opportunamente la gestione di un’eccezione. In tal caso è necessario identificare il metodo come possibile sorgente di “errore non gestito” ed è il chiamante che deve farsi carico di gestire l’eventuale eccezione sollevata (sempre se non la propaga a sua volta).

Un metodo che può sollevare un’eccezione non gestita deve presentare la clausola **throws** all’interno della sua dichiarazione. Se il chiamante gestisce effettivamente l’eccezione, deve racchiudere l’invocazione del metodo all’interno di un blocco try / catch.

Attenzione a non confondere la clausola throws con il comando **throw**, il quale invece è un imperativo che indica il lancio “volontario” di un’eccezione da parte del programmatore; in altre parole, si tratta di un’istruzione che comunica alla JVM che deve entrare in logica di errore. Questa opportunità rende possibile la definizione di nuove eccezioni personalizzate da parte del programmatore, che, appunto, può lanciarle esplicitamente con il comando throw se si verificano determinate condizioni.

**Gerarchia di errori / eccezioni**

Sia gli oggetti errore che gli oggetti eccezione sono sempre istanze di una classe derivata da **java.lang.Throwable**. Gli errori e le eccezioni costituiscono le due macro-categorie di questa gerarchia di base. In particolare:

- Gli errori sono istanze di una sottoclasse di java.lang.Error (che, appunto, è una classe derivata di java.lang.Throwable).

- Le eccezioni sono istanze di una sottoclasse di java.lang.Exception (che, allo stesso modo, è una classe derivata di java.lang.Throwable).

Errori:

Sono anomalie che si verificano all’interno della virtual machine (e.g. dynalic linking, hard failure).

Esempi: OutPfMemoryError, StackOverflowError.

In generale, non si riesce a risolverli e/o a gestirli in modo sano e adeguato poiché sono legati a condizioni abnormali e ad anomalie gravi.

Eccezioni:

Si suddividono a loro volta in due categorie:

- **java.lang.RuntimeException** e sue sottoclassi: sono dette **unchecked** (non verificate), il che implica che non è obbligatorio gestirle, né tantomeno dichiararle nella segnatura del metodo che le genera tramite la clausola throws. Si verificano quando è stato commesso un errore di programmazione (e.g. cast definito male, che dà luogo a una ClassCastException; accesso a un puntatore nullo, che dà luogo a una NullPointerException).

- Tutte le altre classi che non derivano da RunTimeException: sono dette **checked** (verificate), il che implica che è obbligatorio gestirle in un apposito blocco try / catch oppure delegarne la gestione usando la clausola throws nel metodo che le genera. Si verificano quando avviene qualcosa di imprevisto (e.g. errore nell’apertura di un file, che dà luogo a una FileNotFoundException).

**Operazioni in Throwable**

- getMessage(): accede al messaggio contenente i dettagli dell’eccezione.

- toString(): restituisce una breve descrizione dell’oggetto Throwable, compresi i dettagli dell’eccezione, se esistenti.

- printStackTrace(): stampa sullo standard di error l’eccezione con il relativo stack delle chiamate.

- getCause(): accede al messaggio contenente la causa dell’eccezione (a partire da Java 1.4 è possibile impostare l’eccezione originale come “causa” di una nuova eccezione; in altre parole, si possono annidare eccezioni).

- initCause(Throwable cause): configura o sovrascrive la causa di un’eccezione.

**Conversione e chaining di eccezioni**

Tra le altre cose, è anche possibile gestire un’eccezione semplicemente lanciandone una diversa. Questo meccanismo ha un’utilità a livello applicativo: se per esempio si verifica un’eccezione all’interno del sistema e la si vuole comunicare all’utente mostrandola sulla user interface (UI), non è indicato mostrare il messaggio dell’eccezione nudo e crudo con tutti i dettagli tecnici, bensì è preferibile mostrare a schermo un messaggio più *user friendly*, che spieghi in modo chiaro e naturale la causa dell’eccezione e/o il modo per risolvere il problema.

Per far ciò, esistono due possibili tecniche:

- **Conversione**: l’eccezione originale e1 viene gestita lanciandone una nuova (e2); il parametro di e2 sarà e1.getMessage(). Viene propagata la nuova eccezione perdendo traccia dell'errore interno.

- **Chaining** (o wrapping): l’eccezione originale e1 viene gestita lanciandone una nuova (e2); il parametro di e2 sarà e1.getCause(). Viene propagata la nuova eccezione **mantenendo** traccia dell'errore interno.

**Eccezioni e overriding**

In caso di override, le eccezioni di tipo checked previste dalla specifica di un’operazione hanno un impatto anche sull’implementazione dei relativi metodi delle sottoclassi che effettuano l’override. In particolare, se determinate sottoclassi effettuano un override di un metodo M dichiarato nella classe padre, bisogna assicurarsi che le classi figlie lancino all’interno di M solo eccezioni compatibili con (are-a-kind-of) quelle previste / gestite dalla super-classe. Se ciò non si verificasse e una sottoclasse lanciasse in M un’eccezione di tipo diverso (o comunque più generale), si dovrebbe gestire l’eccezione direttamente nella classe figlia oppure effettuare una conversione / un chaining. Tale principio assicura che il codice che funzioa all’interno della classe base funzioni automaticamente anche con qualsiasi oggetto derivato dalla super-classe.

Lo stesso discorso si applica nella definizione dei metodi che implementano un’operazione di un’interfaccia.

Tuttavia, per i costruttori non vale questo principio: i costruttori delle sottoclassi possono sollevare qualsiasi eccezione senza problemi e, inoltre, devono esplicitare anche tutte le eccezioni lanciabili all’interno del costruttore della super-classe. Ciò è dovuto ai seguenti motivi:

- Il costruttore delle sottoclassi invocano esplicitamente o implicitamente il costruttore della super-classe (cosa che in generale non avviene con gli altri metodi).

- Le differenze tra il codice del costruttore della classe padre e il codice del costruttore delle classi figlie determinano differenze tra lo stato della super-classe e lo stato delle sottoclassi (e la porzione di stato “in più” delle classi figlie non coinvolge la classe parent).

P79