LEZIONE 1

INTRODUZIONE ALLA PROGETTAZIONE DEL SOFTWARE

CONTESTO ORGANIZZATIVO:

Attori nella progettazione del software:
– Committente
– Esperti del domino
– Analista
– Progettista
– Programmatore
– Utente finale
– Manutentore

Classificazione delle applicazioni rispetto al flusso di controllo:
• **Sequenziali**: un unico flusso di controllo governa l'evoluzione dell'applicazione.
• **Concorrenti**: le varie attività necessitano di sincronizzazione e comunicazione.
  • Composte da varie attività sequenziali che possono (e devono) essere sincronizzate al fine di garantire la correttezza.
  • Il tempo di esecuzione influenza le prestazioni, non la correttezza.
• **Dipendenti dal tempo**: esistono vincoli temporali riguardanti sia la velocità di esecuzione delle attività sia la necessità di sincronizzare le attività stesse.

Classificazione delle applicazioni rispetto agli elementi di interesse primario:
• **Orientate alla realizzazione di funzioni**: la complessità prevalente del sistema riguarda le funzioni da realizzare.
• **Orientate alla gestione dei dati**: l'aspetto prevalente è rappresentato dai dati che vengono memorizzati, ricercati, e modificati, e che costituiscono il patrimonio informativo di una organizzazione.
• **Orientate al controllo**: la complessità prevalente del sistema riguarda il controllo delle attività che si sincronizzano e cooperano durante l'evoluzione del sistema.

CICLO DI VITA DEL SOFTWARE:

1. Studio di fattibilità e raccolta dei requisiti:
– valutare costi e benefici
– pianificare le attività e le risorse del progetto
– individuare l'ambiente di programmazione (hardware/software)
– raccogliere i requisiti

2. Analisi dei requisiti:
– si occupa del cosa l'applicazione dovrà realizzare
– descrivere il dominio dell'applicazione e specificare le funzioni delle varie componenti: lo schema concettuale

3. Progetto e realizzazione:
– si occupa del come l'applicazione dovrà realizzare le sue funzioni
– definire l'architettura del programma
– scegliere le strutture di rappresentazione
– scrivere il codice del programma e produrre la documentazione

4. Verifica:
– Il programma svolge correttamente, completamente, efficientemente il compito per cui è stato sviluppato?

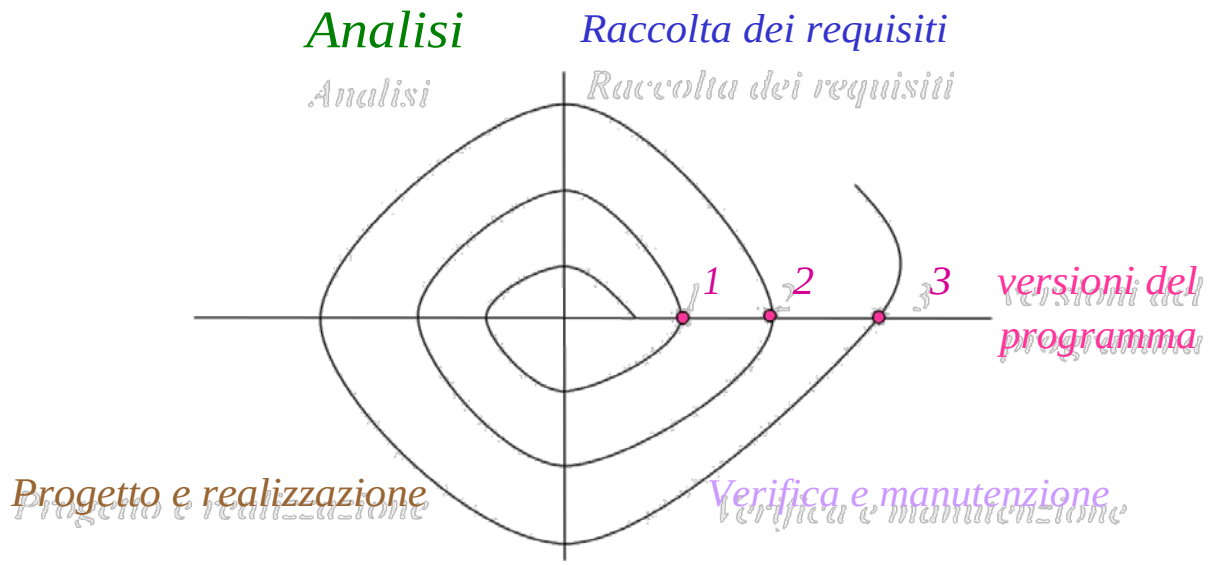
5. Manutenzione:
– Controllo del programma durante l'esercizio
– Correzione e aggiornamento del programma

Fattori di qualità del SW:

ESTERNE	INTERNE
Correttezza	Efficienza
Affidabilità	***Strutturazione***
Robustezza	***Modularità***
Sicurezza	***Comprensibilità***
Innocuità	Verificabilità
Usabilità	Manutenibilità
Estendibilità	Portabilità

Riusabilità
Interoperabilità

Ciclo di vita del software (modello a spirale)″



QUALITA' ESTERNE ED INTERNE:

• Qualità esterne:
– Sono le qualità visibili agli utenti del sistema
– Percepibili anche da chi non è specialista
– Non richiedono l'ispezione del codice sorgente

• Qualità interne:

– Sono le qualità che riguardano gli sviluppatori!
– Valutabili da specialisti
– Richiedono conoscenza della struttura del programma
A volte la distinzione fra qualità esterne ed interne non è perfettamente marcata

• **Correttezza (è fondamentale)**: il software fa quello per il quale è stato progettato
• **Affidabilità**: si può fare affidamento sulle funzionalità del software: vincoli rispettati -> risultati disponibili
• **Robustezza**: comportamento accettabile anche nel caso di situazioni non previste nella specifica dei requisiti
• **Innocuità**: il sistema non entra in certi stati (pericolosi)
• **Sicurezza**: riservatezza nell'accesso alle informazioni
• **Usabilità**: enfasi sull'utente, psicologia cognitiva, fattori fisici/ergonomici, mentalità dell'utente e le interfacce grafiche/visuali vengono spesso completamente ignorate.
• **Estendibilità:** facilità con cui il SW può essere adattato a modifiche delle specifiche
    Nota: Importantissimo in grandi programmi
    Nota: Due principi per l'estendibilità: semplicità di progetto e decentralizzazione nell'architettura del SW
• **Riusabilità**: facilità con cui il SW può essere re-impiegato in applicazioni diverse da quella originaria
    Nota: Evita di reinventare soluzioni
    Nota: Richiede alta compatibilità
– Libreria di componenti riutilizzabili
– Progetto più generale possibile
– Documentazione

– Maggiore affidabilità
• **Interoperabilità**: facilità di interazione con altri moduli al fine di svolgere un compito più complesso, problemi tecnologici e semantici e favorisce la riusabilità
• **Efficienza**: si riferisce al "peso" che il software ha sulle risorse del sistema
– Tempo di esecuzione"
– Utilizzo di memoria
– Teoria della complessità: limiti asintotici, caso medio e caso peggiore
– Simulazioni (e.g., teoria delle reti di code)
– Legata alle prestazioni (quest'ultime percepibili dall'utente, quindi qualità esterne)
• **Strutturazione**: capacità del SW di riflettere con la sua struttura le caratteristiche del problema trattato e delle soluzioni adottate
• **Modularità**: grado di organizzazione del SW in parti ben specificate ed interagenti
• **Comprensibilità**: capacità del SW di essere compreso e controllato anche da parte di chi non ha condotto il progetto
– si applica sia al software che al processo, facilitata da: strutturazione e modularità
– la comprensibilità del software facilita l'analisi della correttezza ed il riuso
• **Verificabilità**: la possibilità di verificare che gli obiettivi proposti siano stati raggiunti
– È una caratteristica sia del processo che del prodotto
– Facile: il codice soddisfa gli standard di codifica?
– Difficile: il codice fa ciò che deve fare? (vedi correttezza)
• **Manutenibilità**: facilità nell'effettuare modifiche
• **Portabilità**: facilità nell'operare su diverse piattaforme modifiche, linguaggi compilabili su più piattaforme e macchine virtuali (html/java)

Non tutte le qualità possono essere massimizzate, alcune sono intrinsecamente in contrasto fra loro, ad esempio: usabilità e sicurezza oppure efficienza e portabilità è quindi necessario scegliere un adeguato bilanciamento

Principi guida nello sviluppo del software
• **Rigore e formalità**: lo sviluppo del software è una attività creativa che va accompagnata da un approccio rigoroso (o addirittura formale: in logica o matematica) permette di realizzare prodotti affidabili, controllarne il costo, aumentare la fiducia nel loro corretto funzionamento.
• **Separazione degli interessi**: affrontare separatamente i diversi aspetti per dominare la complessità.
• **Modularità** (realizza la separazione degli interessi in 2 fasi) : tratta i dettagli di singoli moduli in modo separato e tratta separatamente dai dettagli interni dei singoli moduli le relazioni che sussistono tra i moduli stessi.
• **Astrazione**: identifica aspetti fondamentali ed ignora i dettagli irrilevanti.
• **Anticipazione del cambiamento**: per favorire l'estendibilità e il riuso.
• **Generalità**: ricerca di soluzioni generali.
• **Incrementalità**: per anticipare feedback dell'utente per facilitare verifiche di correttezza per predisporsi alla estendibilità e al riuso.

Modularizzazione :
Principio secondo il quale il software è strutturato secondo unità, dette appunto moduli, un modulo è una unità di programma con le seguenti caratteristiche:
– ha un obiettivo chiaro
– ha relazioni strutturali con altri moduli
– offre un insieme ben definito di servizi agli altri moduli (server)
– può utilizzare servizi di altri moduli (client)

Principi per la modularità:
• **Principio di unitarietà** (incapsulamento, alta coesione): un modulo deve corrispondere ad una unità concettuale ben definita e deve incorporare tutti gli aspetti relativi a tale unità concettuale
• **Poche interfacce** (basso accoppiamento): un modulo deve comunicare con il minor numero di moduli possibile (quelli necessari)
• **Poca comunicazione** (basso accoppiamento): un modulo deve scambiare meno informazioni possibili (quelle necessarie) con gli altri moduli
• **Comunicazione chiara** (interfacciamento esplicito): informazione scambiata predeterminata e più astratta possibile
• **Occultamento di informazioni inessenziali** (information hiding): le informazioni che non devono essere scambiate devono essere gestite privatamente dal modulo.

LEZIONE 2 – SLIDE 1

ANALISI

L'analisi è la fase del ciclo di sviluppo del software caratterizzata da:
INPUT: requisiti raccolti
OUTPUT: schema concettuale (anche detto modello di analisi) dell'applicazione
OBIETTIVO: costruire un modello dell'applicazione che sia completo, preciso e rigoroso ma anche leggibile, indipendente da linguaggi di programmazione e traducibile in un programma e concentrarsi su cosa, e non su come (indipendenza da aspetti realizzativi/tecnologici)

Schema Concettuale: lo schema concettuale è costituito da:
• **Il diagramma delle classi e degli oggetti**: descrive le classi dell'applicazione e le loro proprietà; descrive anche gli oggetti particolarmente significativi.
• **Il diagramma delle attività**: descrive le funzionalità fondamentali che il sistema deve realizzare, in termini di processi modellati nel sistema.
• **Il diagramma degli stati e delle transizioni**: descrive, per le classi significative, il tipico ciclo di vita delle sue istanze
• **I documenti di specifica: d**escrivono con precisione quali condizioni devono soddisfare i programmi che realizzano il sistema **e v**iene prodotto un documento di specifica per ogni classe, ed un documento di specifica per ogni use case .

Modelli e metodi per l'analisi:
• **Orientati alle funzioni** (metodologie utilizzate in passato)**:**
– diagrammi funzionali
– diagrammi di flusso di controllo
– diagrammi di flusso di dati
• **Orientati agli oggetti** (metodologie utilizzate attualmente basati sul linguaggio **UML**):
– Booch
– OOSE (Jacobson)
– OMT (Rambaugh)
– Coad`Yourdon

IL LINGUAGGIO UML

UML sta per Unified Modeling Language nato nel 1994 come unificazione di: Booch, Rumbaugh: OMT (Object Medeling Technique) e Jacobson: OOSE (Object-Oriented Software Engineering).
I diagrammi UML sono di diversi tipi :
• **Diagrammi strutturali**:
– Diagramma delle classi e degli oggetti (class and object diagram)
• **Diagrammi comportamentali:**
– Diagramma delle attività (activity diagram),
– Diagramma degli stati e delle transizioni (state/transition diagram),
– Diagramma degli use case (use case diagram),
– Interaction (Sequence e Collaboration diagram)
• **Diagrammi architetturali:**
– Component diagram
– Deployment diagram

NOTA BENE : Useremo UML con alcune limitazioni e regole precise.
Nella fase di analisi ci si concentra sulle classi più che sugli oggetti, essi servono essenzialmente per descrivere elementi singoli particolarmente significativi (oltre che per scopi didattici)

OGGETTI IN UML

Un oggetto in UML modella un elemento del dominio di analisi che
– ha vita propria
– è identificato univocamente mediante l'identificatore di oggetto
– è istanza di una classe (la cosiddetta classe più specifica – vedremo che, in determinate circostanze, un oggetto è istanza di più classi, ma in ogni caso, tra le classi di cui un oggetto è istanza, esiste sempre la classe più specifica)

CLASSI IN UML

Una classe modella un insieme di oggetti omogenei (le istanze della classe) ai quali sono associate proprietà statiche e dinamiche (operazioni). Ogni classe è descritta da:
– un nome
– un insieme di proprietà "locali" (astrazioni delle proprietà comuni degli oggetti che sono istanze delle classi)

•Tra un oggetto che è istanza di una classe C e la classe C si traccia un arco Instanceof
•Ricordiamo che gli oggetti formano il livello estensionale, mentre le classi a livello intensionale

Proprietà di classi: attributi in UML

Un attributo modella una proprietà locale della classe ed è caratterizzato da un nome e dal tipo dei valori associati:

• Ogni attributo di una classe stabilisce una proprietà locale valida per tutte le istanze della classe. Il fatto che la proprietà sia locale significa che è una proprietà indipendente da altri oggetti.

• Formalmente, un attributo A della classe C si può considerare una funzione che associa un valore di tipo T ad ogni oggetto che è istanza di C

• Gli attributi di una classe determinano gli attributi delle sue istanze

• **Regola importante**: se una classe C ha un attributo A di tipo T, ogni oggetto che è istanza di C ha l'attributo A, con un valore associato di tipo T

• **Regola importante**: un oggetto X non può avere un valore per un attributo non definito nella classe di cui X è istanza

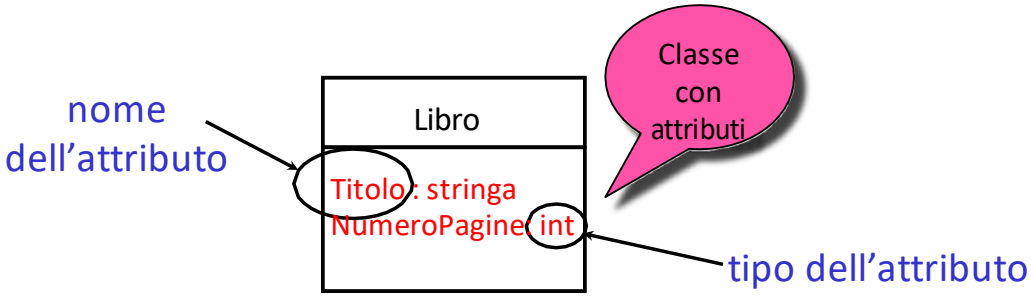
• Due oggetti con identificatori diversi sono comunque distinti, anche se hanno i valori di tutti gli attributi uguali

• Due oggetti diversi devono avere identificatori diversi, anche se possono avere gli stessi valori per tutti gli attributi

Si noti la distinzione tra oggetti (istanze di classi) e valori (di un certo tipo):

– un oggetto è istanza di una classe ed ha vita propria

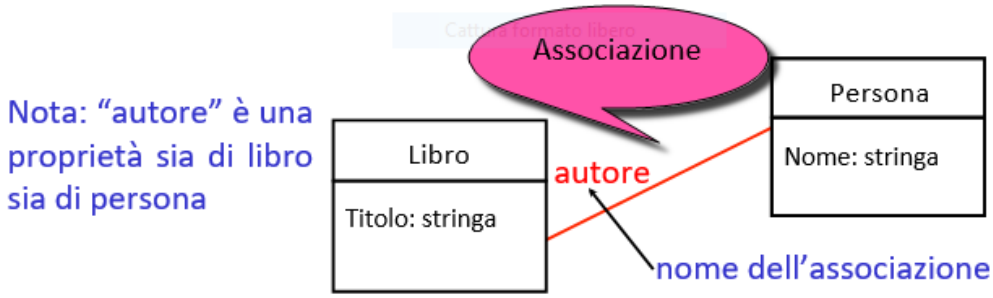
– un valore è un elemento di un tipo, ed ha senso solo se associato ad un oggetto tramite un attributo



Proprietà di classi: associazioni in UML

• Una associazione (o relazione) tra una classe C1 ed una classe C2 modella una relazione matematica tra l'insieme delle istanze di C1 e l'insieme delle istanze di C2

• Gli attributi modellano proprietà locali di una classe, le associazioni modellano proprietà che coinvolgono altre classi. Una associazione tra due classi modella una proprietà di entrambe le classi



Istanze di associazioni: link

• Le istanze di associazioni si chiamano link: se A è una associazione tra le classi C1 e C2, una istanza di A è un link tra due oggetti (in altre parole, una coppia), uno della classe C1 e l'altro della classe C2

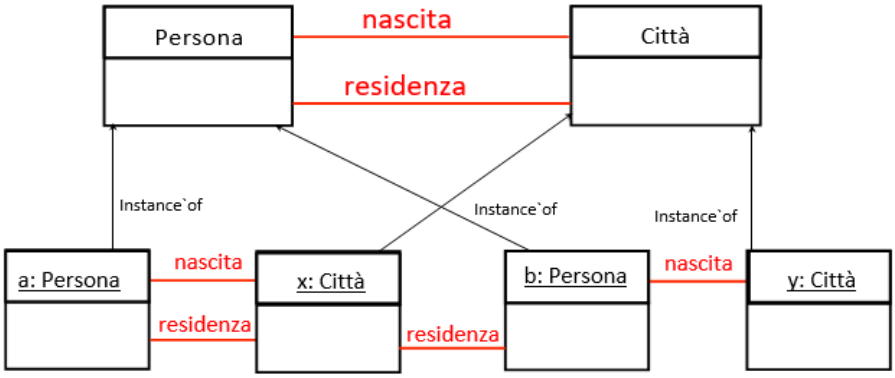
• Come gli oggetti sono istanze delle classi, così i link sono istanze delle associazioni (gli archi instance of non sono necessari)

• Al contrario degli oggetti, però, i link non hanno identificatori espliciti: un link è implicitamente identificato dalla coppia (o in generale dalla ennupla) di oggetti che esso rappresenta

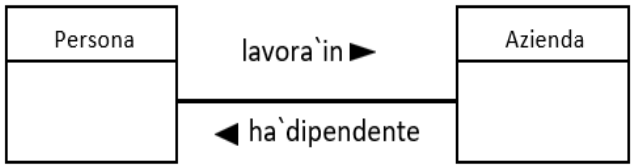
• Ovviamente, tra le stesse due classi possono essere definite più associazioni

• Attenzione: una relazione R tra C1 e C2 non dice nulla sul numero di link di R che coinvolgono due istanze delle classi C1 e C2.

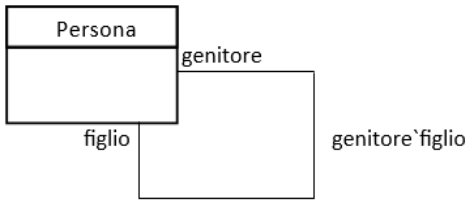
• Una istanza di Persona può essere legata a zero, una, o più istanze di Città da link di tipo "nascita" (se non specificato).



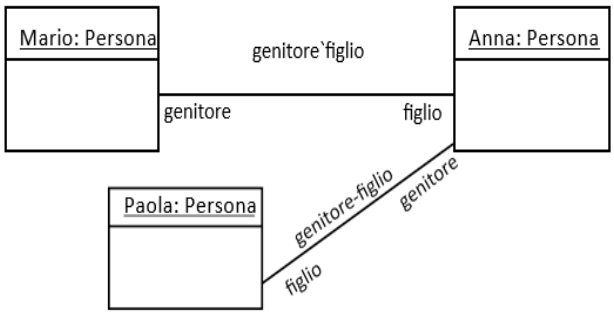
• Alcune volte è interessante specificare un verso per il nome della associazione
• Il verso non è una caratteristica del significato della associazione, ma dice semplicemente che il nome scelto per la associazione evoca un verso.



• Osserviamo ancora che le frecce che simboleggiano il verso non aggiungono nulla al significato della associazione (che formalmente si può considerare sempre una relazione matematica), ma aiutano ad
interpretare il senso dei nomi scelti per l'associazione
• È possibile aggiungere alla associazione una informazione che specifica il ruolo che una classe gioca nella associazione
• Il ruolo si indica con un nome posizionato lungo la linea che rappresenta l'associazione, vicino alla classe alla quale si riferisce
• Nell'esempio, dipendente è il ruolo che la persona gioca nell'associazione "lavora in" con Azienda
• Se nell'associazione A è indicato il ruolo giocato dalla classe C, tale ruolo sarà indicato (vicino alla corrispondente istanza di C) in ogni link che è istanza di A
• Analogamente al verso, il ruolo è generalmente opzionale, e non aggiunge nulla al significato dell'associazione
• L'unico caso in cui il ruolo è obbligatorio è quello in cui l'associazione insiste più volte sulla stessa classe, e rappresenta una relazione non simmetrica



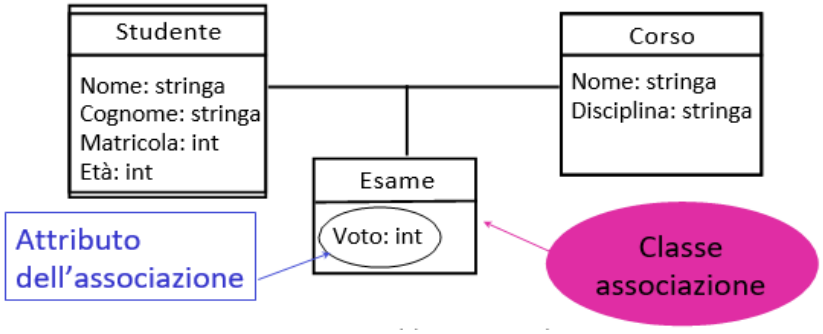
• Se non fossero indicati i ruoli nell'associazione "genitore-figlio", non sapremmo interpretare correttamente i link che sono istanze dell'associazione



• Anche nei casi in cui non è strettamente necessario, il ruolo può essere utile per aumentare la leggibilità del diagramma

Attributi di associazione:

Analogamente alle classi, anche le associazioni possono avere attributi. Formalmente, un attributo di una associazione è una funzione che associa ad ogni link che è istanza dell'associazione un valore di un determinato tipo
Esempio: Voto non è una proprietà né di Studente, né di Corso, ma della associazione Esame tra Studente e Corso



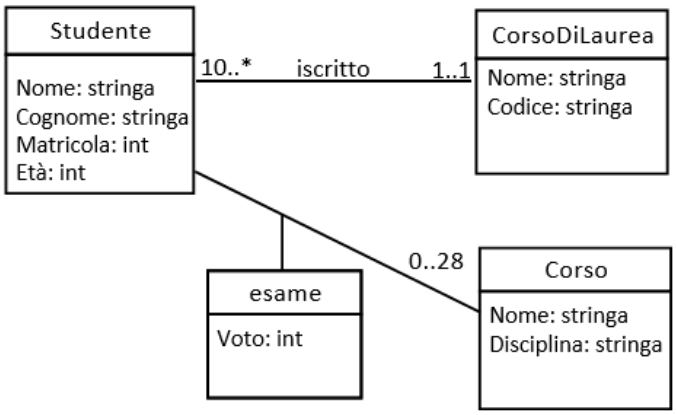
Molteplicità delle associazioni

• Per specificare con maggiore precisione il significato delle associazioni binarie (non ennarie) si possono definire i vincoli di molteplicità (o semplicemente molteplicità) delle associazioni
Esempio: ogni istanza di Persona deve essere legata ad esattamente una istanza (cioè ad almeno una e al massimo una) istanza di Città da link della associazione "nascita"

Molteplicità delle associazioni: notazione

• Le molteplicità si definiscono solo per le associazioni binarie
• Possibili molteplicità:

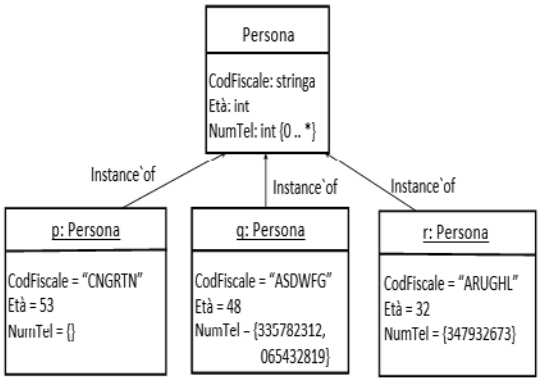
0 .. *	(nessun vincolo, si può evitare di indicare)
0 .. 1	(nessun limite per il minimo, e al massimo una)
1 .. *	(al minimo una, e nessun limite per il massimo)
1 .. 1	(esattamente una)
0 .. y	(nessun limite per il minimo, e al massimo y, con y intero > 0)
x .. *	(al minimo x, con x intero > 0, e nessun limite per il massimo)
x .. y	(al minimo x e al massimo y, con x ,y interi, x > 0 e y > x)



• Ogni studente è iscritto ad un corso di laurea
• Ogni corso di laurea ha almeno 10 iscritti
• Ogni studente può sostenere al massimo 28 esami

MOLTEPLICITA' DI ATTRIBUTI:

• Si possono specificare anche le molteplicità degli attributi. Se le molteplicità di un attributo B di tipo T di una classe C non vengono indicate, vuol dire che B associa ad ogni istanza di C esattamente un valore di T (come detto prima), che è equivalente a dire che la molteplicità è 1..1
• Al contrario, se un attributo B di tipo T di una classe C ha molteplicità x .. y, allora B associa ad ogni istanza di C al minimo x e al massimo y valori di tipo T
• Un attributo di tipo T della classe C con molteplicità diversa da {1..1} si dice **multivalore**, e formalmente non è una funzione totale, ma una relazione tra la classe C ed il tipo T
• Nelle istanze, il valore di un attributo multivalore si indica mediante un insieme.

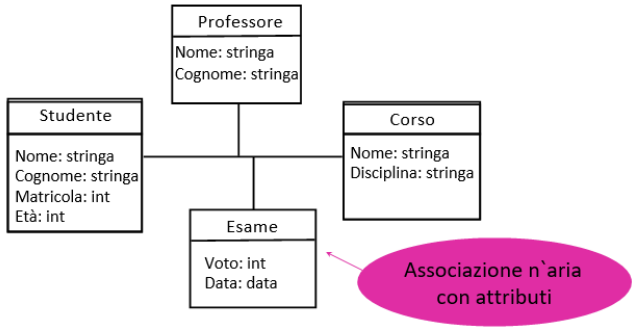


ASSOCIAZIONI N-ARIE:

Una associazione può essere definita su tre o più classi. In tale caso l'associazione si dice n-aria, e modella una relazione matematica tra n insiemi.

Ogni istanza di una associazione n-aria è un link n-ario, cioè che coinvolge n oggetti (è una ennupla).

Ovviamente, anche le associazioni n`arie possono avere attributi.

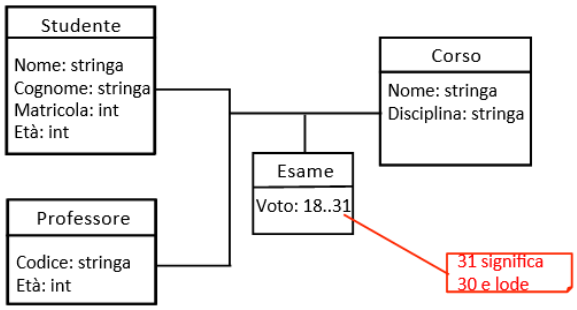


Link n-ari con attributi

I link che sono istanze di associazioni n-arie con attributi, hanno un valore per ogni attributo.

COMMENTI IN UML:

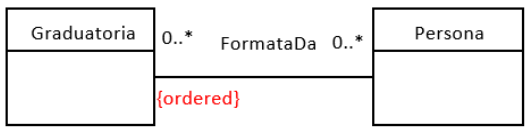
In UML, quando si vuole specificare una caratteristica che non è possibile rappresentare esplicitamente nel diagramma con i meccanismi visti finora, si può usare la nozione di commento.



Associazioni ordinate :

• In UML si può semplificare la descrizione utilizzando l'asserzione **{ordered}**

• {ordered} posto vicino a Graduatoria dice che data una istanza g di Graduatoria le istanze della associazione FormataDa che coinvolgono g sono ordinate (senza menzionare quale attributo utilizziamo per mantenere l'ordine)



• La soluzione con {ordered} e da preferire alla soluzione con un attributo esplicito "posizione" perché

– è più semplice (non fa uso di vincoli esterni espressi nei commenti) ed è quindi più leggibile

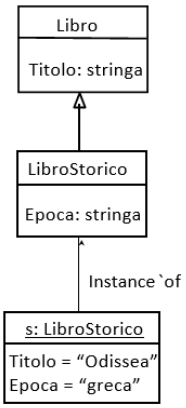
– astrae da come verrà mantenuta l'informazione sull'ordine evitando di introdurre uno specifico attributo ("posizione") necessario a questo scopo.

Generalizzazione in UML:

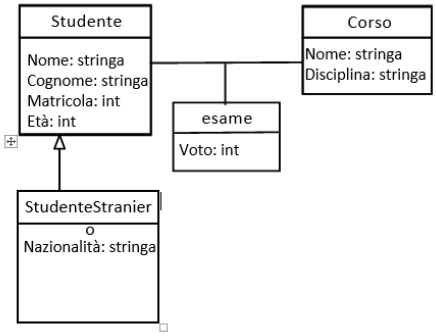
• Fino ad ora abbiamo assunto che due classi siano sempre disgiunte. In realtà sappiamo che può accadere che tra due classi sussista la relazione is-a, e cioè che ogni istanza di una sia anche istanza dell'altra.
• In UML la relazione is-a si modella mediante la nozione di generalizzazione
• La generalizzazione coinvolge una superclasse ed una o più sottoclassi (dette anche classi derivate). Il significato della generalizzazione è il seguente: ogni istanza di ciascuna sottoclasse è anche istanza della superclasse
• Quando la sottoclasse è una, la generalizzazione modella appunto la relazione is-a tra la sottoclasse e la superclasse

Ereditarietà in UML:

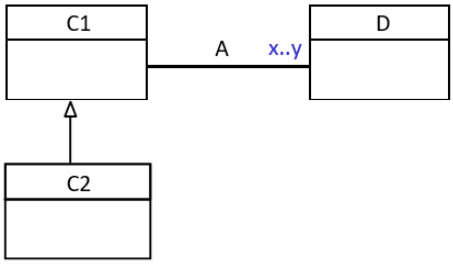
Ogni proprietà della superclasse è anche una proprietà della sottoclasse, e non si riporta esplicitamente nel diagramma.
1) Istanze
• s è istanza sia di LibroStorico (classe più specifica) sia di Libro
• non è più vero che due classi diverse sono disgiunte: Libro e LibroStorico non sono ovviamente disgiunte
• resta comunque vero che ogni istanza ha una ed una sola classe più specifica di cui è istanza; in questo caso la classe più specifica di s è LibroStorico
• s ha un valore per tutti gli attributi di LibroStorico, sia quelli propri, sia quelli ereditati dalla classe Libro



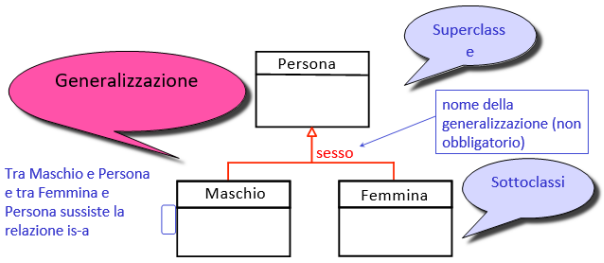
2) Ereditarietà sulle associazioni
• Ogni istanza di Studente può essere coinvolta in un numero qualunque di istanze della associazione "esame"
• Ogni istanza di StudenteStraniero è una istanza di Studente quindi
• Ogni istanza di StudenteStraniero può essere coinvolta in un numero qualunque di istanze della associazione "esame"



3) Ereditarietà sulle molteplicità
• Ogni istanza di C1 è coinvolta in al minimo x e al massimo y istanze dell'associazione A
• Ogni istanza di C2 è una istanza di C1 quindi
• Ogni istanza di C2 è coinvolta in al minimo x e al massimo y istanze dell'associazione A

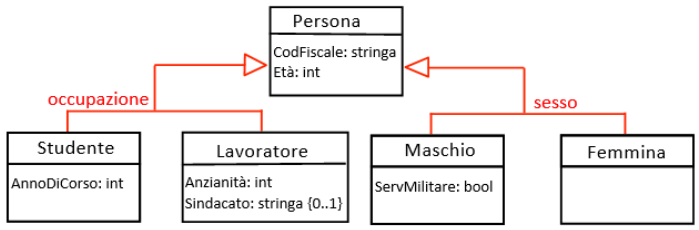


Finora, abbiamo considerato la generalizzazione come mezzo per modellare la relazione is-a tra due classi. La superclasse però può anche generalizzare diverse sottoclassi rispetto ad un unico criterio (che si può indicare con un nome, che è il nome della generalizzazione



Diverse generalizzazioni della stessa classe

La stessa superclasse può partecipare a diverse generalizzazioni. Concettualmente, non c'è alcuna correlazione tra due generalizzazioni diverse, perché rispondono a due criteri diversi di classificare le istanze della superclasse

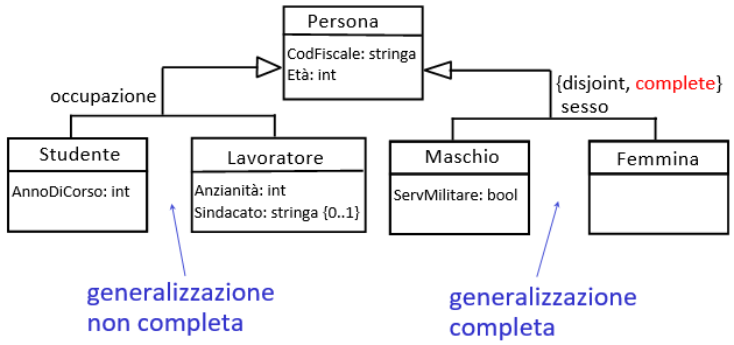


• Generalizzazioni disgiunte

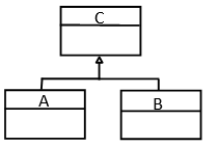
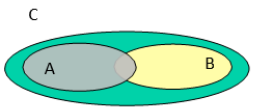
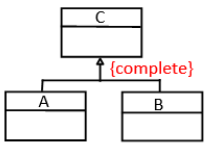
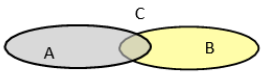
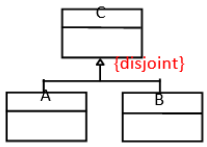
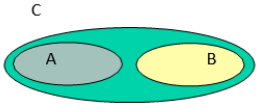
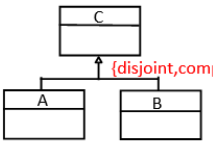
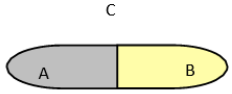
Una generalizzazione può essere disgiunta (le sottoclassi sono disgiunte a coppie) o no

• Generalizzazioni complete

Una generalizzazione può essere completa (l'unione delle istanze delle sottoclassi è uguale all'insieme delle istanze della superclasse) o no

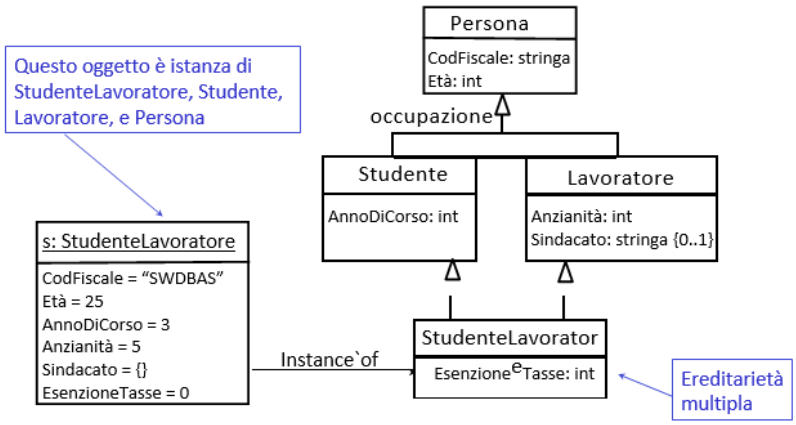


GENERALIZZAZIONI

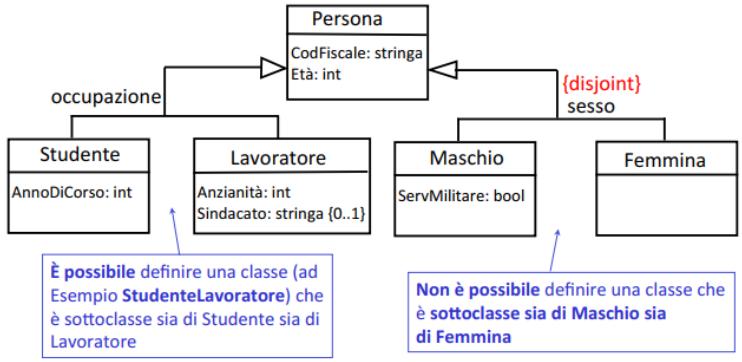
Livello intensionale	Livello estensionale	Livello intensionale	Livello estensionale
			
			

Ereditarietà multipla:

Attenzione: poiché un oggetto è istanza di una sola classe più specifica, due sottoclassi non disgiunte possono avere istanze comuni solo se hanno una sottoclasse comune (ereditarietà multipla)



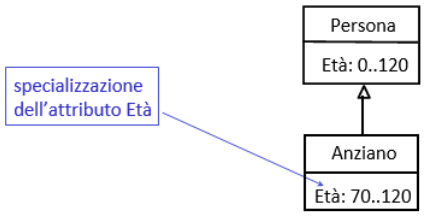
Da quanto detto, la differenza tra due classi mutuamente disgiunte e due classi non mutuamente disgiunte sta solo nel fatto che due classi disgiunte non possono avere sottoclassi comuni, mentre è possibile definire una classe come sottoclasse di due classi non disgiunte



Specializzazione di associazioni

In una generalizzazione la sottoclasse non solo può avere proprietà aggiuntive rispetto alla superclasse, ma può anche specializzare le proprietà ereditate dalla superclasse.

• **Specializzazione di un attributo**: Se una classe C1 ha un attributo A di tipo T1, e se C2 è una sottoclasse di C1, specializzare A in C2 significa definire A anche in C2 ed assegnargli un tipo T2 i cui valori sono un sottoinsieme dei valori di T1.

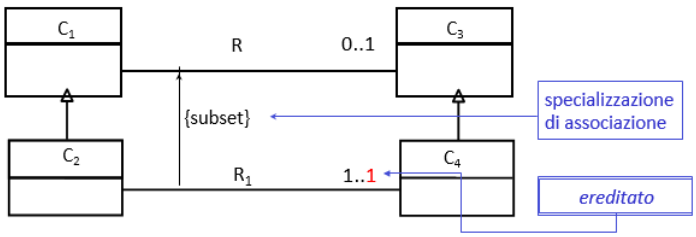


• **Specializzazione di una associazione**: Se una classe C1 partecipa ad una associazione R con un'altra classe C3, e se C2 è una sottoclasse di C1, specializzare R in C2 significa:

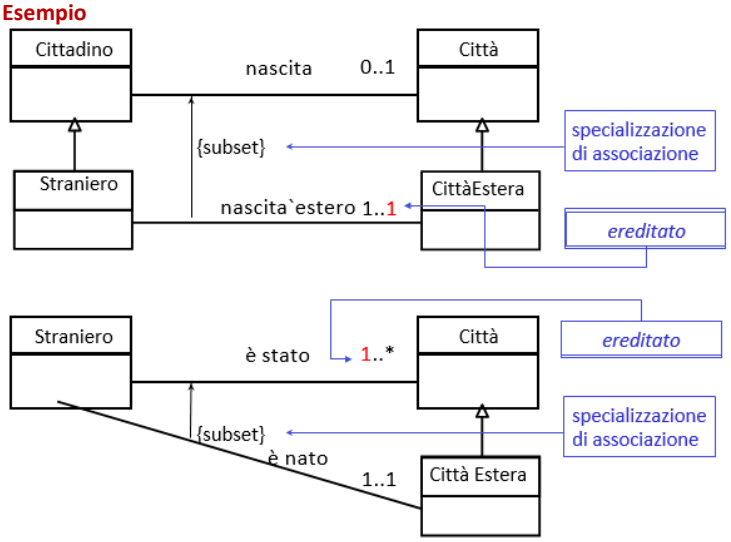
– Definire una nuova associazione R1 tra la classe C2 e una classe C4 che è sottoclasse di C3 (al limite C4 può essere la classe C3 stessa)

– Stabilire una dipendenza di tipo **{subset}** da R1 a R

– Definire eventualmente molteplicità più specifiche su R1 rispetto alle corrispondenti molteplicità definite su R (si noti che la molteplicità massima su R1 deve essere minore o uguale a quella su R).



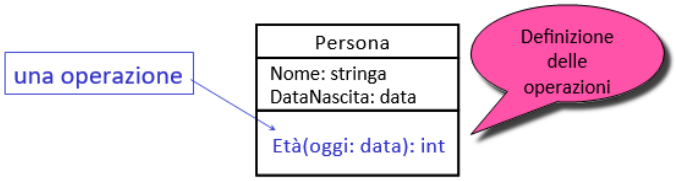
Esempio



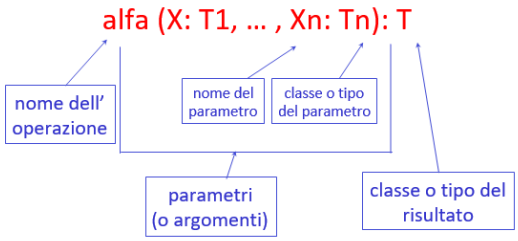
OPERAZIONI :

Finora abbiamo fatto riferimento solamente a proprietà statiche (attributi e associazioni) di classi. In realtà, le classi (e quindi le loro istanze) sono caratterizzate anche da proprietà dinamiche, che in UML si definiscono mediante le operazioni.
Una operazione associata ad una classe C indica che sugli oggetti della classe C si può eseguire una computazione, cioè una elaborazione (detta anche metodo),

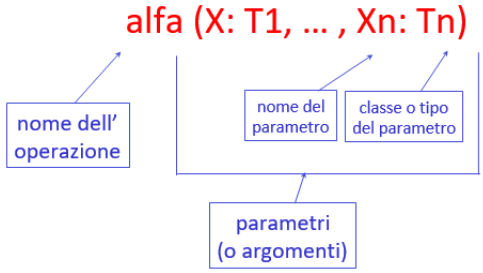
- o per calcolare le proprietà
- o per effettuare cambiamenti di stato (cioè per modificare le proprietà)



In una classe, una operazione si definisce specificando la segnatura (nome, parametri e il tipo dell'eventuale risultato) e non il **metodo** (cioè non la specifica di cosa fa l'operazione)



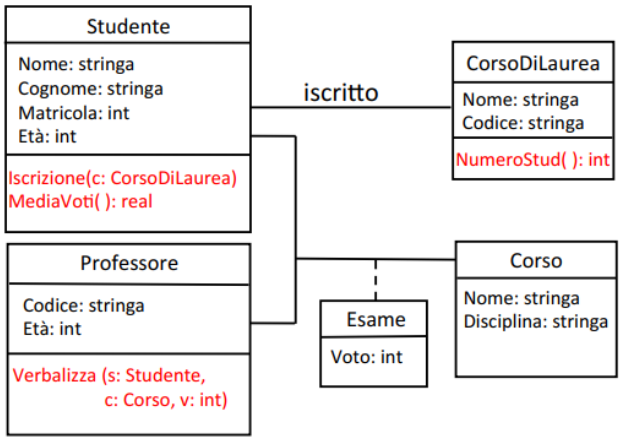
Non è necessario che una operazione restituisca un valore o un oggetto. Una operazione può anche solo effettuare azioni senza calcolare un risultato. In questo caso l'operazione si definisce così:



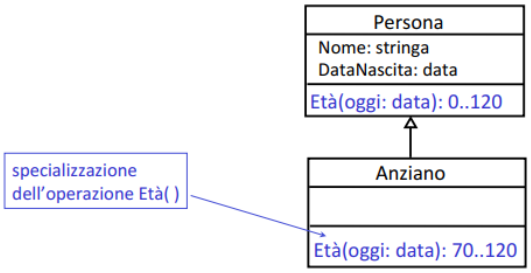
Osservazioni sulle operazioni

- Una operazione di una classe C è pensata per essere invocata facendo riferimento ad una istanza della classe C, chiamata oggetto di invocazione. Esempio di invocazione: p.Età(oggi) (dove p è un oggetto della classe Persona).
- In altre parole, nell'attivazione di ogni operazione, oltre ai parametri c'è sempre implicitamente in gioco un oggetto (l'oggetto di invocazione) della classe in cui l'operazione è definita.
- **Attenzione**: le operazioni che si definiscono sul modello di analisi sono le operazioni che caratterizzano concettualmente la classe.
- Altre operazioni, più orientate alla **realizzazione** del software (come ad esempio le operazioni che consentono di gestire gli attributi, ossia conoscerne o cambiarne il valore), **non devono** essere definite in questa fase.

Esempio per capire:
Tracciare il diagramma delle classi corrispondenti alle seguenti specifiche:
Si vogliono modellare gli studenti (con nome, cognome, numero di matricola, età), il corso di laurea in cui sono iscritti, ed i corsi di cui hanno sostenuto l'esame, con il professore che ha verbalizzato l'esame, ed il voto conseguito. Di ogni corso di laurea interessa il codice e il nome. Di ogni corso interessa il nome e la disciplina a cui appartiene (ad esempio: matematica, fisica, informatica, ecc.). Di ogni professore interessa codice ed età.
Al momento dell'iscrizione, lo studente specifica il corso di laurea a cui si iscrive.
Dopo l'effe0uazione di un esame, il professore comunica l'avvenuta verbalizzazione dell'esame con i dati relativi (studente, corso, voto).
La segreteria vuole periodicamente calcolare la media dei voti di uno studente, e il numero di studenti di un corso di laurea.



Specializzazione di operazioni:
Oltre agli attributi e alle associazioni, anche le operazioni si possono specializzare nelle sottoclassi. Una operazione si specializza specializzando i parametri e/o il tipo di ritorno.
Si noti che il metodo associato ad una operazione specializzata in una sottoclasse è in genere diverso dal metodo associato alla stessa operazione nella superclasse



Semantica dei diagrammi delle classi: riassunto

Concetto	Significato	Note
Oggetto	Elemento	*Ogni oggetto ha vita propria ed ha un unico identificatore*
Classe	Insieme di oggetti	*Insieme con operazioni*
Tipo	Insieme di valori	*Un valore non ha vita propria*
Attributo	Funzione (o relazione, se multivalore)	*Da classi (e associazioni) a tipi*
Associazione	Relazione	*Sottoinsieme del prodotto cartesiano*
Relazione is-a	Sottoinsieme	*Implica ereditarietà*
Generalizzazione disgiunta e completa	Partizione	*Le sottoclassi formano una partizione della superclasse*
Operazione	Computazione	*Le operazioni vengono definite nelle classi*

Aspetti metodologici nella costruzione del diagramma delle classi

Un metodo comunemente usato per costruire il diagramma delle classi prevede i seguenti passi:

- Individua le classi e gli oggetti di interesse
- Individua gli attributi delle classi
- Individua le associazioni tra classi
- Individua gli attributi delle associazioni
- Determina le molteplicità di associazioni e attributi
- Individua le generalizzazioni, partendo o dalla classe più generale e scendendo nella gerarchia, oppure dalle classi più specifiche e risalendo nella gerarchia
- Determina le specializzazioni
- Individua le operazioni ed associale alle classi
- Controllo di qualità

Scelta tra attributi e classi

La scelta deve avvenire tenendo presente le seguenti differenze tra classi e tipi:

	Classe	Tipo
Istanze	**oggetti**	**valore**
Istanze identificate da	**identificatore di oggetto**	**valore**
Uguaglianza	**basata su identificatore**	**basata su valore**
Realizzazione	**da progettare**	**tipicamente predefinita, oppure basata su strutture di dati predefinite**

Scelta tra classi e attributi

- Un concetto verrà modellato come una **classe** se:
  – le sue istanze hanno **vita propria**
  – se le sue istanze possono essere identificate **indipendentemente** da altri oggetti
  – se ha o si prevede che avrà delle **proprietà indipendenti** dagli altri concetti
  – se su di esso si **"predica"** nello schema concettuale
- Un concetto verrà modellato come un **attributo** se:
  – se le sue istanze **non hanno vita propria**
  – se ha senso solo per **rappresentare proprietà di altri concetti**
  – se **non si "predica"** su di esso nello schema

Scelta tra classi e associazioni

- Un concetto verrà modellato come una **classe** se:
  – le sue istanze hanno **vita propria**
  – se le sue istanze possono essere identificate **indipendentemente** da altri oggetti
  – se ha o si prevede che avrà delle **proprietà indipendenti** dagli altri concetti
- Un concetto verrà modellato come una **associazione** se:
  – se le sue istanze rappresentano **n-ple di altre istanze**
  – se non ha senso pensare alla partecipazione delle sue instanze ad **altre associazioni**

Verifiche sulle generalizzazioni

Il grafo delle generalizzazioni non può contenere cicli!

Verifiche sulle specializzazioni

- **Specializzazione di un attributo**: se una classe C1 ha un attributo A di tipo T1, se C2 è una sottoclasse di C1, e se A è specializzato in C2, allora il tipo assegnato ad A in C2 deve essere un tipo T2 i cui valori sono un **sottoinsieme** dei valori di T1.
- **Specializzazione di una associazione**: se una classe C1 partecipa ad una associazione R con un'altra classe C3, se C2 è una sottoclasse di C1, ed R è specializzata in C2 in una associazione R1 con C4 allora:
  – tra R1 ed R deve esserci una **dipendenza di tipo {subset}**
  – per R1 deve essere definita una molteplicità massima **uguale o più ristretta** che per R
  – C4 è una sottoclasse di C3 (al limite C3 e C4 sono uguali)

LEZIONE 2 – SLIDE 2

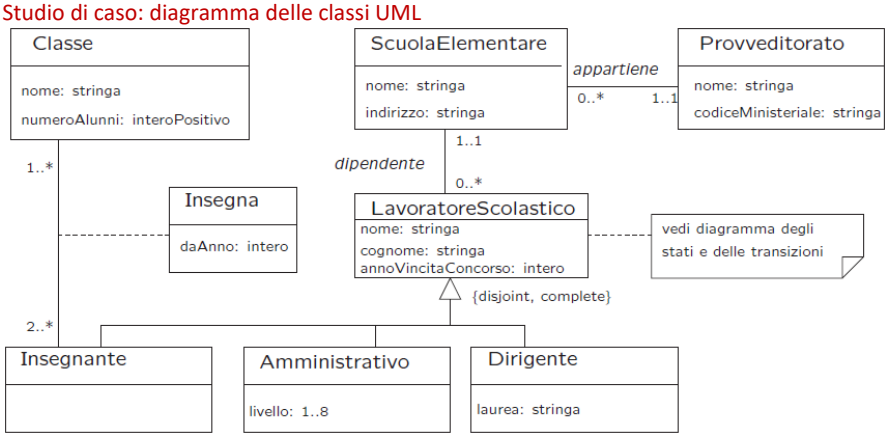
Progetto e realizzazione:
Si occupa del come l'applicazione dovrà realizzare le sue funzioni:
– definire l'architettura del programma,
– scegliere le strutture di rappresentazione,
– produrre la documentazione,
– scrivere il codice del programma.

Input alla fase di progetto:
È l'output della fase di analisi, ed è costituito da:
• lo schema concettuale, formato da:
– diagramma delle classi e degli oggetti;
– diagramma delle attività;
– diagramma degli stati e delle transizioni;
• la specifica (formale) delle operazioni.

Output della fase di progetto:
È l'input della fase di realizzazione, ed è costituito da:
– scelta delle classi UML che hanno responsabilità sulle associazioni;
– scelta/progetto delle strutture di dati;
– scelta della corrispondenza fra tipi UML e Java;
– scelta della gestione delle proprietà di una classe UML: immutabili, note alla nascita, ...
– progetto della API delle principali classi Java,

Esempio - Studio di caso: scuola elementare
Requisiti: L'applicazione da progettare riguarda le informazioni su provveditorati scolastici, scuole elementari e lavoratori scolastici. Di ogni scuola elementare interessa il nome, l'indirizzo e il provveditorato di appartenenza. Di ogni provveditorato interessa il nome e il codice attribuitogli dal Ministero. Dei lavoratori scolastici interessa la scuola elementare di cui sono dipendenti, il nome, il cognome e l'anno di vincita del concorso.
Esistono solamente tre categorie di lavoratori scolastici, che sono fra loro disgiunte: dirigenti, amministrativi e insegnanti. Dei primi interessa il tipo di laurea che hanno conseguito, dei secondi il livello (intero compreso fra 1 e 8), mentre dei terzi interessano le classi in cui insegnano, e, per ogni classe, da quale anno.
Ogni insegnante insegna in almeno una classe, e ogni classe ha almeno due insegnanti. Di ogni classe interessa il nome (ad es. "IV A") e il numero di alunni.
Il Ministero dell'Istruzione deve poter effettuare, come cliente della nostra applicazione, dei controlli sull'insegnamento. A questo scopo, si faccia riferimento alle seguenti operazioni:
– data una classe e l'anno corrente, calcolare quanti insegnanti della classe hanno vinto il concorso da più di 15 anni;
– dato un insegnante, calcolare il numero totale di alunni a cui insegna;
– dato un insieme di insegnanti, calcolare il numero medio di alunni a cui insegnano.

Studio di caso: diagramma delle classi UML



Progetto:responsabilità sulle associazioni
Prima di realizzare una classe UML che è coinvolta in un'associazione, ci dobbiamo chiedere se la classe ha responsabilità sull'associazione.
Diciamo che una classe C ha responsabilità sull'associazione A, quando, per ogni oggetto x che è istanza di C vogliamo poter eseguire opportune operazioni sulle istanze di A a cui x partecipa, che hanno lo scopo di:
– conoscere l'istanza (o le istanze) di A alle quali x partecipa,
– aggiungere una nuova istanza di A alla quale x partecipa,
– cancellare una istanza di A alla quale x partecipa,
– aggiornare il valore di un attributo di una istanza di A alla quale x partecipa.

Studio di caso: responsabilità

Prendiamo in considerazione il criterio 1.

– Di ogni scuola elementare interessa [. . .] il provveditorato di appartenenza.
→ ScuolaElementare ha responsabilità su appartiene.
– Dei lavoratori scolastici interessa la scuola elementare di cui sono dipendenti.
→ LavoratoreScolastico ha responsabilità su dipendente.
– [Degli insegnanti] interessano le classi in cui insegnano.
→ Insegnante ha responsabilità su insegna.

Prendiamo in considerazione il criterio 2.

– Prendiamo in considerazione l'operazione NumeroInsegnantiDaAggiornare. E` evidente che per la sua realizzazione è necessario che, a partire da un oggetto c che è istanza di Classe possiamo conoscere le istanze di insegna alle quali c partecipa.
→ Classe ha responsabilità su insegna.
– Prendiamo in considerazione le operazioni NumeroAlunniPerDocente e NumeroMedioAlunniPerDocente.
È evidente che per la loro realizzazione è necessario che, a partire da un oggetto i che è istanza di Insegnante possiamo conoscere le istanze di insegna alle quali i partecipa.
→ Insegnante ha responsabilità su insegna.
Si noti che eravamo già a conoscenza di questa responsabilità.

Molteplicità e responsabilità

• L'esistenza di alcuni vincoli di molteplicità di associazione ha come conseguenza l'esistenza di responsabilità su tali associazioni.
• In particolare, quando una classe C partecipa ad un'associazione A con un vincolo di:
– molteplicità massima finita, oppure
– molteplicità minima diversa da zero,
la classe C ha necessariamente responsabilità su A.
• Il motivo, che sarà ulteriormente chiarito nella fase di realizzazione, risiede nella necessità di poter interrogare gli oggetti della classe C sul numero di link di tipo A a cui partecipano, al fine di verificare il soddisfacimento del vincolo di molteplicità.

Studio di caso: responsabilità

Prendiamo in considerazione il criterio 3.
– Esiste un vincolo di molteplicità minima diversa da zero (1..*) nell'associazione insegna.
→ Classe ha responsabilità su insegna.
Si noti che eravamo già a conoscenza di questa responsabilità.
• Anche tutte le altre responsabilità determinate in precedenza possono essere evinte utilizzando questo criterio.

Studio di caso: tabella delle responsabilità

Possiamo riassumere il risultato delle considerazioni precedenti nella seguente tabella delle responsabilità.

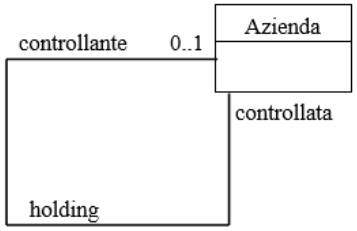
Associazione	Classe	ha resp.
insegna	*Classe*	SÌ[2,3]
	Insegnante	SÌ[1,2,3]
dipendente	*ScuolaElementare*	NO
	LavoratoreScolastico	SÌ[1,3]
appartiene	*ScuolaElementare*	SÌ[1,3]
	Provveditorato	NO

1. dai requisiti
2. dalle operazioni
3. dai vincoli di molteplicità

Criterio di verifica: per ogni associazione deve esserci almeno un "SI".

Responsabilità dei ruoli

Quanto detto vale anche per il caso in cui l'associazione coinvolga più volte la stessa classe. In questo caso il concetto di responsabilità si attribuisce ai ruoli, piuttosto che alle classi.



Ad esempio, la classe Azienda potrebbe avere la responsabilità sull'associazione holding, solo nel ruolo controllata. Questo significa che, dato un oggetto x della classe Azienda, vogliamo poter eseguire operazioni su x per conoscere l'eventuale azienda controllante, per aggiornare l'azienda controllante, ecc.

Scelta delle strutture di dati

• Prendendo in considerazione:
– il diagramma delle classi,
– la tabella delle responsabilità,
– gli argomenti delle operazioni e i loro valori restituiti,
è possibile determinare se si avrà bisogno di strutture per la rappresentazione dei dati della nostra applicazione.
• Facendo riferimento allo studio di caso, notiamo che:
– poiché la classe Insegnante ha responsabilità sulla associazione insegna, la cui molteplicità è 1..*, per la realizzazione di quest'ultima avremo bisogno di rappresentare insiemi di link;
– lo stesso si può dire prendendo in considerazione la classe Classe;
– per rappresentare l'input dell'operazione NumeroMedioAlunniPerDocente avremo bisogno di un opportuno insieme di insegnanti.
• In generale, emerge la necessità di rappresentare collezioni omogenee di oggetti.
• Per fare ciò, utilizzeremo il collection framework di Java che, attraverso l'uso dei generics permette di realizzare collezioni omogenee, in particolare:
– Set<Elem>, HashSet<Elem>, . . . ,
– List<Elem>, LinkedList<Elem>, . . . ,

Corrispondenza fra tipi UML e Java

• Prendendo in considerazione:
– il diagramma delle classi,
– la specifica delle operazioni,
– la scelta delle strutture di dati,
è possibile compilare una lista dei tipi UML per i quali dobbiamo decidere la rappresentazione in Java.
• In generale, per la rappresentazione è opportuno scegliere un tipo base Java (int, float, . . .) o una classe Java di libreria (String, . . .) ogni volta ci sia una chiara corrispondenza con il tipo UML

Studio di caso: tabella di corrispondenza dei tipi UML

Tipo UML	Rappresentazione in Java
intero	`int`
interoPositivo	`int`
1..8	`int`
reale	`double`
stringa	`String`
Insieme	`Set`

Corrispondenza fra tipi UML e Java

Per due casi servono ulteriori considerazioni:
1. quando il tipo UML è necessario per un attributo di classe con una sua molteplicità (ad es., NumeroTelefonico: Stringa {0..∗});
2. quando non esiste in Java un tipo base o una classe predefinita che corrisponda chiaramente al tipo dell'attributo UML (ad es., Indirizzo).

Ulteriori considerazioni: caso 1

• In questo caso, coerentemente con la scelta precedente relativa alle strutture di dati, scegliamo di rappresentare l'attributo mediante una classe Java per collezioni omogenee di oggetti, come Set.
• Va notato che tali classi non consentono la rappresentazione di insiemi di valori di tipi base (int, float, etc.), ma solamente di oggetti.
• Di conseguenza, per la rappresentazione dei valori atomici dobbiamo utilizzare classi Java di libreria, quali Integer, Float, etc.
• Ad esempio, per rappresentare un ipotetico attributo AnniVincitaConcorsi: intero {0..∗}, useremo un oggetto costruito mediante un'espressione come: HashSet<Integer> s = new HashSet<Integer>();

Ulteriori considerazioni: caso 2

•Per quanto riguarda la realizzazione di tipi UML tramite classi Java, notiamo che, dal punto di vista formale, il tipo UML andrebbe specificato ulteriormente tramite le operazioni previste per esso. (non propriamente trattati in questo corso)
• L'approccio seguito è simile a quello della realizzazione di classi UML con alcune regole da seguire per le funzioni speciali:
toString(): si può prevedere di farne overriding, per avere una rappresentazione testuale dell'oggetto.
equals(): è necessario fare overriding della funzione equals() ereditata dalla classe Object.
Infatti due valori sono uguali solo se sono lo stesso valore, e quindi il comportamento di default della funzione equals() non è corretto.
hashCode(): è necessario fare overriding della funzione hashCode() ereditata dalla classe Object.
Infatti in Java deve sempre valere il principio secondo il quale se due oggetti sono uguali secondo equals() allora questi devono avere lo stesso codice di hash secondo hashCode(). Quindi poichè ridefiniamo equals() dobbiamo anche ridefinire coerentemente a detto principio hashCode().

clone(): ci sono due possibilità:
1. Nessuna funzione della classe Java effettua side-effect. In questo caso, clone() non si ridefinisce (gli oggetti sono immutabili).
2. Qualche funzione della classe Java effettua side-effect. In questo caso, poichè i moduli clienti hanno tipicamente la necessità di copiare valori (ad esempio, se sono argomenti di funzioni) si mette a disposizione la possibilità di copiare un oggetto, rendendo disponibile la funzione clone() (facendo overriding della funzione protected ereditata da Object).

Realizzazione di tipi UML

• A titolo di esempio, vediamo la realizzazione del tipo UML Data, inteso come aggregato di un giorno, un mese ed un anno validi secondo il calendario gregoriano, e per cui le operazioni previste sono:
– selezione del giorno, del mese e dell'anno;
– verifica se una data sia precedente ad un'altra;
– avanzamento di un giorno.
• Realizziamo il tipo UML Data mediante la classe Java Data, rappresentando il giorno, il mese e l'anno come campi dati private di tipo int.
• Scegliamo di realizzare l'operazione di avanzamento di un giorno mediante una funzione Java che fa side-effect sull'oggetto di invocazione.

Esempio

```
// File Tipi/Data.java
public class Data implements Cloneable {
    // SERVE LA RIDEFINIZIONE DI clone(), in quanto una funzione fa side-effect
    public Data() {
        giorno = 1;
        mese = 1;
        anno = 2000;
    }
    public Data(int a, int me, int g) {
        giorno = g;
        mese = me;
        anno = a;
        if (!valida()) {
        giorno = 1;
        mese = 1;
        anno   = 2000;
      }
    }
    public int giorno() {
        return giorno;
    }
    public int mese() {
        return mese;
    }
    public int anno() {
        return anno;
    }
    public boolean prima(Data d) {
        return ((anno < d.anno) || (anno == d.anno && mese < d.mese) ||
        (anno == d.anno && mese == d.mese && giorno < d.giorno));
    }
    public void avanzaUnGiorno() {
        // FA SIDE-EFFECT SULL'OGGETTO DI INVOCAZIONE
        if (giorno == giorniDelMese())
           if (mese == 12) {
              giorno = 1;
              mese = 1;
              anno++;
            }
           else {
              giorno = 1;
              mese++;
            }
       else giorno++;
}
```

```
    public String toString() {
        return giorno + "/" + mese + "/" + anno;
    }

    public Object clone() {
        try {
            Data d = (Data)super.clone();
            return d;
        } catch (CloneNotSupportedException e) {
            // non può accadere, ma va comunque gestito
            throw new InternalError(e.toString());
        }
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            Data d = (Data)o;
            return d.giorno == giorno && d.mese == mese && d.anno == anno;
        }
        else return false;
    }
    public int hashCode() {
        return giorno + mese + anno;  //possiamo naturalmente realizzare una
                                      //funzione di hash più sofisticata
    }

    // CAMPI DATI
    private int giorno, mese, anno;

    // FUNZIONI DI SERVIZIO
    private int giorniDelMese() {
          switch (mese) {
          case 2:
             if (bisestile()) return 29;
             else return 28;
          case 4: case 6: case 9: case 11: return 30;
          default: return 31;
          }
     }
     private boolean bisestile() {
         return ((anno % 4 == 0) && (anno % 100 != 0)) || (anno % 400 == 0);
     }
     private boolean valida() {
         return anno > 0 && anno < 3000 && mese > 0 && mese < 13 && giorno > 0 && giorno <= giorniDelMese();
     }
}
```

Esempio 2

È possibile realizzare l'operazione di avanzamento di un giorno senza fare side-effect sull'oggetto di invocazione, ovvero nella cosiddetta maniera funzionale.
Ad esempio, ciò è possibile prevedendo un metodo Java pubblico con la seguente dichiarazione.

```
// File Tipi/DataFunzionale.java public class DataFunzionale {
// ...
     public static DataFunzionale unGiornoDopo(DataFunzionale d) {
     // NON FA SIDE-EFFECT
     // ...
}
```

Si noti che la funzione è statica e riceve l'input tramite il suo argomento, e non tramite l'oggetto di invocazione.
Realizzare la funzione Java unGiornoDopo() in maniera che restituisca il giorno successivo al suo argomento, senza modificare quest'ultimo.

Gestione delle precondizioni

• Nella fase precedente è possibile che sia stato scelto un tipo o classe Java semanticamente più esteso del corrispondente tipo UML, ovvero che ha dei valori **non ammessi** per quest'ultimo.
Ad esempio, abbiamo scelto il tipo Java int per la rappresentazione dell'attributo
livello: 1..8 della classe Amministrativo).
• Vedremo che una situazione simile si ha quando un'operazione ha **precondizioni**. Ad esempio, l'operazione NumeroMedioAlunniPerDocente ha la precondizione che il suo argomento non sia l'insieme vuoto.
• In tali casi si pone il problema di assicurare che i valori usati nei parametri attuali di varie funzioni Java siano coerenti con i valori ammessi per il tipo UML, ad esempio:
– che il parametro attuale del costruttore della classe Java Amministrativo sia compreso fra 1 e 8),
– che il parametro attuale della funzione Java che realizza l'operazione NumeroMedioAlunniPerDocente non sia l'insieme vuoto.

Verifica nel lato client

Con questo approccio è sempre il cliente a doversi preoccupare che siano verificate le condizioni di ammissibilità.
Come esempio, facciamo riferimento alla classe Java **Amministrativo** per lo studio di e ad un suo potenziale cliente che ha la necessità di creare un oggetto.

```
// File Precondizioni/LatoClient/Amministrativo.java
public class Amministrativo {
    private int livello;
    public Amministrativo(int l) { livello = l; }
    public int getLivello() { return livello;  }
    public void setLivello(int l) { livello = l; }
    public String toString() {
        return " (livello = " + livello + ")";
    }
}

// File Precondizioni/LatoClient/ClientAmministrativo.java
public class ClientAmministrativo {
    public static void main(String[] args) {
        Amministrativo giovanni =  null;
        boolean ok = false;
        while (!ok)  {
            System.out.println("Inserisci livello");
            int livello = InOut.readInt();
            if (livello >= 1 && livello <= 8) {  // CONTROLLO PRECONDIZIONI
               giovanni = new Amministrativo(livello);
               ok = true;
            }
        }
        System.out.println(giovanni);
    }
}
```

Problemi dell'approccio lato client

Con tale approccio, il cliente ha bisogno di un certo grado di conoscenza dei meccanismi di funzionamento della classe, il che potrebbe causare un aumento dell'accoppiamento.
Inoltre, il controllo delle precondizioni verrà duplicato in ognuno dei clienti, con indebolimento dell'estendibilità e della modularità.
Per questo motivo, un altro approccio tipico prevede che sia la classe a doversi preoccupare della verifica delle condizioni di ammissibilità (si tratta, in altre parole, di un approccio lato server).
In tale approccio, le funzioni della classe lanceranno un'eccezione nel caso in cui le condizioni non siano rispettate. Il cliente intercetterà tali eccezioni, e intraprenderà le opportune azioni.

Verifica nel lato server

In questo approccio, quindi:
• Va definita un'opportuna classe (derivata da Exception) che rappresenta le eccezioni sulle precondizioni. La classe tipicamente farà overriding di toString(), per poter stampare un opportuno messaggio.
• Nella classe server, le funzioni devono lanciare (mediante il costrutto throw) eccezioni nel caso in cui le condizioni di ammissibilità non siano verificate.
• La classe client deve intercettare mediante il costrutto try catch (o rilanciare) l'eccezione, e prendere gli opportuni provvedimenti.

Esempio

```
// File Precondizioni/LatoServer/EccezionePrecondizioni.java
public class EccezionePrecondizioni extends Exception {
    private String messaggio;
    public EccezionePrecondizioni(String m) {
        messaggio = m;
    }
    public EccezionePrecondizioni() {
        messaggio = "Si e' verificata una violazione delle precondizioni";
    }
    public String toString() {
        return messaggio;
    }
}

// File Precondizioni/LatoServer/Amministrativo.java
public class Amministrativo {
    private int livello;
    public Amministrativo(int l) throws EccezionePrecondizioni {
        if (l < 1 || l > 8)          // CONTROLLO PRECONDIZIONI
          throw new EccezionePrecondizioni("Il livello deve essere compreso fra 1 e 8");
        livello = l;
    }
    public int getLivello() { return livello; }
    public void setLivello(int l) throws EccezionePrecondizioni {
        if (l < 1 || l > 8)  // CONTROLLO PRECONDIZIONI
           throw new EccezionePrecondizioni();
    livello = l;
    }
    public String toString() {
        return " (livello = " + livello + ")";
    }
}

// File Precondizioni/LatoServer/ClientAmministrativo.java
public class ClientAmministrativo {
    public static void main(String[] args) {
        Amministrativo giovanni =  null;
        boolean ok = false;
        while (!ok)  {
            System.out.println("Inserisci livello");
            int livello = InOut.readInt();
            try {
              giovanni =  new Amministrativo(livello);
              ok = true;
            } catch (EccezionePrecondizioni e) { System.out.println(e);
            }
        }
        System.out.println(giovanni);
     }
}
```

Verifica Precondizioni

• Riassumendo, per la verifica della coerenza dei parametri attuali delle funzioni Java nel contesto di:
– precondizioni delle operazioni,
– tipi UML rappresentati con tipi Java che hanno valori non ammessi,
in fase di progetto dobbiamo scegliere l'approccio lato client oppure quello lato server.
• Fatta salva la migliore qualità (come evidenziato in precedenza) dell'approccio lato server, per pure esigenze di compattezza del codice mostrato, nella quarta parte del corso adotteremo l'approccio lato client.

Gestione delle proprietà di classi UML

• In generale le proprietà (attributi e associazioni) di un oggetto UML evolvono in maniera arbitraria durante il suo ciclo di vita.
• Esistono però alcuni casi particolari che vanno presi in considerazione nella fase di progetto.

Definiamo una proprietà:
– **non nota alla nascita**, se non è completamente specificata nel momento in cui nasce l'oggetto;
– **immutabile**, se, una volta che è stata specificata, rimane la stessa per tutto il ciclo di vita dell'oggetto.
Alcuni esempi relativi allo studio di caso:
proprietà immutabile: attributo nome della classe UML LavoratoreScolastico;
proprietà mutabile: associazione insegna nel ruolo di Insegnante;
proprietà non nota alla nascita: associazione dipendente nel ruolo di ScuolaElementare;
proprietà nota alla nascita: associazione appartiene nel ruolo di ScuolaElementare.

Assunzioni di default

• Distinguiamo innanzitutto fra:
– proprietà **singole**, ovvero attributi (di classe o di associazione) e associazioni (o meglio, ruoli) con molteplicità 1..1;
– proprietà **multiple**, tutte le altre.
• Le nostre assunzioni di default, ovvero che valgono in assenza di ulteriori elementi, sono le seguenti:
– tutte le proprietà sono **mutabili**;
– le proprietà singole sono **note alla nascita**;
– le proprietà multiple **NON sono note alla nascita**.

Ovviamente, in presenza di motivi validi, possiamo operare scelte diverse da quelle di default.
Riassumeremo tutte le nostre scelte differenti da quelle di default mediante la tabella delle proprietà immutabili e la tabella delle assunzioni sulla nascita.

Classe UML	Proprietà immutabile
ProvveditoratO	*nome*
LavoratoreScolastico	*nome*
	cognome
	annoVincitaConcorso
Dirigente	*laurea*
ScuolaElementare	*appartiene*

Classe UML	Proprietà	
	nota alla nascita	non nota alla nascita
LavoratoreScolastico	–	*dipendente*

Sequenza di nascita degli oggetti

Sono necessari alcuni commenti sulla seconda tabella appena mostrata.
• In generale, non possiamo dire nulla sull'ordine in cui gli oggetti nascono. Ad esempio, facendo riferimento allo studio di caso, non sappiamo se nasceranno prima gli oggetti di classe Insegnante o quelli di classe Classe.
• L'assunzione che quando nasce un oggetto Java corrispondente ad una scuola elementare sia noto il suo provveditorato di appartenenza è ragionevole poiché le responsabilità su **appartiene** è singola, e la molteplicità è 1..1. Questa assunzione implica che nascano prima i provveditorati delle scuole elementari.
• Viceversa, quando nasce un oggetto Java corrispondente ad un lavoratore scolastico non assumiamo che sia nota la scuola elementare di cui è dipendente.

Valori alla nascita

Per tutte le proprietà che sono note alla nascita potremmo chiederci se per esse esiste un valore di default (valido per tutti gli oggetti) oppure no.
Ad esempio:
• l'attributo nome della classe UML LavoratoreScolastico è noto alla nascita dell'oggetto, ed è in generale diverso per oggetti distinti;
• nell'ipotesi di aggiungere alla classe LavoratoreScolastico l'attributo intero noteDiDemerito, potremmo assumere che sia noto alla nascita, che il valore iniziale sia 0 per tutti gli oggetti, e che sia mutabile.
Queste informazioni potrebbero essere rappresentate mediante un'opportuna tabella.

Api delle classi Java progettate

• Prendendo in considerazione:
– il diagramma delle classi,
– la tabella di corrispondenza fra tipi UML e rappresentazione in Java,
– la tabella delle proprietà immutabili,
– la tabella delle assunzioni sulla nascita,
è possibile dare l'interfaccia pubblica (API) di molte delle classi Java che realizzeremo nella fase successiva.
In particolare, possiamo dare le API per le classi Java corrispondenti alle classi UML. Per tali classi dobbiamo prevedere le seguenti categorie di funzioni Java:
• costruttori;
• funzioni per la gestione degli attributi;
• funzioni per la gestione delle associazioni;
• funzioni corrispondenti alle operazioni di classe;
• funzioni per la gestione degli eventi e altro;
• funzioni di servizio (ad es., per la stampa).

Ad esempio, possiamo definire la API della classe Java ScuolaElementare nel seguente modo (per maggiori dettagli e per le classi corrispondenti alle associazioni rimandiamo alla parte successiva del corso, in quanto servono ulteriori considerazioni).

```
public class ScuolaElementare {
// COSTRUTTORI
/** specificare, se opportuno, il significato degli argomenti, ad esempio in formato utile per javadoc */
      public ScuolaElementare (String nome, String indirizzo, Provveditorato appartiene) { };
// GESTIONE ATTRIBUTI
      public String getNome(){return null;};
      public String getIndirizzo(){return null;};
      public void  setIndirizzo(String  i){};
      public void setNome(String n){};
// GESTIONE ASSOCIAZIONI
      // - appartiene
      public Provveditorato getProvveditorato(){return null;};
     //  -  dipendente: non ha responsabilità
// OPERAZIONI DI CLASSE
    // assenti
// GESTIONE EVENTI
   // assenti
// STAMPA
      public String toString(){return null;};
}
```

Struttura dei file e dei package

• Prendendo in considerazione le regole di visibilità di Java e le esigenze di **information hiding**, possiamo definire la cosiddetta struttura dei file, dei direttori e dei package, che costituisce un ulteriore aspetto progettuale relativo all'architettura dei moduli software della nostra applicazione.
• In generale, sceglieremo di disporre tutti i file dell'applicazione in un package Java P, nel direttorio P.
• Saranno presenti opportuni sottodirettori e sottopackag.

LEZIONE 2 – SLIDE 3

La fase di realizzazione

La fase di realizzazione si occupa di scrivere il **codice** del programma, e produrre parte della **documentazione**.
Il suo input è costituito dall'output della fase di **analisi**, e l'output della fase di **progetto.**

Traduzione in Java del diagramma delle classi

Nell'esposizione di questo argomento, seguiremo quest'ordine:
1. realizzazione di singole classi,
2. realizzazione delle associazioni,
3. realizzazione delle generalizzazioni.

Realizzazione di una classe UML con soli attributi

Assumiamo, per il momento, che la molteplicità di tutti gli attributi sia 1..1.
• Gli attributi della classe UML diventano campi privati (o protetti) della classe Java, gestiti da opportune funzioni pubbliche:
– la funzione **get** serve a restituire al cliente il valore dell'attributo;
– la funzione **set** consente al cliente di cambiare il valore dell'attributo.
• I tipi Java per gli attributi vanno scelti secondo la tabella di corrispondenza dei tipi UML prodotta durante la fase di progetto.
• Si sceglie un opportuno valore iniziale per ogni attributo:
– affidandosi al valore di default di Java, oppure
– fissandone il valore nella dichiarazione (se tale valore iniziale va bene per tutti gli oggetti), oppure
– facendo in modo che il valore iniziale sia fissato, oggetto per oggetto, mediante un costruttore.
• Per quegli attributi per i quali non ha senso prevedere di cambiare il valore (secondo la tabella delle proprietà immutabili prodotta durante la fase di progetto), non si definisce la corrispondente funzione set.

Metodologia per la realizzazione

Da classe UML C a classe Java C.
• La classe Java C è public e si trova in un file dal nome C.java.
• C è derivata da Object (no extends).

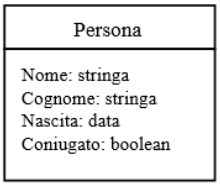
I campi dati della classe Java C corrispondono agli attributi della classe UML C. Le regole principali sono le seguenti:
• I campi dati di C sono tutti private o protected, per incrementare l'information hiding.
• Tali campi possono essere dichiarati final, se non vengono più cambiati dopo la creazione dell'oggetto (secondo la tabella delle proprietà immutabili prodotta nella fase di progetto).

Nota: dichiarando final un campo dati si impone che esso non possa essere modificato dopo l'inizializzazione. Ma se il campo dati contiene è un riferimento ad un oggetto nulla impedisce di modificare l'oggetto stesso. Quindi l'efficacia di final è limitata.

I campi funzione della classe C sono tutti public.
Costruttori: devono inizializzare tutti i campi dati, esplicitamente o implicitamente.
Nel primo caso, le informazioni per l'inizializzazione vengono tipicamente acquisite tramite gli argomenti.
Funzioni get: in generale, vanno previste per tutti i campi dati.
Funzioni set: vanno previste solo per quei campi dati che possono mutare (tipicamente, non dichiarati final).
equals(): tipicamente, non è necessario fare overriding della funzione equals() ereditata dalla classe Object.
Infatti due entità sono uguali solo se in realtà sono la stessa entità e quindi il comportamento di default della funzione equals() è corretto.
clone(): in molti casi, è ragionevole decidere di non mettere a disposizione la possibilità di copiare un oggetto, e non rendere disponibile la funzione clone() (non facendo overriding della funzione protected ereditata da Object).
Questa scelta deve essere fatta solo nel caso in cui si vuole che i moduli clienti utilizzino ogni oggetto della classe singolarmente e direttamente.
toString(): si può prevedere di farne overriding, per avere una rappresentazione testuale dell'oggetto.

Singola classe UML con soli attributi: esempio

Risultato fase di analisi:



Risultato fase di progetto:

Tipo UML	Rappresentazione in Java
stringa	String
data	int,int,int
booleano	boolean

Classe UML	Proprietà immutabile
Persona	*nome*
	cognome
	nascita

Classe UML	Proprietà	
	nota alla nascita	non nota alla nascita

Realizzazione in Java

```
// File SoloAttributi/Persona.java
public class Persona {
    private final String nome, cognome;
    private final int giorno_nascita, mese_nascita, anno_nascita;
    private boolean coniugato;
    public Persona(String n, String c, int g, int m, int a)  {
        nome = n;
        cognome = c;
        giorno_nascita = g;
        mese_nascita = m;
        anno_nascita = a;
     }
     public String getNome() { return nome; }
     public String getCognome() { return cognome; }
     public int getGiornoNascita() { return giorno_nascita; }
     public int getMeseNascita() { return mese_nascita; }
     public int getAnnoNascita() { return anno_nascita; }
     public void setConiugato(boolean c) { coniugato = c; }
     public boolean getConiugato() { return coniugato; }
     }
     public String toString() {
          return nome + ' ' + cognome + ", " + giorno_nascita + "/" + mese_nascita + "/" + anno_nascita + ", " +
(coniugato?"coniugato":"celibe");
     }
}
```

Il problema dei valori non ammessi

Ricordiamo che, in alcuni casi, il tipo base Java usato per rappresentare il tipo di un attributo ha dei valori non ammessi per quest'ultimo.
Ad esempio, nella classe UML Persona potrebbe essere presente un attributo età, con valori interi ammessi compresi fra 0 e 120.
In tali casi la fase di progetto ha stabilito se dobbiamo utilizzare nella realizzazione un approccio di verifica lato client o lato server.
Successivamente, per pure esigenze di compattezza del codice mostrato, adotteremo l'approccio lato client.

```
// File SoloAttributi/VerificaLatoServer/Persona.java
public class Persona {
    private int eta;
    public Persona(int e) throws EccezionePrecondizioni {
        if (e < 0 || e > 120) // CONTROLLO PRECONDIZIONI
           throw new EccezionePrecondizioni("L'eta' deve essere compresa fra 0 e 120");
        eta = e;
    }
    public int getEta() { return eta; }
    public void setEta(int e) throws EccezionePrecondizioni {
        if (e  <  0  || e >  120)    // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni(); eta = e;
    }
    public String  toString()  {
        return " (" + eta + " anni)";
    }
}
```

Esempio di cliente

InizioSpecificaOperazioni : Analisi Statisitica
QuantiConiugati (i: Insieme(Persona)): intero
pre: nessuna
post: result è il numero di coniugati nell'insieme di persone i
FineSpecifica

Realizzazione del cliente

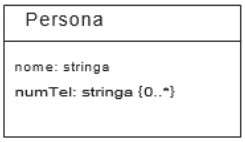
```
// File SoloAttributi/AnalisiStatistica.java
import java.util.*;
public final class AnalisiStatistica {
    public static int quantiConiugati(Set<Persona> i) {
        int quanti = 0;
        Iterator<Persona> it = i.iterator();
        while(it.hasNext()) {
            Persona elem = it.next();
            if (elem.getConiugato())
                quanti++;
        }
        return quanti;
    }
    private AnalisiStatistica() {}
}
```

Molteplicità di attributi

Quando la classe UML C ha attributi UML con una loro molteplicità (ad es., numTel: stringa {0..∗}), possiamo usare per la loro rappresentazione una classe contenitore apposita, come HashSet<String>.
In particolare, va previsto un campo dato di tale classe, che va inizializzato con new() dal costruttore della classe Java C.
Per la gestione di questo campo vanno previste opportune funzioni public:
• per la scrittura del campo sono necessarie due funzioni, rispettivamente per l'inserimento di elementi nell'insieme e per la loro cancellazione;
• per la lettura del campo è necessaria una funzione **get**.

Molteplicità di attributi: esempio

Realizziamo la classe Persona in maniera che ogni persona possa avere un numero qualsiasi di numeri di telefono.



```
// File MolteplicitaAttributi/Persona.java
import java.util.*;

public class Persona {
    private final String nome;
    private HashSet<String> numTel;
    public Persona(String n) {
        numTel = new HashSet<String>();
        nome = n;
    }
    public String getNome() { return nome; }
    public void aggiungiNumTel(String n) {
        if (n != null) numTel.add(n);
    }
    public void eliminaNumTel(String n) { numTel.remove(n); }
    public Set<String> getNumTel() {
        return (HashSet<String>)numTel.clone();
    }
    public String  toString()  {
        return nome + ' ' + numTel;
    }
}
```

Classe Java Persona: considerazioni

• La classe ha un campo dato di tipo HashSet.
• Il costruttore della classe Persona crea un oggetto di tale classe, usandone il costruttore. Di fatto, viene creato un insieme vuoto di riferimenti di tipo String.
• Ci sono varie funzioni che permettono di gestire l'insieme:
– aggiungiNumTel(String): permette di inserire un nuovo numero telefonico;
il fatto che non vengano creati duplicati nella struttura di dati è garantito dal funzionamento della funzione Set.add(), che verifica tramite la funzione equals() (in questo caso di String) l'eventuale presenza dell'oggetto di cui si richiede l'inserimento;
– eliminaNumTel(Stringa): permette di eliminare un numero telefonico;
– getNumTel(): permette di ottenere tutti i numeri telefonici di una persona.
• Si noti che la funzione getNumTel() restituisce un Set<String>. L'uso dell'interfaccia Set invece di una classe concreta che la realizza (come HashSet) permette ai clienti della classe di astrarre della specifica struttura dati utilizzata per realizzare le funzionalità previste da Set, aumentando così l'information hiding.
• Si noti che la funzione getNumTel() restituisce una copia dell'insieme dei numeri di telefono (ovvero della struttura di dati), in quanto abbiamo scelto che l'attributo numTel venga gestito solamente dalla classe Persona.
• Se così non fosse, daremmo al cliente della classe Persona la possibilità di modificare l'insieme che rappresenta l'attributo numTel a suo piacimento, distruggendo la modularizzazione.
• Queste considerazioni valgono ogni volta che restituiamo un valore di un tipo UML realizzato mediante una classe Java i cui oggetti sono mutabili.

Cliente della classe Java Persona

Per comprendere meglio questo aspetto, consideriamo un cliente della classe Persona specificato come segue.

InizioSpecificaOperazioni : **Gestione Rubrica**
TuttiNumTel (p1 : Persona, p2 : Persona): Insieme(stringa)
pre: nessuna
post: result è l'insieme unione dei numeri di telefono di p1 e p2
FineSpecifica

Per l'operazione TuttiNumTel(p1,p2) adottiamo il seguente algoritmo:

```
Insieme(stringa) result = p1.numTel;
per ogni elemento el di p2.numTel
    aggiungi el a result
return result
```

```
// File MolteplicitaAttributi/GestioneRubrica.java
import java.util.*;
public final class GestioneRubrica{
    public static Set<String> tuttiNumTel (Persona p1, Persona p2) {
        Set<String> result = p1.getNumTel();
        Iterator<String> it = p2.getNumTel().iterator();
        while(it.hasNext())
            result.add(it.next());
        return result;
     }
     private GestioneRubrica() { };
}
```

Questa funzione farebbe side-effect indesiderato su p1 se getNumTel() non restituisse una copia dell'insieme dei numeri di telefono.
Notiamo che la funzione cliente tuttiNumTel() si basa sull'assunzione che la funzione getNumTel() restituisca una copia della struttura di dati che rappresenta i numeri di telefono.
Se così non fosse (cioè se la funzione tuttiNumTel() non lavorasse su una copia, ma sull'originale) verrebbe completamente distrutta la struttura di dati, mediante le ripetute operazioni di inserimento.
L'errore di progettazione che consiste nel permettere al cliente di distruggere le strutture di dati private di un oggetto si chiama **interferenza.**

Esercizio 1: altro cliente della classe

Realizzare in Java le seguenti operazioni Analisi Recapiti:

InizioSpecificaOperazioni Analisi Recapiti

Convivono (p1 : Persona, p2 : Persona): booleano
pre: nessuna
post: result vale true se p1 e p2 hanno almeno un numero telefonico in comune, vale false, altrimenti

FineSpecifica

Altra realizzazione della classe Java Persona

• La funzione getNumTel(), che permette di ottenere tutti i numeri telefonici di una persona, potrebbe essere realizzata restituendo un iteratore dell'insieme dei numeri di telefono.
• Il vantaggio di questa scelta consiste in un minore utilizzo di memoria.
• Lo svantaggio risiede nel fatto che tipicamente i clienti devono realizzare funzioni più complesse.
• Per eliminare la possibilità che i clienti facciano interferenza, restituiamo un iteratore realizzato tramite la classe IteratoreSolaLettura<T>, che elimina remove() da Iterator.
Riassumendo, possiamo scegliere di realizzare la classe Persona attraverso due schemi realizzativi differenti.

	getNumTel()	Vantaggi	Svantaggi
Senza condivisione memoria	restituisce copia profonda (clone())	cliente più semplice	potenziale spreco memoria
Con condivisione memoria	restituisce IteratoreSolaLettura	cliente più complicato	risparmio memoria

Schema realizzativo con condivisione

```
// File MolteplicitaAttributiCond/Persona.java
import java.util.*;
import IteratoreSolaLettura.*;
public class Persona {
    private final String nome;
    private HashSet<String> numTel;
    public Persona(String n) {
        numTel = new HashSet<String>();
        nome = n;
    }
    public String getNome() { return nome; }
    public void aggiungiNumTel(String n) {
        if (n != null) numTel.add(n);
    }
    public void eliminaNumTel(String n) { numTel.remove(n);}
    public Iterator<String> getNumTel() {
        return new IteratoreSolaLettura<String>(numTel.iterator());
    }
    public String  toString()  { return nome + ' ' + numTel; }
}
```

La classe Java IteratoreSolaLettura

```
// File IteratoreSolaLettura/IteratoreSolaLettura.java
package IteratoreSolaLettura;
import java.util.*;
public class IteratoreSolaLettura<T> implements Iterator<T> {
    // elimina remove() da Iterator
    private Iterator<T> i;
    public IteratoreSolaLettura(Iterator<T> it) { i = it; }
    public T next() { return i.next(); }
    public boolean hasNext() { return i.hasNext(); }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Esercizio 2: clienti per la nuova versione della classe Persona

Facendo riferimento all'ultima realizzazione della classe Persona (quella con lo schema realizzativo con condivisione di memoria), realizzare le operazioni tuttiNumTel() e Convivono() come opportune funzioni cliente.

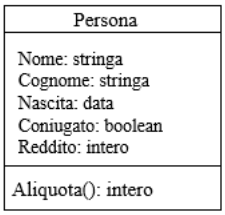
Realizzazione di classe con attributi e operazioni

- Si procede come prima per quanto riguarda gli attributi.
- Si analizza la specifica della classe UML C e gli algoritmi associati alle operazioni di tale classe, che forniscono le informazioni sul significato di ogni operazione.
- Ogni operazione viene realizzata da una funzione public della classe Java.

Sono possibili eventuali funzioni private o protected che dovessero servire per la realizzazione dei metodi della classe C, ma che non vogliamo rendere disponibili ai clienti.

Singola classe con attributi e operazioni: esempio

Consideriamo un raffinamento della classe UML Persona vista in uno degli esempi precedenti. Si noti che ora una persona ha anche un reddito.



Specifica della classe UML

InizioSpecificaOperazioniClasse Persona

Aliquota (): intero

pre: nessuna

post: result vale 0 se this.Reddito è inferiore a 5001, vale 20 se this.Reddito è compreso fra 5001 e 10000, vale 30 se this.Reddito è compreso fra 10001 e 30000, vale 40 se this.Reddito è superiore a 30000.

FineSpecifica

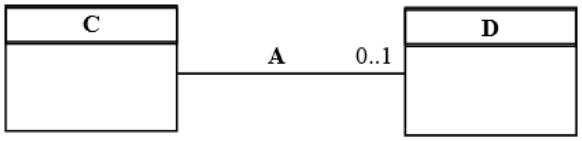
Realizzazione in Java

```
// File AttributiEOperazioni/Persona.java
public class Persona {
    private final String nome, cognome;
    private final int giorno_nascita, mese_nascita, anno_nascita;
    private boolean coniugato;
    private int reddito;
    public Persona(String n, String c, int g, int m, int a)  {
        nome = n;
        cognome = c;
        giorno_nascita = g;
        mese_nascita = m;
        anno_nascita = a;
     }
     public String getNome() { return nome; }
     public String getCognome() { return cognome; }
     public int getGiornoNascita() { return giorno_nascita; }
     public int getMeseNascita() { return mese_nascita; }
```

```
    public int getAnnoNascita() { return anno_nascita;  }
    public void setConiugato(boolean c) { coniugato = c; }
    public boolean getConiugato() { return coniugato; }
    public void setReddito(int r) { reddito = r; }
    public int getReddito() { return reddito; }
    public int aliquota() {
       if (reddito < 5001) return 0;
       else if (reddito < 10001) return 20;
       else if (reddito < 30001) return 30;
       else return 40;
     }
     public String toString() {
           return nome + ' ' + cognome + ", " + giorno_nascita + "/" + mese_nascita + "/" + anno_nascita + ", " +
(coniugato?"coniugato":"celibe") + ", aliquota fiscale: " + aliquota();
     }
}
```

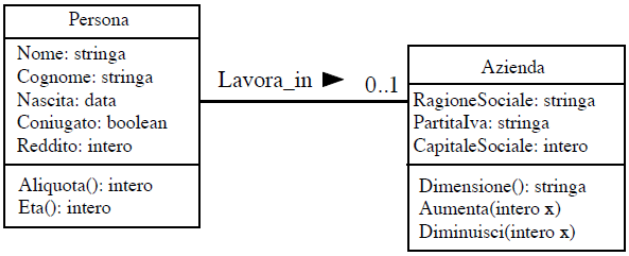
Realizzazione di associazioni

Associazione con molteplicità 0..1 a responsabilità singola, senza attributi



Consideriamo il caso in cui
• l'associazione sia binaria;
• l'associazione colleghi ogni istanza di C a zero o una istanza di D (molteplicità 0..1),
• la tabella delle responsabilità prodotta in fase di progetto ci dica che C è l'unica ad avere responsabilità sull'associazione A (cioè dobbiamo realizzare un "solo verso" dell'associazione)
• l'associazione A non abbia attributi.
In questo caso, la realizzazione è simile a quella per un attributo. Infatti, oltre a quanto stabilito per gli attributi e le operazioni, per ogni associazione A del tipo mostrato in figura, aggiungiamo alla classe Java C:
• un campo dato di tipo D nella parte private (o protected) che rappresenta, per ogni oggetto x della classe C, l'oggetto della classe D connesso ad x tramite l'associazione A,
• una funzione get che consente di calcolare, per ogni oggetto x della classe C, l'oggetto della classe D connesso a x tramite l'associazione A (la funzione restituisce null se x non partecipa ad alcuna istanza di A),
• una funzione set, che consente di stabilire che l'oggetto x della classe C è legato ad un oggetto y della classe D tramite l'associazione A (sostituendo l'eventuale legame già presente); se la tale funzione viene chiamata con null come argomento, allora la chiamata stabilisce che l'oggetto x della classe C non è più legato ad alcun oggetto della classe D tramite l'associazione A.

Due classi legate da associazione: esempio



Assumiamo di avere stabilito, nella fase di progetto, che:
• la ragione sociale e la partita Iva di un'azienda non cambiano;
• solo Persona abbia responsabilità sull'associazione (non ci interessa conoscere i dipendenti di un'azienda, ma solo in quale azienda lavora una persona).

Specifica della classe UML Azienda

InizioSpecificaOperazioniClasse Azienda
Dimensione (): stringa
    pre: nessuna
    post: result vale "Piccola" se this.CapitaleSociale è inferiore a 51, vale "Media" se this.CapitaleSociale è compreso fra 51 e 250, vale "Grande" se this.CapitaleSociale è superiore a 250
Aumenta (i: intero)
    pre: i > 0
    post: this.CapitaleSociale vale pre(this.CapitaleSociale) + i

Diminuisci (i: intero)
pre: 1 ≤ i ≤ this.CapitaleSociale
post: this.CapitaleSociale vale pre(this.CapitaleSociale) - i
FineSpecifica

Classe Java Azienda

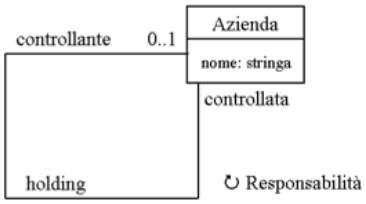
```
// File Associazioni01/Azienda.java
public class Azienda {
   private final String ragioneSociale, partitaIva;
   private int capitaleSociale;
   public Azienda(String r, String p) {
      ragioneSociale = r;
      partitaIva = p;
   }
   public String getRagioneSociale() { return ragioneSociale;  }
   public String getPartitaIva() { return partitaIva; }
   public int getCapitaleSociale() { return capitaleSociale; }
   public void aumenta(int i) { capitaleSociale += i; }
   public void diminuisci(int i) { capitaleSociale -= i; }
   public String dimensione() {
      if (capitaleSociale < 51) return "Piccola";
      else if (capitaleSociale < 251) return "Media";
      else return "Grande";
   }
   public String toString() {
    return ragioneSociale + " (P.I.: " + partitaIva +  "), capitale sociale:  "  +  getCapitaleSociale()  + ", tipo azienda: " + dimensione(); }
}
```

Classe Java Persona

```
public class Persona {
   // altri campi dati e funzione private Azienda lavoraIn;
   public Azienda getLavoraIn() { return lavoraIn; }
   public void setLavoraIn(Azienda a) { lavoraIn = a; }
   public String toString() {
          return nome + ' ' + cognome + ", " + giorno_nascita + "/" + mese_nascita + "/" + anno_nascita + ", " +
          (coniugato?"coniugato":"celibe") + ", aliquota fiscale: " + aliquota() + (lavoraIn != null?", lavora presso la ditta " + lavoraIn:
          ", disoccupato");
    }
}
```

Associazioni che insistono più volte sulla stessa classe

Quanto detto vale anche per il caso in cui l'associazione coinvolga più volte la stessa classe. In questo caso il concetto di responsabilità si attribuisce ai ruoli, piuttosto che alle classi.



Supponiamo che la classe Azienda abbia la responsabilità su holding, solo nel ruolo controllata. Questo significa che, dato un oggetto x della classe Azienda, vogliamo poter eseguire operazioni su x per conoscere l'azienda controllante, per aggiornare l'azienda controllante, ecc.

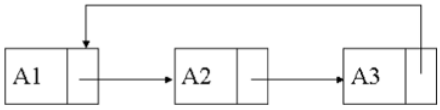
Esempio

In questo caso, il nome del campo dato che rappresenta l'associazione viene in genere scelto uguale al nome del ruolo (nell'esempio, il nome è controllante).

```
// File Ruoli/Azienda.java
public class Azienda {
    private final String nome;
    private  Azienda  controllante;  //  il nome del campo è uguale al ruolo
    public Azienda(String n) { nome = n; }
    public Azienda getControllante() { return controllante; }
    public void setControllante(Azienda a) { controllante = a; }
    public String toString() {
            return nome + ((controllante == null)?"": (" controllata da: "+controllante));
    }
}
```

Potenziale situazione anomala

L'azienda A1 ha come controllante A2, che ha come controllante A3, che ha a sua volta come controllante A1.
Diciamo che L'azienda A1 è "di fatto controllata da se stessa".



Associazioni con molteplicità 0..* a responsabilità singola, senza attributi

Ci concentriamo su associazioni binarie con molteplicità 0..*, con le seguenti assunzioni:
• non abbiano attributi di associazione;
• solo una delle due classi haresponsabilità sull'associazione (dobbiamo rappresentare un solo verso dell'associazione).
Per rappresentare l'associazione As fra le classi UML A e B con molteplicità 0..* abbiamo bisogno di una struttura di dati per rappresentare i link fra un oggetto di classe A e più oggetti di classe B.
In particolare, la classe Java A avrà:
• un campo dato di un tipo opportuno (ad esempio HashSet), per rappresentare la struttura di dati;
• dei campi funzione che permettano di gestire tale struttura di dati (funzioni get, inserisci, elimina).

Esempio

Assumiamo che la fase di progetto abbia stabilito che solo Persona ha responsabilità sull'associazione (non ci interessa conoscere i dipendenti passati di un'azienda, ma solo in quale azienda ha lavorato una persona).
Assumiamo anche che dai requisiti si evinca che è possibile eliminare un link di tipo Ha lavorato.

Classe Java Persona

```
// File Ass0STAR/Persona.java
import java.util.*;
public class Persona {
    private final String nome;
    private HashSet<Azienda> insieme_link;
    public Persona(String n) {
        nome = n;
        insieme_link = new HashSet<Azienda>();
    }
    public String getNome() { return  nome; }
    public void inserisciLinkHaLavorato(Azienda az) {
        if (az != null) insieme_link.add(az);
    }
    public void eliminaLinkHaLavorato(Azienda az) {
        if (az != null) insieme_link.remove(az);
    }
    public Set<Azienda> getLinkHaLavorato() { return (HashSet<Azienda>)insieme_link.clone();
    }
}
```

Classe Java Persona: considerazioni

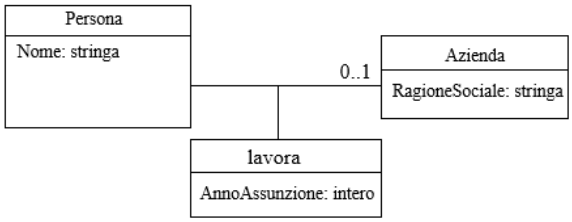
• La classe ha un campo dato di tipo HashSet<Azienda>.
• Il costruttore della classe Persona crea un oggetto di tale classe, usandone il costruttore. Di fatto, viene creato un insieme vuoto di riferimenti di tipo Azienda.
• Ci sono varie funzioni che permettono di gestire l'insieme:
– inserisciLinkHaLavorato(Azienda): permette di inserire un nuovo link;
– eliminaLinkHaLavorato(Azienda): permette di eliminare un link esistente;
– getLinkHaLavorato(): permette di ottenere tutti i link di una persona.
• Si noti che la funzione getLinkHaLavorato() restituisce un Set<Azienda> e non un HashSet<Azienda>. Come detto precedentemente nel caso di attributi con molteplicità 0..*, l'uso dell'interfaccia Set invece di una classe concreta che la realizza (come HashSet) permette ai clienti della classe di astrarre della specifica struttura dati utilizzata per realizzare le funzionalità previste da Set, aumentando così l'information hiding.
• Seguendo lo schema realizzativo senza condivisione di memoria, la funzione getLinkHaLavorato() restituisce una copia dell'insieme dei link (ovvero della struttura di dati).
• Questa situazione è infatti analoga a quella degli attributi di classe con molteplicità 0..* visti in precedenza, e scegliamo che i link dell'associazione HaLavorato vengano gestiti solamente dalla classe Persona, che ha responsabilità sull'associazione.
• Per semplicità, nel seguito utilizzeremo sempre lo schema realizzativo senza condivisione di memoria.

Attributi di associazione

Consideriamo il caso in cui la classe C sia l'unica ad avere la responsabilità sull'associazione A, e l'associazione A abbia uno o più attributi di molteplicità 1..1.
Considereremo
• inizialmente che A abbia, rispetto a C, molteplicità 0..1;
• dopo anche molteplicità 0..*.
Gli altri casi, come altre molteplicità per attributi (immediato), o responsabilità sull'associazione di entrambe le classi (difficile), verranno considerati in seguito.

Rappresentazione di attributi di associazione realizzazione naive

Esempio (solo Persona ha responsabilità sull'associazione):



Realizzazione naive:
1. si aggiunge alla classe C un campo per ogni attributo dell'associazione A, che viene trattato in modo simile ad un attributo della classe C.
2. si fa uso di una struttura di dati ad hoc per rappresentare istanze dell'associazione (link).

Consideriamo l'esempio, scegliendo la prima strategia.

```
// File Ass01Attr-NoLink/Persona.java
public class Persona {
    private final String nome;
    private Azienda lavora;
    private int annoAssunzione;
    public Persona(String n) { nome = n; }
    public String getNome() { return nome; }
    public Azienda getLavora() { return lavora; }
    public int getAnnoAssunzione() throws EccezionePrecondizioni {
        if (lavora==null) throw new EccezionePrecondizioni(this + " Non partecipa alla associazione Lavora");
            return annoAssunzione;
    }
    public void setLavora(Azienda a, int x) {
        if (a != null) {
            lavora = a;
            annoAssunzione = x; }
    }
    public void eliminaLavora() { lavora = null; }
}
```

Classe Java EccezionePrecondizioni

```
// File Ass01Attr/EccezionePrecondizioni.java
public class EccezionePrecondizioni extends Exception {
    private String messaggio;
    public EccezionePrecondizioni(String m) { messaggio = m; }
    public EccezionePrecondizioni() { messaggio = "Si e' verificata una violazione delle precondizioni";  }
    public String toString() { return messaggio; }
}
```

Rappresentazione di attributi di associazione realizzazione naive: osservazioni

La funzione setLavora() ha ora due parametri, perché nel momento in cui si lega un oggetto della classe C ad un oggetto della classe D tramite A, occorre specificare anche il valore dell'attributo dell'associazione (essendo tale attributo di molteplicità 1..1).
Il cliente della classe ha la responsabilità di chiamare la funzione getAnnoAssunzione() correttamente, cioè quando l'oggetto di invocazione x effettivamente partecipa ad una istanza della associazione lavora (x.getLavora() != null). Altrimenti viene generata una opportuna istanza di EccezionePrecondizioni.
Il fatto che l'attributo dell'associazione venga realizzato attraverso un campo dato della classe C non deve trarre in inganno: concettualmente l'attributo appartiene all'associazione, ma è evidente che, essendo l'associazione 0..1 da C a D, ed essendo l'attributo di tipo 1..1, dato un oggetto x di C che partecipa all'associazione A, associato ad x c'è uno ed un solo valore per l'attributo. Quindi è corretto, in fase di implementazione, attribuire alla classe C il campo dato che rappresenta l'attributo dell'associazione.
Importante: questa strategia realizzativa naive non può essere estesa ad associazioni con molteplicità 0..*!

Attributi di associazione: realizzazione

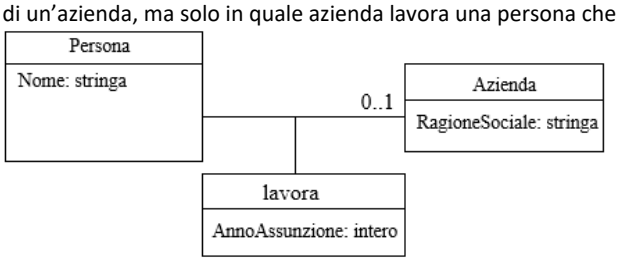
Consideriamo adesso una strategia di realizzazione più ragionata, che è quella da preferirsi.
La presenza degli attributi sull'associazione impedisce di usare i meccanismi base di Java (cioè i riferimenti) per rappresentare i link UML.
Dobbiamo quindi rappresentare la nozione di link in modo esplicito attraverso una classe.
Per rappresentare l'associazione A fra le classi UML C e D introduciamo una ulteriore classe Java TipoLinkA, che ha lo scopo di rappresentare i link (tuple -in questo caso coppie) fra gli oggetti delle classi C e D.
Si noti che questi link (tuple) sono valori, non oggetti. Quindi la classe TipoLinkA rappresenta un tipo, non una classe UML.
In particolare, ci sarà un oggetto di classe TipoLinkA per ogni link (presente al livello estensionale) fra un oggetto di classe C ed uno di classe D.
La classe Java TipoLinkA avrà campi dati per rappresentare:
• gli attributi dell'associazione;
• i riferimenti agli oggetti delle classi C e D che costituiscono le componenti della tupla che il link rappresenta (Nota per essere precisi tali riferimenti sono variabili che contengano gli identificatori degli oggetti coinvolti).
Supponendo che solo la classe UML C abbia responsabilità sull'associazione A, la classe Java C che la realizza dovrà tenere conto della presenza dei link.
In particolare, la classe Java C avrà:
• un campo dato di tipo TipoLinkA, per rappresentare l'eventuale link; in particolare, se tale campo vale null, allora significa che l'oggetto di classe C non è associato ad un oggetto di classe D;
• dei campi funzione che permettano di gestire il link (funzioni get, inserisci, elimina).

Funzioni della classe Java TipoLinkA

La classe Java TipoLinkA avrà inoltre le seguenti funzioni:
• funzioni per la gestione dei suoi campi dati:
– costruttore (lancia un'eccezione di tipo EccezionePrecondizioni se i riferimenti di tipo C e D passati come argomenti sono null),
– funzioni get;
• funzione equals() ridefinita in maniera tale da verificare l'uguaglianza solo sugli oggetti collegati dal link, ignorando gli attributi.
• funzione hashCode() ridefinita in maniera tale da verificare il principio secondo il quale se due oggetti sono uguali secondo equals() allora questi devono avere lo stesso codice di hash secondo hashCode().
Non avrà invece funzioni set: i suoi oggetti sono immutabili, ovvero una volta creati non possono più essere cambiati.

Realizzazione

Ricordiamo che stiamo assumendo che solo Persona abbia responsabilità sull'associazione (non ci interessa conoscere i dipendenti di un'azienda, ma solo in quale azienda lavora una persona che lavora).



Classe Java TipoLinkLavora

```
// File Ass01Attr/TipoLinkLavora

public class TipoLinkLavora {
    private final Persona laPersona;
    private final Azienda laAzienda;
    private final int annoAssunzione;
    public TipoLinkLavora(Azienda x, Persona y, int a) throws EccezionePrecondizioni {
        if (x == null || y == null)  // CONTROLLO PRECONDIZIONI
           throw new EccezionePrecondizioni ("Gli oggetti devono essere inizializzati");
        laAzienda = x;
        laPersona = y;
        annoAssunzione = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkLavora b = (TipoLinkLavora)o;
            return b.laPersona == laPersona && b.laAzienda == laAzienda;
        }
        else return false;
        }
    public int hashCode() { return laPersona.hashCode() + laAzienda.hashCode();  }
    public Azienda getAzienda() { return laAzienda; }
    public Persona getPersona() { return laPersona; }
    public int getAnnoAssunzione() { return annoAssunzione; }
}
```

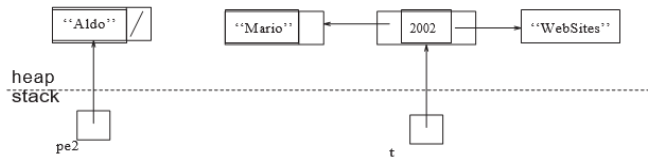
Classe Java Persona

```
// File Ass01Attr/Persona.java

public class Persona {
    private final String nome;
    private TipoLinkLavora link;
    public Persona(String n) { nome = n; }
    public String getNome() { return nome; }
    public void inserisciLinkLavora(TipoLinkLavora t) {
        if (link == null && t != null && t.getPersona() == this) link = t;
   }
   public void eliminaLinkLavora() { link = null; }
   public  TipoLinkLavora  getLinkLavora()  { return link; }
}
```

Considerazioni sulle classi Java

• Si noti che i campi dati nella classe TipoLinkLavora sono tutti final.
Di fatto un oggetto di tale classe è immutabile, ovvero una volta creato non può più essere cambiato.
• La funzione inserisciLinkLavora() della classe Persona deve assicurarsi che:
– la persona oggetto di invocazione non sia già associata ad un link;
– l'oggetto che rappresenta il link esista;
– la persona a cui si riferisce il link sia l'oggetto di invocazione.
• Per cambiare l'oggetto della classe Azienda a cui una persona è legata tramite l'associazione lavora è necessario invocare prima eliminaLinkLavora() e poi inserisciLinkLavora().

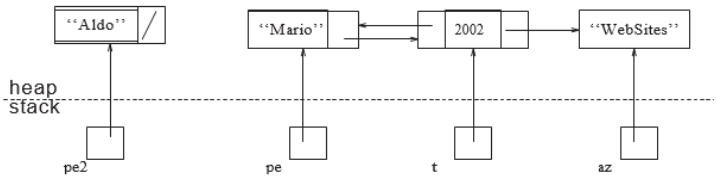
Controllo coerenza riferimenti



Il link t non si riferisce all'oggetto "Aldo".
Quindi, se chiediamo all'oggetto "Aldo" di inserire tale link, non deve essere modificato nulla.
Infatti la funzione inserisciLinkLavora() della classe Persona si assicura che la persona a cui si riferisce il link sia l'oggetto di invocazione.

Possibile stato della memoria

Due oggetti di classe Persona, di cui uno che lavora ed uno no.



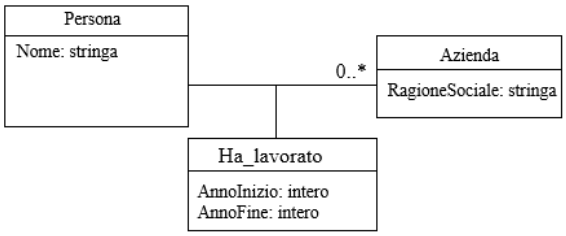
Realizzazione della situazione di esempio

```
Azienda az = new Azienda("WebSites");
Persona pe = new Persona("Mario"),
    pe2 = new Persona("Aldo");
TipoLinkLavora t = null;
try {
    t = new TipoLinkLavora(az,pe,2002);
} catch (EccezionePrecondizioni e) { System.out.println(e); }
pe.inserisciLinkLavora(t);
```

Associazioni 0..* con attributi

Ci concentriamo ora su associazioni binarie con molteplicità 0..*,e con attributi. Ci riferiremo al seguente esempio (si noti che non è possibile rappresentare che una persona ha lavorato due o più volte per la stessa azienda). Assumiamo per semplicità che si lavori sempre per anni interi.

Schema concettuale da realizzare in Java (solo la classe Persona ha responsabilità sull'associazione):



Dobbiamo combinare le scelte fatte in precedenza:

1. come per tutte le associazioni con attributi, dobbiamo definire una apposita classe Java per la rappresentazione del link (TipoLinkHaLavorato); inoltre, dobbiamo prevedere la possibilità che il costruttore di questa classe lanci un'eccezione nel caso in cui i riferimenti passatigli come argomento siano pari a null;
2. come per tutte le associazioni con vincolo di molteplicità 0..*, dobbiamo utilizzare una struttura di dati per la rappresentazione dei link.

Rappresentazione dei link

La classe Java TipoLinkHaLavorato per la rappresentazione dei link deve gestire:

- gli attributi dell'associazione (AnnoInizio, AnnoFine);
- i riferimenti agli oggetti relativi al link (di classe Persona e Azienda).

Pertanto, avrà gli opportuni campi dati e funzioni (costruttori e get).

Inoltre, avrà la funzione equals per verificare l'uguaglianza solo sugli oggetti collegati dal link, ignorando gli attributi e la funzione hashCode ridefinita di conseguenza.

```
// File Ass0STARAttr/TipoLinkHaLavorato
public class TipoLinkHaLavorato {
   private final Persona laPersona;
   private final Azienda laAzienda;
   private final int annoInizio, annoFine;
   public TipoLinkHaLavorato(Azienda x, Persona y, int ai, int af) throws EccezionePrecondizioni {
      if (x == null || y == null) // CONTROLLO PRECONDIZIONI
         throw new EccezionePrecondizioni ("Gli oggetti devono essere inizializzati");
      laAzienda = x;
      laPersona = y;
      annoInizio = ai;
      annoFine = af;
      }
   public boolean equals(Object o) {
      if (o != null && getClass().equals(o.getClass())) {
         TipoLinkHaLavorato b = (TipoLinkHaLavorato)o;
```

```
            return b.laPersona == laPersona && b.laAzienda == laAzienda;
        }
        else return false;
        }
    public int hashCode() { return laPersona.hashCode() + laAzienda.hashCode(); }
    public Azienda getAzienda() { return laAzienda; }
    public Persona getPersona() { return laPersona; }
    public int getAnnoInizio() { return annoInizio; }
    public int getAnnoFine() { return annoFine; }
}
```

Classe Java Persona

```
// File Ass0STARAttr/Persona.java
 import java.util.*;
public class Persona {
    private final String  nome;
    private HashSet<TipoLinkHaLavorato> insieme_link;
    public Persona(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkHaLavorato>();
    }
    public String getNome() { return nome; }
    public void inserisciLinkHaLavorato(TipoLinkHaLavorato t) {
        if (t != null && t.getPersona() == this) insieme_link.add(t);
    }
    public void eliminaLinkHaLavorato(TipoLinkHaLavorato t) {
        if (t != null && t.getPersona() == this) insieme_link.remove(t);
   }
    public Set<TipoLinkHaLavorato> getLinkHaLavorato() { return (HashSet<TipoLinkHaLavorato>)insieme_link.clone(); }
}
```

Riassunto metodi per responsabilità singola

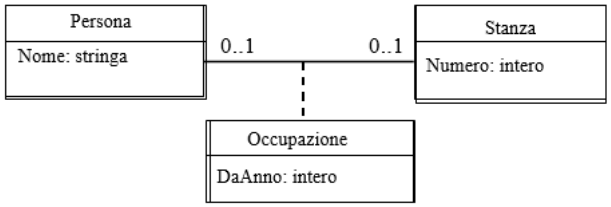
La seguente tabella riassume gli argomenti ed i controlli necessari per le funzioni di inserimento e cancellazione nei casi di associazione (a responsabilità singola) finora esaminati.

		0..1		0..*	
		no attr.	attributo	no attr.	attributo
inser.	arg.	rif. a oggetto	rif. a link	rif. a oggetto	rif. a link
	controllo	–	arg != null link si rif. a this link == null	arg != null	arg != null link si rif. a this
canc.	arg.	null	nessuno	rif. a oggetto	rif. a link
	controllo	–	–	arg != null	arg != null link si rif. a this

Responsabilità di entrambe le classi UML

Affrontiamo il caso di associazione binaria in cui entrambe le classi abbiano la responsabilità sull'associazione. Per il momento, assumiamo che la molteplicità sia 0..1 per entrambe le classi.

Supponiamo che sia Persona sia Stanza abbiano responsabilità sull'associazione.



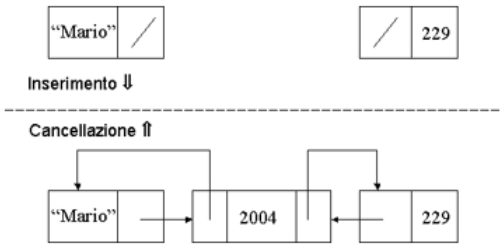
Problema di fondo:

quando creiamo un link fra un oggetto Java pe di classe Persona un oggetto Java st di classe Stanza, dobbiamo cambiare lo stato sia di pe sia di st. In particolare:

- l'oggetto pe si deve riferire all'oggetto st;
- l'oggetto st si deve riferire all'oggetto pe.

Discorso analogo vale quando eliminiamo un link fra due oggetti.

Mantenimento coerenza



Chiaramente, non possiamo dare al cliente delle classi Persona e Stanza questo onere, che deve essere gestito invece da queste ultime.
Per motivi che saranno chiariti in seguito, è preferibile centralizzare la responsabilità di assegnare i riferimenti in maniera corretta. In particolare, realizziamo una ulteriore classe Java (chiamata ManagerOccupazione) che gestisce la corretta creazione della rete dei riferimenti. Questa classe è di fatto un modulo per l'inserimento e la cancellazione di link di tipo Occupazione. Ogni suo oggetto ha un riferimento ad un oggetto Java che rappresenta un link di tipo Occupazione.
Continuiamo ad utilizzare (come in tutti i casi in cui c'è necessità di rappresentare attributi di associazione) una classe Java per i link, in questo caso TipoLinkOccupazione, che modella tuple del prodotto cartesiano tra Stanza e Persona con attributo DaAnno.

Caratteristiche delle classi Java

Persona: oltre ai campi dati e funzione per la gestione dei suoi attributi, avrà:
• un campo di tipo TipoLinkOccupazione, che viene inizializzato a null dal costruttore;
• funzioni per la gestione di questo campo, in particolare:
– void inserisciLinkOccupazione(TipoLinkOccupazione), per associare un link all'oggetto, ma che delega l'operazione effettiva a ManagerOccupazione;
– void eliminaLinkOccupazione(TipoLinkeOccupazione), per rimuovere l'associazione di un link all'oggetto, ma che anche esso delega l'operazione effettiva ManagerOccupazione;
– TipoLinkOccupazione getLinkOccupazione(), per interrogare l'oggetto; Funzioni speciali utilizzabili solo da ManagerOccupazione:
– void inserisciPerManagerOccupazione(ManagerOccupazione), funzione speciale per gestire l'inserimento di link TipoLinkeOccupazione nelle proprie strutture dati (molto semplici in questo caso);
– void eliminaPerManagerOccupazione(ManagerOccupazione), funzione speciale per gestire eliminazione di link TipoLinkeOccupazione nelle proprie strutture dati.

Stanza: del tutto simile a Persona.

Classe Java Persona

```
// File RespEntrambi01/Persona.java
public class Persona {
    private final String nome;
    private TipoLinkOccupazione link;
    public Persona(String n) { nome = n; }
    public String getNome() { return nome; }
    public void inserisciLinkOccupazione(TipoLinkOccupazione t) {
        if (t != null && t.getPersona()==this)
            ManagerOccupazione.inserisci(t);
    }
    public void eliminaLinkOccupazione(TipoLinkOccupazione t) {
        if (t != null && t.getPersona()==this)
            ManagerOccupazione.elimina(t);
    }
    public TipoLinkOccupazione getLinkOccupazione() { return link; }
    public void inserisciPerManagerOccupazione(ManagerOccupazione a) {
        if (a != null) link = a.getLink();
    }
    public void eliminaPerManagerOccupazione(ManagerOccupazione a) {
        if (a != null) link = null;
    }
}
```

Classe Java Stanza

```
// File RespEntrambi01/Stanza.java
public class Stanza {
    private final int numero;
    private TipoLinkOccupazione link;
```

```
    public Stanza(int n) { numero = n; }
    public int getNumero() { return numero; }
    public void inserisciLinkOccupazione(TipoLinkOccupazione t) {
        if (t != null && t.getStanza()==this)
            ManagerOccupazione.inserisci(t);
    }
    public void eliminaLinkOccupazione(TipoLinkOccupazione t) {
       if (t != null && t.getStanza()==this)
           ManagerOccupazione.elimina(t);
    }
    public TipoLinkOccupazione getLinkOccupazione() { return link; }
    public void inserisciPerManagerOccupazione(ManagerOccupazione a) {
       if (a != null)
          link = a.getLink();
    }
    public void eliminaPerManagerOccupazione(ManagerOccupazione a) {
      if (a != null)
          link = null;
    }
}
```

Caratteristiche delle classi Java

TipoLinkOccupazione: sarà del tutto simile al caso in cui la responsabilità sull'associazione è singola. Avrà:
• tre campi dati (per la stanza, per la persona e per l'attributo dell'associazione);
• un costruttore, che inizializza questi campi utilizzando i suoi argomenti; lancia un'eccezione se i riferimenti alla stanza o alla persona sono null;
• tre funzioni get, per interrogare l'oggetto;
• la funzione equals per verificare l'uguaglianza solo sugli oggetti collegati dal link, ignorando gli attributi;
• la funzione hashCode, ridefinita di conseguenza.

Classe Java TipoLinkOccupazione

```
// File RespEntrambi01/TipoLinkOccupazione.java
public class TipoLinkOccupazione {
    private final Stanza laStanza;
    private final Persona laPersona;
    private final int daAnno;
    public TipoLinkOccupazione(Stanza x, Persona y, int a) throws EccezionePrecondizioni {
        if (x == null || y == null)  // CONTROLLO PRECONDIZIONI
          throw new EccezionePrecondizioni ("Gli oggetti devono essere inizializzati");
        laStanza = x;
        laPersona = y;
        daAnno = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkOccupazione b = (TipoLinkOccupazione)o;
            return b.laPersona == laPersona && b.laStanza == laStanza;
        }
    else return false;
    }
    public int hashCode() { return laPersona.hashCode() + laStanza.hashCode(); }
    public Stanza getStanza() { return laStanza; }
    public Persona getPersona() { return laPersona; }
    public int getDaAnno() { return daAnno; }
}
```

Caratteristiche delle classi Java

ManagerOccupazione: avrà:
• un campo dato, di tipo TipoLinkOccupazione per la rappresentazione del link;
• funzioni per la gestione di questo campo, in particolare:
– static void inserisci(TipoLinkOccupazione), per associare un link fra una persona ed una stanza;
– static void elimina(TipoLinkOccupazione), per rimuovere un link fra una persona ed una stanza;
– TipoLinkOccupazione getLink(), per ottenere il link;
• il costruttore sarà privato;
• la classe sarà final, per evitare che si possa definire una sottoclasse in cui il costruttore è pubblico.

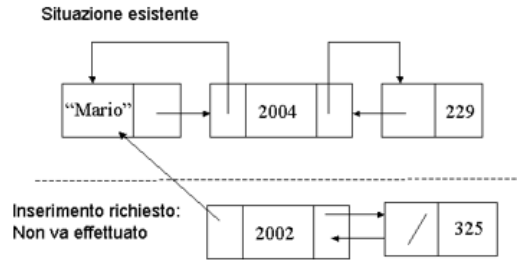
Classe Java ManagerOccupazione

```
// File RespEntrambi01/ManagerOccupazione.java
public final class ManagerOccupazione {
    private ManagerOccupazione(TipoLinkOccupazione x) { link = x; }
    private TipoLinkOccupazione link;
    public TipoLinkOccupazione getLink() { return link; }
    public static void inserisci(TipoLinkOccupazione y) {
        if (y !=  null  && y.getPersona().getLinkOccupazione() == null && y.getStanza().getLinkOccupazione() == null) {
            ManagerOccupazione k = new ManagerOccupazione(y);
            y.getStanza().inserisciPerManagerOccupazione(k);
            y.getPersona().inserisciPerManagerOccupazione(k);
        }
    }
    public static void elimina(TipoLinkOccupazione y) {
        if (y != null && y.getPersona().getLinkOccupazione().equals(y)) {
            ManagerOccupazione k = new ManagerOccupazione(y);
            y.getStanza().eliminaPerManagerOccupazione(k);
            y.getPersona().eliminaPerManagerOccupazione(k);
        }
    }
}
```

Inserimento di link: controlli

Si noti che è necessario prevenire la possibilità di richiedere agli oggetti di tipo Stanza o Persona di inserire link quando gli oggetti sono già "occupati" .

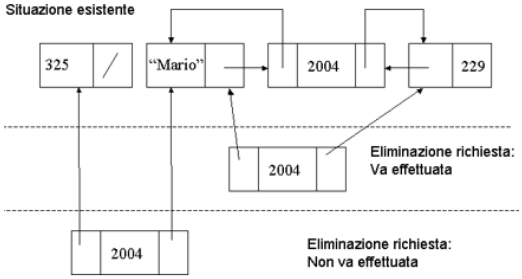
Per tale motivo la funzione inserisci() verifica (tramite getPerManagerOccupazione()) che il link y che le viene passato come argomento si riferisca ad oggetti di tipo Stanza e Persona che non sono associati ad alcun link di tipo Occupazione.



Eliminazione di link: controlli

Si noti che, al fine di prevenire la possibilità di richiedere agli oggetti di tipo Stanza o Persona di eliminare link inesistenti (creando in questa maniera situazioni inconsistenti) la funzione elimina() deve verificare (tramite equals()) che il link y che le viene passato come argomento si riferisca agli stessi oggetti di tipo Stanza e Persona del campo dato link.

Per fare ciò è sufficiente effettuare la verifica mediante il link da cui si arriva tramite la persona.



Classe Java ManagerOccupazione

Il costruttore della classe ManagerOccupazione è privato in quanto non vogliamo che i clienti siano in grado di creare oggetti di questa classe.

I clienti saranno in grado di:

• creare link, di tipo TipoLinkOccupazione, stabilendo contestualmente la stanza, la persona e l'anno;

• associare link agli oggetti di classe Stanza e Persona, mediante una chiamata alla funzione ManagerOccupazione.inserisci();

• rimuovere link, mediante una chiamata alla funzione ManagerOccupazione.elimina().

Si noti viene effettuato il controllo che gli argomenti di queste ultime due funzioni corrispondano ad oggetti (non siano null).

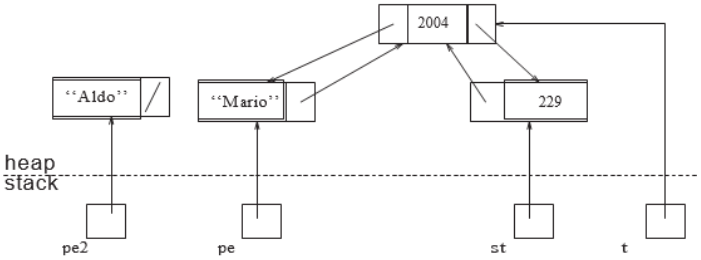
• Le funzioni inserisciPerManagerOccupazione() ed eliminaPerManagerOccupazione() della classe Persona di fatto possono essere invocate solamente dalla classe ManagerOccupazione, in quanto:

– per invocarle dobbiamo passare loro degli argomenti di tipo ManagerOccupazione, e gli oggetti della classe ManagerOccupazione non possono essere creati, se non attraverso le funzioni inserisci() ed elimina() di quest'ultima.

• Forziamo quindi i clienti che vogliono stabilire o rimuovere dei link ad usare le funzioni (statiche, in quanto svincolate da oggetti di invocazione) inserisci() ed elimina() della classe ManagerOccupazione anche quando effettuano inserimenti e cancellazioni di link mediante le classi Persona e Stanza: queste infatti delegano tali inserimenti e cancellazioni alla classe ManagerOccupazione.

Possibile stato della memoria

Due oggetti di classe Persona, di cui uno con una stanza associata ed uno no.
Si noti che l'oggetto di classe ManagerOccupazione non è direttamente accessibile dai clienti.

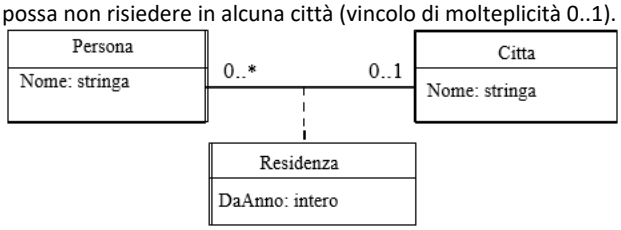


Realizzazione della situazione di esempio

```
Stanza st = new Stanza(229);
Persona pe = new Persona("Mario");
Persona pe2 = new Persona("Aldo");
TipoLinkOccupazione t = null;
try {
    t = new TipoLinkOccupazione(st,pe,2004);
} catch (EccezionePrecondizioni e) { System.out.println(e); }
ManagerOccupazione.inserisci(t);
```

Resp. di entrambe le classi UML: molteplicità 0..*

Affrontiamo il caso di associazione binaria in cui entrambe le classi abbiano la responsabilità sull'associazione, ed in cui una delle molteplicità sia 0..*. Ci riferiremo al seguente esempio.
Supponiamo che sia Persona sia Città abbiano responsabilità sull'associazione. Per semplificare, ammettiamo che una persona possa non risiedere in alcuna città (vincolo di molteplicità 0..1).



La metodologia proposta per la molteplicità 0..1 può essere usata anche per la molteplicità 0..* (per il momento, una delle due molteplicità è ancora 0..1). Le differenze principali sono le seguenti:
• La classe Java (nel nostro esempio: Citta) i cui oggetti possono essere legati a più oggetti dell'altra classe Java (nel nostro esempio: Persona) ha le seguenti caratteristiche:
– ha un ulteriore campo dato di tipo Set, per poter rappresentare tutti i link; l'oggetto di classe Set viene creato tramite il costruttore;
– ha tre campi funzione (inserisciLinkResidenza(), eliminaLinkResidenza() e getLinkResidenza()) per la gestione dell'insieme dei link; quest'ultima restituisce una copia dell'insieme dei link;
– ha i due campi funzioni speciali di ausilio per ManagerResidenza.
• La classe Java (nel nostro esempio: Persona) i cui oggetti possono essere legati al più ad un oggetto dell'altra classe Java (nel nostro esempio: Citta) è esattamente identica al caso di entrambe le molteplicità 0..1.
• Analogamente, la classe Java per la rappresentazione dei link per la rappresentazione di tuple del prodotto cartesiano tra Città e Persona, con attributo DaAnno (nel nostro esempio: TipoLinkResidenza) è esattamente identica al caso della molteplicità 0..1.

Classe Java Citta

```
// File RespEntrambi0STAR/Citta.java

import java.util.*;
public class Citta {
   private final String nome;
   private HashSet<TipoLinkResidenza> insieme_link;
   public Citta(String n) {
      nome = n;
      insieme_link = new HashSet<TipoLinkResidenza>()
   }
   public String getNome() { return nome; }
```

```
    public void inserisciLinkResidenza(TipoLinkResidenza t) {
        if (t != null && t.getCitta()==this)
            ManagerResidenza.inserisci(t);
    }
    public void eliminaLinkResidenza(TipoLinkResidenza t) {
        if (t != null && t.getCitta()==this)
            ManagerResidenza.elimina(t);
    }
    public Set<TipoLinkResidenza> getLinkResidenza() { return (HashSet<TipoLinkResidenza>)insieme_link.clone(); }
    public void inserisciPerManagerResidenza(ManagerResidenza a) {
        if (a != null)
          insieme_link.add(a.getLink());
    }
    public void eliminaPerManagerResidenza(ManagerResidenza a) {
        if (a != null)
          insieme_link.remove(a.getLink());
    }
}
```

Classe Java Persona

```
// File RespEntrambi0STAR/Persona.java

public class Persona {
    private final String nome;
    private TipoLinkResidenza link;
    public Persona(String n) { nome = n; }
    public String getNome() { return nome; }
    public void inserisciLinkResidenza(TipoLinkResidenza t) {
        if (t != null && t.getPersona()==this)
            ManagerResidenza.inserisci(t);
    }
    public void eliminaLinkResidenza(TipoLinkResidenza t) {
        if (t != null && t.getPersona()==this)
            ManagerResidenza.elimina(t);
    }
    public TipoLinkResidenza getLinkResidenza() { return link; }
    public void inserisciPerManagerResidenza(ManagerResidenza a) {
        if (a != null)
            link = a.getLink();
    }
    public void eliminaPerManagerResidenza(ManagerResidenza a) {
        if (a != null) link = null;
    }
}
```

Classe Java TipoLinkResidenza

```
// File RespEntrambi0STAR/TipoLinkResidenza.java

public class TipoLinkResidenza {
    private final Citta laCitta;
    private final Persona laPersona;
    private final int daAnno;
    public TipoLinkResidenza(Citta x, Persona y, int a) throws EccezionePrecondizioni {
        if (x  ==  null  || y  ==  null)   // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni ("Gli oggetti devono essere inizializzati");
        laCitta = x;
        laPersona = y;
        daAnno = a;
    }
    public boolean equals(Object o) {
        if (o != null && getClass().equals(o.getClass())) {
            TipoLinkResidenza b = (TipoLinkResidenza)o;
            return b.laPersona == laPersona && b.laCitta == laCitta;
    }
    else return false;
```

```
    }
    public int hashCode() { return laPersona.hashCode() + laCitta.hashCode();
    public Citta getCitta() { return laCitta; }
    public Persona getPersona() { return laPersona; }
    public int getDaAnno() { return daAnno; }
}
```

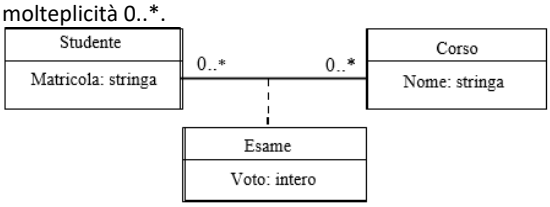
Classe Java ManagerResidenza

```
// File RespEntrambi0STAR/ManagerResidenza.java
public final class ManagerResidenza {
    private ManagerResidenza(TipoLinkResidenza x) { link = x; }
    private TipoLinkResidenza link;
    public TipoLinkResidenza getLink() { return link; }
    public static void inserisci(TipoLinkResidenza y) {
        if (y != null && y.getPersona().getLinkResidenza() == null) {
            ManagerResidenza k = new ManagerResidenza(y);
            y.getCitta().inserisciPerManagerResidenza(k);
            y.getPersona().inserisciPerManagerResidenza(k);
        }
    }
    public static void elimina(TipoLinkResidenza y) {
        if (y != null && y.getPersona().getLinkResidenza().equals(y)) {
            ManagerResidenza k = new ManagerResidenza(y);
            y.getCitta().eliminaPerManagerResidenza(k);
            y.getPersona().eliminaPerManagerResidenza(k);
        }
    }
}
```

Come per il caso di entrambe le molteplicità 0..1, anche in questo caso la funzione elimina() deve prendere opportuni provvedimenti al fine di evitare di eliminare link inesistenti.

Entrambe le molteplicità sono 0..*

Affrontiamo il caso di associazione binaria in cui entrambe le classi abbiano la responsabilità sull'associazione, ed entrambe con molteplicità 0..*.



Supponiamo che sia Studente sia Corso abbiano responsabilità sull'associazione.
La stessa metodologia proposta per il caso in cui entrambe le classi abbiano responsabilità sull'associazione può essere usata anche quando entrambe le molteplicità sono 0..*.
In particolare, ora le due classi Java sono strutturalmente simili:

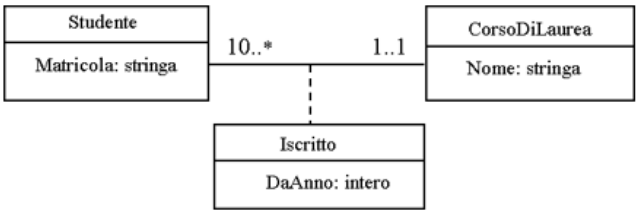
- hanno un ulteriore campo dato di tipo HashSet<TipoLinkEsame>,per poter rappresentare tutti i link; l'oggetto di classe HashSet<TipoLinkEsame> viene creato tramite il costruttore;
- hanno tre campi funzione (inserisciLinkResidenza(), eliminaLinkResidenza() e getLinkResidenza()) per la gestione dell'insieme dei link; quest'ultima restituisce una copia dell'insieme dei link;
- hanno i due campi funzioni speciali di ausilio per ManagerResidenza.

Altre molteplicità di associazione

Per quanto riguarda le altre molteplicità di associazione, tratteremo (brevemente) i seguenti due casi:
1. molteplicità minima diversa da zero;
2. molteplicità massima finita.

Come già chiarito nella fase di progetto, in generale prevediamo che la classe Java rispetto a cui esiste uno dei vincoli di cui sopra abbia necessariamente responsabilità sull'associazione. Il motivo, che verrà chiarito in seguito, è che gli oggetti di tale classe devono poter essere interrogati sul numero di link esistenti.
L'ideale sarebbe fare in modo che tutti i vincoli di molteplicità di un diagramma delle classi fossero rispettati in ogni momento. Ma ciò è, in generale, molto complicato.
La strategia che seguiremo semplifica il problema, ammettendo che gli oggetti possano essere in uno stato che non rispetta vincoli di molteplicità massima finita diversa da 1 e vincoli di molteplicità minima diversa da 0, ma lanciando una eccezione nel momento in cui un cliente chieda di utilizzare un link (relativo ad una associazione A) di un oggetto che non rispetta tali vincoli sull'associazione A.

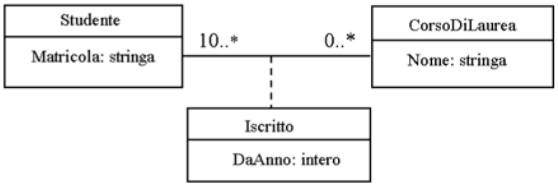
Considerazioni sulla molteplicità arbitrarie



Questo esempio dimostra bene il fatto che imporre che tutti i vincoli di molteplicità di un diagramma delle classi siano rispettati in ogni momento è, in generale, molto complicato.
Infatti, uno studente potrebbe nascere solamente nel momento in cui esiste già un corso di laurea, ma un corso di laurea deve avere almeno dieci studenti, e questo indica una intrinseca complessità nel creare oggetti, e al tempo stesso fare in modo che essi non violino vincoli di cardinalità minima. Problemi simili si hanno nel momento in cui i link vengono eliminati.
Come già detto, la strategia che seguiremo semplifica il problema, ammettendo che gli oggetti possano essere in uno stato che non rispetta il vincolo di molteplicità minima, ma lanciando una eccezione nel momento in cui un cliente chieda di utilizzare un link (relativo ad una associazione A) di un oggetto che non rispetta tale vincolo sull'associazione A.

Molteplicità minima diversa da zero

Consideriamo quindi la seguente versione semplificata del diagramma delle classi (si notino i diversi vincoli di molteplicità).



• Rispetto al caso di associazione con responsabilità doppia e in cui i vincoli di molteplicità siano entrambi 0..*, la classe Java CorsoDiLaurea si differenzia nei seguenti aspetti:
1. Ha un'ulteriore funzione pubblica int quantiIscritti(), che restituisce il numero di studenti iscritti per il corso di laurea oggetto di invocazione.
In questa maniera, il cliente si può rendere conto se il vincolo di molteplicità sia rispettato oppure no.
2. La funzione int getLinkIscritto() lancia una opportuna eccezione (di tipo EccezioneCardMin) quando l'oggetto di invocazione non rispetta il vincolo di cardinalità minima sull'associazione Iscritto.
Rimane invece inalterata, rispetto al caso di associazione con responsabilità doppia e vincoli di molteplicità entrambi 0..*, la metodologia di realizzazione delle seguenti classi Java:
• Studente, TipoLinkIscritto, EccezionePrecondizioni, ManagerIscritto.

Classe Java EccezioneCardMin

```
// File MoltMin/EccezioneCardMin.java
public class EccezioneCardMin extends Exception {
    private String messaggio;
    public EccezioneCardMin(String m) { messaggio = m; }
    public String toString() { return messaggio; }
}
```

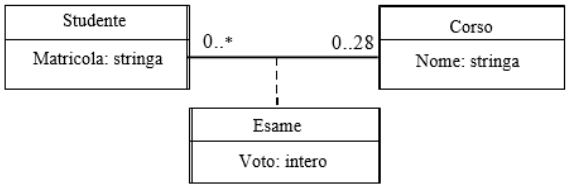
Classe Java CorsoDiLaurea

```
// File MoltMin/CorsoDiLaurea.java
import java.util.*;
public class CorsoDiLaurea {
    private final String nome;
    private HashSet<TipoLinkIscritto> insieme_link;
    public static final int MIN_LINK_ISCRITTO = 10;  // PER IL VINCOLO DI MOLTEPLICITA'
    public CorsoDiLaurea(String n) {
        nome = n;
        insieme_link = new HashSet<TipoLinkIscritto>();
    }
    public String getNome() { return nome; }
    public int quantiIscritti() { // FUNZIONE NUOVA
        return insieme_link.size();
    }
    public void inserisciLinkIscritto(TipoLinkIscritto t) {
        if (t != null && t.getCorsoDiLaurea()==this)
            ManagerIscritto.inserisci(t);
    }
```

```
   public void eliminaLinkIscritto(TipoLinkIscritto t) {
      if (t != null && t.getCorsoDiLaurea()==this)
          ManagerIscritto.elimina(t);
   }
   public Set<TipoLinkIscritto> getLinkIscritto() throws EccezioneCardMin {
      if (quantiIscritti() < MIN_LINK_ISCRITTO)
         throw new EccezioneCardMin("Cardinalita' minima violata");
      else return (HashSet<TipoLinkIscritto>)insieme_link.clone();
   }
   public void inserisciPerManagerIscritto(ManagerIscritto a) {
      if (a != null)
        insieme_link.add(a.getLink());
   }
   public void eliminaPerManagerIscritto(ManagerIscritto a) {
      if (a != null)
          insieme_link.remove(a.getLink());
   }
}
```

Molteplicità massima finita

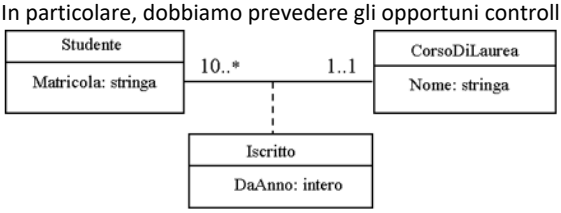
Supponiamo che sia Studente sia Corso abbiano responsabilità sull'associazione.



• Rispetto al caso di associazione con responsabilità doppia e in cui i vincoli di molteplicità siano entrambi 0..*, la classe Java Studente si differenzia nei seguenti aspetti:
1. Ha un'ulteriore funzione pubblica int quantiEsami(), che restituisce il numero di esami sostenuti dallo studente oggetto di invocazione. In questa maniera, il cliente si può rendere conto se sia possibile inserire un nuovo esame senza violare i vincoli di molteplicità oppure no.
2. La funzione int getLinkEsami() lancia una opportuna eccezione (di tipo EccezioneCardMax) quando l'oggetto di invocazione non rispetta il vincolo di cardinalità massima sull'associazione Iscritto.
Rimangono invece inalterate, rispetto al caso di associazione con responsabilità doppia e vincoli di molteplicità entrambi 0..* le seguenti classi Java:
• Corso, TipoLinkEsame, EccezionePrecondizioni, ManagerEsame.

Molteplicità massima 1

Un caso particolare di molteplicità massima finita si ha quando essa sia pari a 1. In tal caso dobbiamo gestire l'associazione secondo il modello e le strutture di dati visti in precedenza per il vincolo di molteplicità 0..1.
In particolare, dobbiamo prevedere gli opportuni controlli per la classe che gestisce l'associazione.



Prenderemo in considerazione il diagramma delle classi già visto in precedenza, nella sua versione non semplificata.
Notiamo che dai vincoli di molteplicità si evince che sia Studente sia Corso hanno responsabilità sull'associazione.

Classe Java Studente

```
// File MoltMax1/Studente.java
public class Studente {
    private final String matricola;
    private TipoLinkIscritto link;
    public static final int MIN_LINK_ISCRITTO = 1;
    public Studente(String n) { matricola =  n; }
    public String getMatricola() { return matricola; }
    public int quantiIscritti() { // FUNZIONE NUOVA
        if (link == null) return 0;
        else return 1;
}
```

```
   public void inserisciLinkIscritto(TipoLinkIscritto t) {
      if (t != null && t.getStudente()==this)
           ManagerIscritto.inserisci(t);
   }
   public void eliminaLinkIscritto(TipoLinkIscritto t) {
      if (t != null && t.getStudente()==this)
            ManagerIscritto.elimina(t);
   }
   public TipoLinkIscritto getLinkIscritto() throws EccezioneCardMin {
      if (link == null)
         throw new EccezioneCardMin("Cardinalita' minima violata");
      else return link;
   }
   public void inserisciPerManagerIscritto(ManagerIscritto a) {
      if (a != null) link = a.getLink();
   }
   public void eliminaPerManagerIscritto(ManagerIscritto  a)  {
      if (a != null) link = null;
   }
}
```

Classe Java ManagerIscritto

```
// File MoltMax1/ManagerIscritto.java
public final class ManagerIscritto {
    private ManagerIscritto(TipoLinkIscritto x) { link = x; }
    private TipoLinkIscritto link;
    public TipoLinkIscritto getLink() { return link; }
    public static void inserisci(TipoLinkIscritto y) {
       if (y != null && y.getStudente().quantiIscritti() == 0) {
            ManagerIscritto k = new ManagerIscritto(y);
            k.link.getCorsoDiLaurea().inserisciPerManagerIscritto(k);
            k.link.getStudente().inserisciPerManagerIscritto(k);
       }
   }
   public static void elimina(TipoLinkIscritto y) {
      try {
          if (y != null && y.getStudente().getLinkIscritto().equals(y) ) {
               ManagerIscritto k = new ManagerIscritto(y);
               k.link.getCorsoDiLaurea().eliminaPerManagerIscritto(k);
               k.link.getStudente().eliminaPerManagerIscritto(k);
          }
      } catch (EccezioneCardMin e) { System.out.println(e); }
   }
}
```

Notiamo che la classe Studente ha ovviamente un campo dato di tipo TipoLinkIscritto (e non Set) e che la classe ManagerIscritto deve effettuare i controlli del caso, in particolare:
• che l'inserimento avvenga solo se lo studente non è iscritto (sfruttando la funzione quantiIscritti() di Studente),
• che la cancellazione avvenga solo per link esistenti.
Rimangono invece inalterate le seguenti classi Java:
• TipoLinkIscritto/EccezionePrecondizioni (realizzate come sempre),
• CorsoDiLaurea (realizzata come nel caso semplificato visto in precedenza in cui lo studente può essere iscritto ad un numero qualsiasi di corsi di laurea).

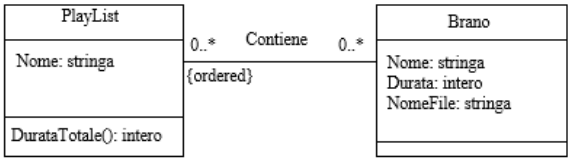
Associazioni n-arie

Si trattano generalizzando quanto visto per le associazioni binarie.
Ricordiamo che noi assumiamo che le molteplicità delle associazioni n-arie siano sempre 0..*.
In ogni caso, per un'associazione n-aria A, anche se non ha attributi, si definisce la corrispondente classe TipoLinkA.
Nel caso di responsabilità di una sola classe, si prevede la struttura di dati per rappresentare i link solo in quella classe.
Nel caso di responsabilità di più classi, si definisce anche la classe ManagerA, secondo le regole viste per le associazioni binarie.

Associazioni Ordinate

• Le associazioni ordinate si realizzano in modo del tutto analogo alle relazioni non ordinate.
• Per mantenere l'ordinamento però si fa uso delle classi List e LinkedList invece di Set e HashSet.
• Si noti che l'ordinamento sulle associazini è realizzato con List e LinkedList e non con una struttura dati per insiemi ordinati come OrderedSet e TreeSet, perchè l'ordinamento non nasce da proprietà degli elementi dell'insieme (cioè dei link)
ma viene definito esternamente ed essi come appunto succede quando mettiamo degli elementi in una lista.
• Si noti inoltre che nel memorizzare i link in una lista dobbiamo stare attenti a non avere ripetizioni perchè in una associazione, anche se ordinata, non ci possono essere due link uguali.

Esempio con responsabilità singola



Assumiamo di avere stabilito, nella fase di progetto, che:
• il nome di una playlist, e il nome, la durata ed il nome del file associati ad un brano non cambiano;
• solo PlayList ha responsabilità sull'associazione (ci interessa conoscere quali brani sono contenuti in una playlist, ma non ci interessa conoscere le playlist che contengono un dato brano).

Specifica della classe UML PlayList

InizioSpecificaClasse PlayList
    durataTotale (): intero
        pre: nessuna
        post: result è pari alla somma delle durate dei brani contenuti in this
FineSpecifica

Realizzazione in Java della classe PlayList

```
// File Ordinate0STAR/PlayList.java
import java.util.*;
public class PlayList {
    private final String nome;
    private LinkedList<Brano> insieme_link;
    public PlayList(String n) {
        nome = n;
        insieme_link = new LinkedList<Brano>();
   }
   public String getNome() { return nome; }
   public void inserisciLinkContiene(Brano b) {
       if (b != null && !insieme_link.contains(b))
           insieme_link.add(b);
   }
   public void eliminaLinkContiene(Brano b) {
       if (b != null) insieme_link.remove(b);
   }
   public List<Brano> getLinkContiene() {
       return (LinkedList<Brano>)insieme_link.clone();
   }
   public int durataTotale() {
       int result = 0;
       Iterator<Brano> ib = insieme_link.iterator();
       while (ib.hasNext()) {
           Brano b = ib.next();
           result = result + b.getDurata();
       }
       return result;
   }
}
```

Realizzazione in Java della classe Brano

```
// File Ordinate0STAR/Brano.java
public class Brano {
    private final String nome;
    private final int durata;
    private final String nomefile;
    public Brano(String n, int d, String f) {
        nome = n;
        durata = d;
        nomefile = f;
   }
   public String getNome() { return nome; }
   public int getDurata() { return durata; }
   public String getNomeFile() { return nomefile; }
}
```

Un cliente

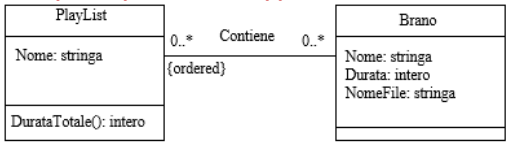
Realizziamo ora in Java il cliente Analisi PlayList, specificato di seguito:

InizioSpecificaCliente Analisi PlayList
Più Lunghe (i: Insieme(PlayList)): Insieme(PlayList)
pre: nessuna
post: result è costituito dalle PlayList di i la cui durata totale è massima
FineSpecifica

Realizzazione in Java del cliente

```
// File Ordinate0STAR/AnalisiPlayList.java
import java.util.*;
public final class AnalisiPlayList {
    public static Set<PlayList> piuLunghe(Set<PlayList> i) {
       HashSet<PlayList> result = new HashSet<PlayList>();
       int duratamax = maxDurata(i);
       Iterator<PlayList> it = i.iterator();
       while(it.hasNext()) {
          PlayList pl = it.next();
          int durata = pl.durataTotale();
          if (durata == duratamax) result.add(pl);
       }
       return result;
   }
   private static int maxDurata(Set<PlayList> i) {
       int duratamax = 0;
       Iterator<PlayList> it = i.iterator();
       while(it.hasNext()) {
           PlayList pl = it.next();
           int durata = pl.durataTotale();
           if (durata > duratamax) duratamax = durata;
       }
       return duratamax;
   }
}
```

Esempio responsabilità doppia



Assumiamo di avere stabilito, nella fase di progetto, che:

• il nome di una playlist, e il nome, la durata ed il nome del file associati ad un brano non cambiano;

• sia PlayList che Brano hanno responsabilità sull'associazione (ci interessa conoscere sia quali brani sono contenuti in una playlist, che le playlist che contengono un dato brano).

• In questo caso dobbiamo adattare la metodologia generale, prevedendo:

– la realizzazione della classe TipoLinkContiene,

– la realizzazione della classe ManagerContiene,

– che la classe Brano abbia un campo dato di tipo HashSet, per rappresentare la struttura di dati non ordinata,

– che la classe PlayList abbia un campo dato di tipo LinkedList, per rappresentare la struttura di dati ordinata.

Realizzazione in Java della classe PlayList

```
// File OrdinateEntrambi0STAR/PlayList.java
import java.util.*;
public class PlayList {
    private final String nome;
    private LinkedList<TipoLinkContiene> insieme_link;
    public PlayList(String n) {
        nome = n;
        insieme_link = new LinkedList<TipoLinkContiene>();
   }
   public String getNome() { return nome; }
   public void inserisciLinkContiene(TipoLinkIscritto t) {
       if (t != null && t.getPlayList()==this)
           ManagerContiene.inserisci(t);
   }
   public void eliminaLinkContiene(TipoLinkIscritto t) {
       if (t != null && t.getPlayList()==this)
           ManagerContiene.elimina(t);
   }
   public List<TipoLinkContiene> getLinkContiene() { return (LinkedList<TipoLinkContiene>)insieme_link.clone(); }
   public void inserisciPerManagerContiene(ManagerContiene a){
       if (a != null && !insieme_link.contains(a.getLink()))
          insieme_link.add(a.getLink());
   }
   public void eliminaPerManagerContiene(ManagerContiene a) {
       if (a != null) insieme_link.remove(a.getLink());
   }
   public int durataTotale() {
       int result = 0;
       Iterator<TipoLinkContiene> il = insieme_link.iterator();
       while (il.hasNext()) {
            Brano b = il.next().getBrano();
            result = result + b.getDurata();
       }
       return result;
   }
}
```

Realizzazione in Java della classe Brano

```
// File OrdinateEntrambi0STAR/Brano.java
import java.util.*;
public class Brano {
    private final String nome;
    private final int durata;
    private final String nomefile;
    private HashSet<TipoLinkContiene> insieme_link;
    public Brano(String n, int d, String f) {
        nome = n;
        durata = d;
        nomefile = f;
        insieme_link = new HashSet<TipoLinkContiene>();
   }
   public String getNome() { return nome; }
   public int getDurata() { return durata; }
   public String getNomeFile() { return nomefile; }
   public void inserisciLinkContiene(TipoLinkContiene t) {
       if (t != null && t.getBrano()==this)
           ManagerContiene.inserisci(t);
   }
   public void eliminaLinkContiene(TipoLinkContiene t) {
       if (t != null && t.getBrano()==this)
          ManagerContiene.elimina(t);
   }
```

```
   public Set<TipoLinkContiene> getLinkContiene() { return (HashSet<TipoLinkContiene>)insieme_link.clone(); }
   public void inserisciPerManagerContiene(ManagerContiene a){
      if (a != null) insieme_link.add(a.getLink());
   }
   public void eliminaPerManagerContiene(ManagerContiene a) { if (a != null) insieme_link.remove(a.getLink()); }
}
```

Realizzazione della classe TipoLinkContiene

```
// File OrdinateEntrambi0STAR/TipoLinkContiene.java
public class TipoLinkContiene {
    private final PlayList laPlayList;
    private final Brano ilBrano;
    public TipoLinkContiene(PlayList x, Brano y) throws EccezionePrecondizioni {
        if (x  ==  null  || y  ==  null) // CONTROLLO PRECONDIZIONI
            throw new EccezionePrecondizioni ("Gli oggetti devono essere inizializzati");
        laPlayList = x;
        ilBrano = y;
   }
   public boolean equals(Object o) {
      if (o != null && getClass().equals(o.getClass())) {
           TipoLinkContiene b = (TipoLinkContiene)o;
           return b.ilBrano == ilBrano && b.laPlayList == laPlayList;
      }
      else return false;
   }
   public int hashCode() { return laPlayList.hashCode() + ilBrano.hashCode(); }
   public PlayList getPlayList() { return laPlayList; }
   public Brano getBrano() { return ilBrano; }
}
```

Realizzazione della classe ManagerContiene

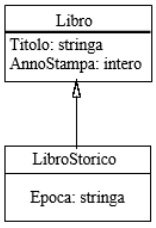
```
// File OrdinateEntrambi0STAR/ManagerContiene.java
public final class ManagerContiene {
    private ManagerContiene(TipoLinkContiene x) { link = x; }
    private TipoLinkContiene link;
    public TipoLinkContiene getLink() { return link; }
    public static void inserisci(TipoLinkContiene y) {
        if (y != null) {
            ManagerContiene k = new ManagerContiene(y);
            y.getPlayList().inserisciPerManagerContiene(k);
            y.getBrano().inserisciPerManagerContiene(k);
        }
   }
   public static void elimina(TipoLinkContiene y) {
        if (y != null) {
            ManagerContiene k = new ManagerContiene(y);
            y.getPlayList().eliminaPerManagerContiene(k);
            y.getBrano().eliminaPerManagerContiene(k);
        }
   }
}
```

Realizzazione di generalizzazioni

Nell'esposizione di questo argomento, seguiremo quest'ordine:

- relazione is-a fra due classi;
- specializzazione di operazioni;
- generalizzazioni disgiunte e complete.

Affrontiamo ora il caso in cui abbiamo una generalizzazione nel diagramma delle classi.



1. La superclasse UML (Libro) diventa una classe base Java (Libro), e la sottoclasse UML (LibroStorico) diventa una classe derivata Java (LibroStorico). Infatti, poichè ogni istanza di LibroStorico è anche istanza di Libro, vogliamo:
• poter usare un oggetto della classe LibroStorico ogni volta che è lecito usare un oggetto della classe Libro, e dare la possibilità ai clienti della classe LibroStorico di usare le funzioni pubbliche di Libro.
2. Poichè ogni proprietà della classe Libro è anche una proprietà del tipo LibroStorico, in Libro tutto ciò che si vuole ereditare è protetto.
Si noti che la possibilità di utilizzare la parte protetta di Libro implica che il progettista della classe LibroStorico (e delle classi eventualmente derivate da LibroStorico) deve avere una buona conoscenza dei metodi di rappresentazione e delle funzioni della classe Libro.
3. Nella classe LibroStorico:
• ci si affida alla definizione di Libro per quelle proprietà (ad es., AnnoStampa, Titolo) che sono identiche per gli oggetti della classe LibroStorico;
• si definiscono tutte le proprietà (dati e funzioni) che gli oggetti di LibroStorico hanno in più rispetto a quelle ereditate da Libro.

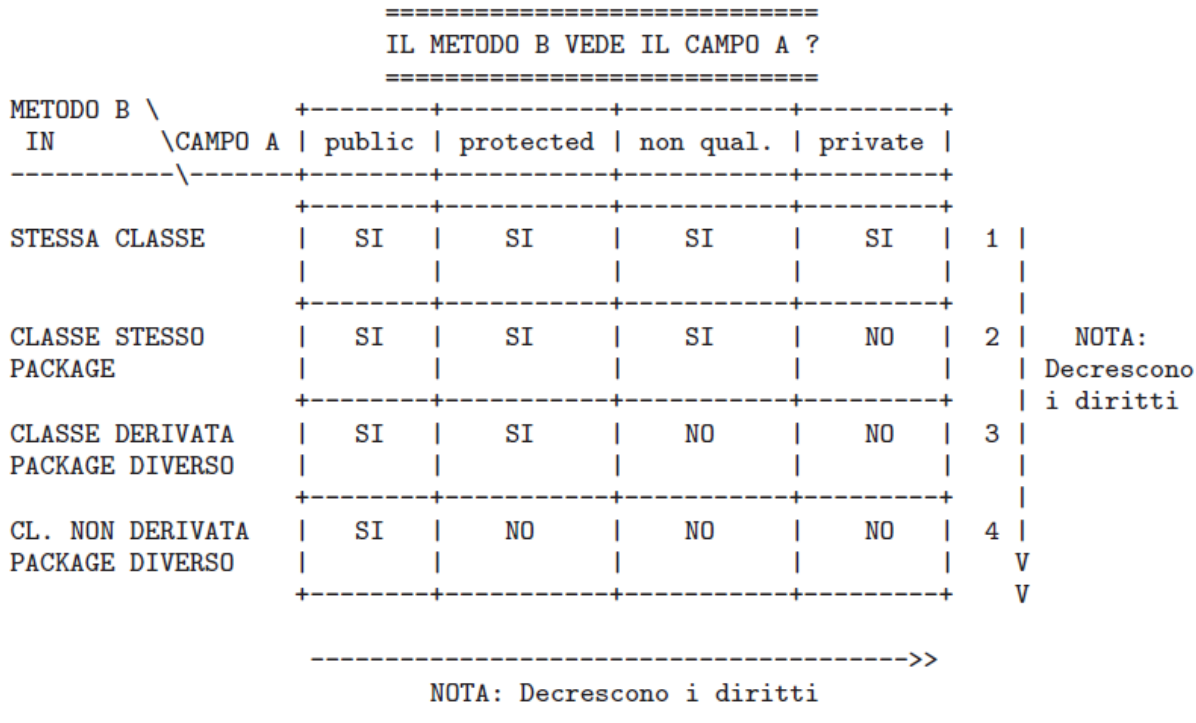
Information hiding

Fino ad ora abbiamo seguito il seguente approccio per garantire un alto livello di information hiding nella realizzazione di una classe UML C mediante una classe Java C:
• gli attributi di C corrispondono a campi privati della classe Java C;
• le operazioni di C corrispondono a campi pubblici di C;
• sono pubblici anche i costruttori di C e le funzioni get e set;
• sono invece private eventuali funzioni che dovessero servire per la realizzazione dei metodi della classe C (ma che non vogliamo rendere disponibili ai clienti), e i campi dati per la realizzazione di associazioni;
• tutte le classi Java sono nello stesso package (senza nome).
Nell'ambito della realizzazione di generalizzazioni, è più ragionevole che i campi di C che non vogliamo che i clienti possano vedere siano protetti, e non privati. Infatti, in questa maniera raggiungiamo un duplice scopo:
1. continuiamo ad impedire ai clienti generici di accedere direttamente ai metodi di rappresentazione e alle strutture di dati, mantenendo così alto il livello di information hiding;
2. diamo tale possibilità ai progettisti delle classi derivate da C (che non devono essere considerati clienti qualsiasi) garantendo in tal modo maggiore efficienza.

Ripasso: livelli di accesso nelle classi Java

Un campo di una classe (dato, funzione o classe) può essere specificato con uno fra quattro livelli di accesso:
A. public,
B. protected,
C. non qualificato (è il default, intermedio fra protetto e privato),
D. private.
Anche un'intera classe C (solo se non è interna ad altra classe) può essere dichiarata public, ed in tale caso la classe deve essere dichiarata nel file C.java.

Classi: regole di visibilità



Information hiding e generalizzazione

Occorre tenere opportunamente conto delle regole di visibilità di Java, che garantiscono maggiori diritti ad una classe di uno stesso package, rispetto ad una classe derivata, ma di package diverso.
Non possiamo più, quindi, prevedere un solo package per tutte le classi Java, in quanto sarebbe vanificata la strutturazione in parte pubblica e parte protetta, poichè tutte le classi (anche quelle non derivate) avrebbero accesso ai campi protetti.
Da ciò emerge la necessità di prevedere un package diverso per ogni classe Java che ha campi protetti (tipicamente, ciò avviene quando la corrispondente classe UML fa parte di una gerarchia).

Generalizzazione e strutturazione in package

In particolare, seguiremo le seguenti regole:
• continueremo per il momento ad assumere di lavorare con il package senza nome
• per ogni classe Java C che ha campi protetti prevediamo un package dal nome C, realizzato nel direttorio C, che contiene solamente il file dal nome C.java;
• ogni classe Java D che deve accedere ai campi di C conterrà la dichiarazione import C.*;

La classe Java Libro

```
// File Generalizzazione/Libro/Libro.java

package Libro;
public class Libro {
    protected final String titolo;
    protected final int annoStampa;
    public Libro(String t, int a) {
        titolo = t;
        annoStampa = a;}
   public String getTitolo() { return titolo; }
   public int getAnnoStampa() { return annoStampa; }
   public String toString() { return titolo + ", dato alle stampe nel " + annoStampa; }
}
```

Costruttori di classi derivate

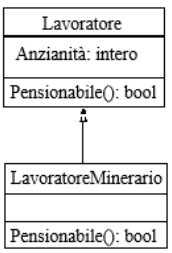
Comportamento di un costruttore di una classe D derivata da B:
1. se ha come prima istruzione super(), allora viene chiamato il costruttore di B esplicitamente invocato; altrimenti viene chiamato il costruttore senza argomenti di B;
2. viene eseguito il corpo del costruttore.
Questo vale anche per il costruttore standard di D senza argomenti (come al solito, disponibile se e solo se in D non vengono definiti esplicitamente costruttori).

Ridefinizione

Nella classe derivata è possibile fare overriding (dall'inglese, ridefinizione, sovrascrittura) delle funzioni della classe base.
Fare overriding di una funzione f() della classe base B vuol dire definire nella classe derivata D una funzione f() in cui sono uguali il numero e il tipo degli argomenti, mentre il tipo di ritorno deve essere identico.
Nella classe Java derivata si ridefinisce una funzione F() già definita nella classe base ogni volta che F(), quando viene eseguita su un oggetto della classe derivata, deve compiere operazioni diverse rispetto a quelle della classe base, ad esempio operazioni che riguardano le proprietà specifiche che la classe derivata possiede rispetto a quelle definite per quella base.

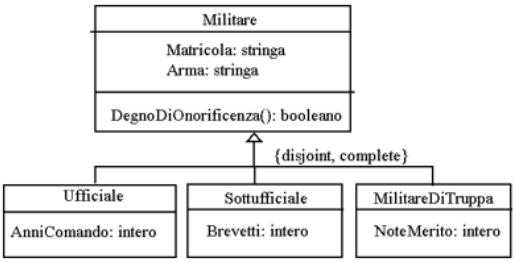
Esempio

I lavoratori sono pensionabili con un'anzianità di 30 anni.
I lavoratori minerari sono pensionabili con un'anzianità di 25 anni



Generalizzazioni disgiunte e complete

Poichè Java non supporta l'ereditarietà multipla, assumiamo che ogni generalizzazione sia disgiunta (ciò può essere ottenuto mediante opportune trasformazioni, come descritto nella parte del corso dedicata all'analisi).
Quando la generalizzazione è anche completa, occorre fare delle considerazioni ulteriori.



Il diagramma delle classi ci dice che non esistono istanze di Militare che non siano istanze di almeno una delle classi Ufficiale, Sottufficiale o MilitareDiTruppa.
Per questo motivo la classe Java Militare deve essere una abstract class. La definizione di Militare come classe base astratta consente di progettare clienti che astraggono rispetto alle peculiarità delle sue sottoclassi.
In questo modo, infatti, non si potranno definire oggetti che sono istanze dirette della classe Militare.
Viceversa, le classi Java Ufficiale, Sottufficiale e MilitareDiTruppa saranno classi non abstract (a meno che siano anch'esse superclassi per generalizzazioni disgiunte e complete).

Funzioni Java non astratte

Alcune proprietà della classe UML Militare, come ad esempio l'attributo "Arma", sono dettagliabili completamente al livello della classe stessa.
La gestione di queste proprietà verrà realizzata tramite funzioni non abstract della classe Java Militare.

Funzioni Java astratte

Tra le operazioni che associamo a Militare ve ne possono essere invece alcune che sono dettagliabili solo quando vengono associate ad una delle sottoclassi.
Ad esempio, l'operazione che determina se un militare è degno di onoreficenza potrebbe dipendere da parametri relativi al fatto se esso è ufficiale, sottufficiale oppure di truppa. L'operazione DegnoDiOnoreficenza si può associare alla classe Militare solo concettualmente, mentre il calcolo che essa effettua si può rappresentare in modo preciso solo al livello della sottoclasse.
La corrispondente funzione Java verrà dichiarata come abstract nella classe Militare. La sua definizione viene demandata alle classi java Ufficiale, Sottufficiale o MilitareDiTruppa.

La classe astratta Java Militare

```
// File Generalizzazione/Militare/Militare.java
package Militare;
public abstract class Militare {
    protected String arma;
    protected String matricola;
    public Militare(String a, String m) {
        arma = a;
        matricola = m; }
    public String getArma() { return arma; }
    public String getMatricola() { return matricola; }
    abstract public boolean degnoDiOnoreficenza();
    public String toString() { return "Matricola: " + matricola + ". Arma di appartenenza:  " + arma; }
}
```

Un cliente della classe astratta

```
public static void stampaStatoDiServizio(Militare mil) { System.out.println("========== FORZE ARMATE ========== ");
System.out.println("STATO DI SERVIZIO DEL MILITARE");
System.out.println(mil);
if (mil.degnoDiOnoreficenza())
     System.out.println("SI E' PARTICOLARMENTE DISTINTO IN SERVIZIO");
}
```

La classe Java Ufficiale

```
// File Generalizzazione/Ufficiale/Ufficiale.java
package Ufficiale;
import Militare.*;
public class Ufficiale extends Militare {
    protected int anni_comando;
    public Ufficiale(String a, String  m)  {  super(a,m);  }
    public int getAnniComando() { return anni_comando; }
    public void incrementaAnniComando() { anni_comando++; }
    public boolean degnoDiOnoreficenza() {return anni_comando > 10; }
}
```

La classe Java Sottufficiale

```
// File Generalizzazione/Sottufficiale/Sottufficiale.java
package Sottufficiale;
import Militare.*;
public class Sottufficiale extends Militare {
    protected int brevetti_specializzazione;
    public Sottufficiale(String a, String m) { super(a,m); }
    public int getBrevettiSpecializzazione() { return brevetti_specializzazione; }
    public void incrementaBrevettiSpecializzazione() { brevetti_specializzazione++; }
    public boolean degnoDiOnoreficenza() { return brevetti_specializzazione > 4; }
}
```

La classe Java MilitareDiTruppa

```
// File Generalizzazione/MilitareDiTruppa/MilitareDiTruppa.java
package MilitareDiTruppa;
import Militare.*;
public class MilitareDiTruppa extends Militare {
    protected int note_di_merito;
    public MilitareDiTruppa(String a, String m) { super(a,m); }
    public int  getNoteDiMerito() { return note_di_merito; }
    public void incrementaNoteDiMerito() { note_di_merito++; }
    public boolean degnoDiOnoreficenza() { return note_di_merito > 2; }
}
```

Organizzazione in packages

Per evitare ogni potenziale conflitto sull'uso degli identificatori di classe, è possibile strutturare i file sorgente in package.
Una regola possibile è la seguente:

- Tutta l'applicazione viene messa in un package Java P, nel direttorio P.
- Ogni classe Java dell'applicazione proveniente dal diagramma delle classi (anche quelle definite per le associazioni) viene messa nel package P.
- Ciò vale anche per quelle definite per i tipi, a meno che siano in opportuni direttori resi accessibili mediante la variabile d'ambiente classpath.
- Nel caso di classi con campi protetti, vanno previsti sottodirettori e sottopackage, come visto in precedenza.

LEZIONE 3 – DIAGRAMMA DELLE ATTIVITA'

Processi

Accanto a rappresentare quali sono le informazioni di interesse nel dominio della nostra applicazione, dobbiamo anche rappresentare come la nostra applicazione accede, modifica, usa tali informazioni.
Cioè dobbiamo analizzare i processi che sono di interesse per la nostra applicazione. La descrizione dei processi si ottiene attraverso l'interazione tra analista ed utente in fase di analisi.
In UML per descrivere i processi si usa il diagramma delle attività.
Il diagramma delle attività descrive le attività che il sistema deve supportare in termini di:

- **Attività atomiche** realizzate nel sistema e **Flusso lavoro** (workflow) in cui sono coinvolte.

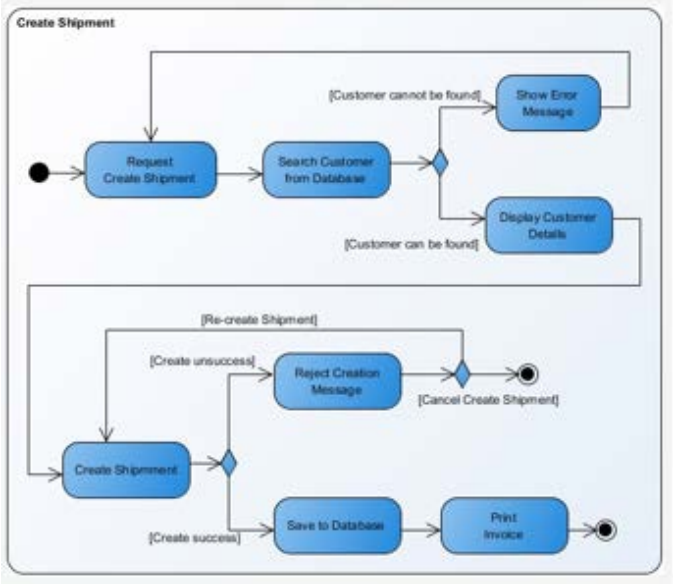
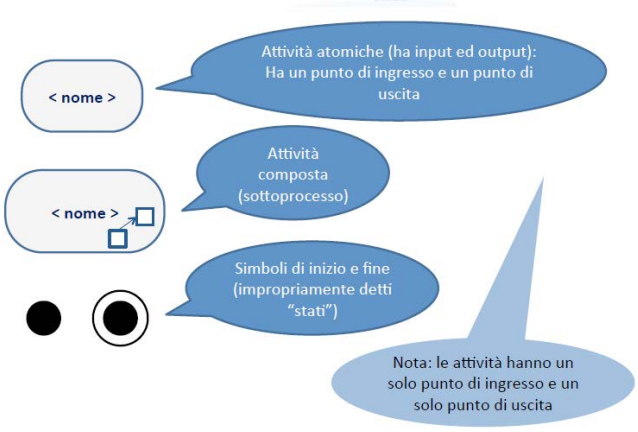
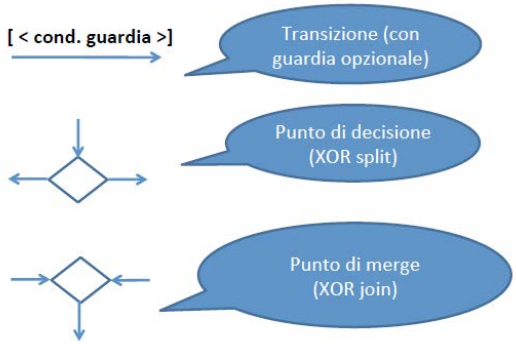


Diagramma delle Attività

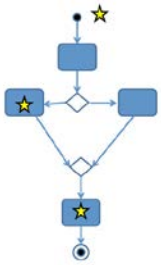
Le attività rappresentano le operazioni e le processi (organizzativi, tecnici, ecc.) che il sistema deve supportare
Le attività agiscono, accedono e aggiornano, l'istanziazione del diagramma delle classi, modificandone lo stato corrente del sistema, cioè le informazioni memorizzate nel sistema stesso.
Inoltre, le attività accedono all'esterno del sistema per fornire e ricevere informazioni dai suoi utilizzatori.
Si noti che anche un'attività è trasversale rispetto alle classi, cioè coinvolge più classi.
Un diagramma delle attività è composto:
• da diverse attività atomiche, che rappresentano operazioni (che chiameremo task) sul dominio dell'applicazione, così come l'abbiamo modellato nel diagramma delle classi
• dal flusso di controllo tra queste attività atomiche
• da segnali di I/O (input/output) che realizzano l'interfacciamento (attraverso una qualche interfaccia utente testuale, grafica, ecc.) del sistema con l'utente.



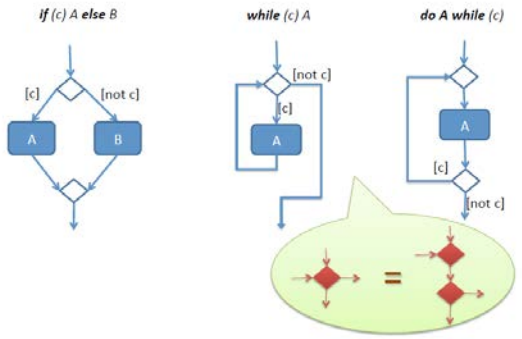
Costrutti per controllo di flusso sequenziale



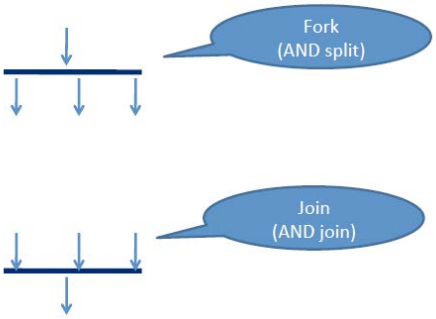
Esempio flusso sequenziale



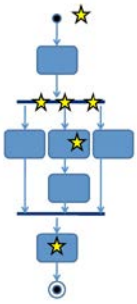
If-else, while, do-while



Costrutti per il controllo di flusso concorrente



Esempio flusso concorrente



Significato del Fork/Join

Per ogni flusso (**thread**) di esecuzione abbiamo un "esecutore" che è in carico di gestore l'avanzamento del processo

- Nel **fork**, il numero dei thread viene aumentato, in quanto prima era unico e dopo ce ne è uno per ogni transizione in uscita
- Nel **join** il numero dei thread viene diminuito, in quanto prima ce ne erano tanti (uno per ogni transizione in entrata) e dopo ce ne è uno solo: il thread d'uscita
- Il **thread d'uscita** può partire solo dopo che sono terminati tutti quelli che devono fare join
- Cioè **sincronizza** l'esecuzione dei vari rami del processo

Differenza tra Fork/Join e Punto di Decisione/Merge

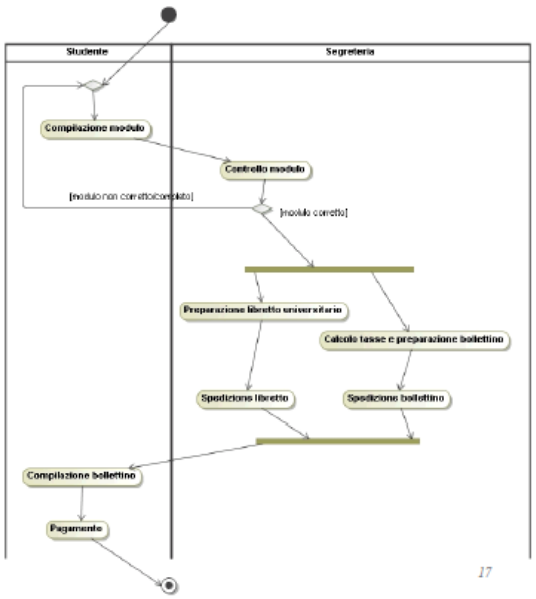
Attenzione, alla differenza tra il fork (anche detto and-split) e il punto di decisione (anche detto xor-split). In quest'ultimo caso, solo una delle transizioni in uscita viene eseguita, mentre nel fork tutte le transizioni di uscita vengono eseguite contemporaneamente. Una simile differenza sussiste anche tra il join (anche detto and-join) e il merge (anche detto xor-join). Nel join tutte le transizioni entranti devono essere completate per proseguire, mentre nel merge, solo una transizione entrante è significativa.

Swimlane

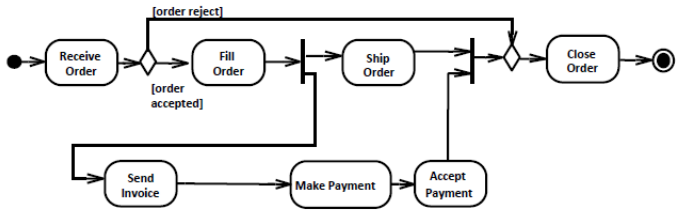
- Osservazione: il diagramma non evidenzia chi deve fare cosa (ovvero gli attori che eseguono le attività).
- Se si vuole mettere in evidenza chi fa cosa si devono introdurre le swimlane.
- Una swimlane ("corsia di nuoto") permette di evidenziare quali azioni sono eseguite da un dato attore

– Graficamente è una linea verticale che identifica la "corsia"

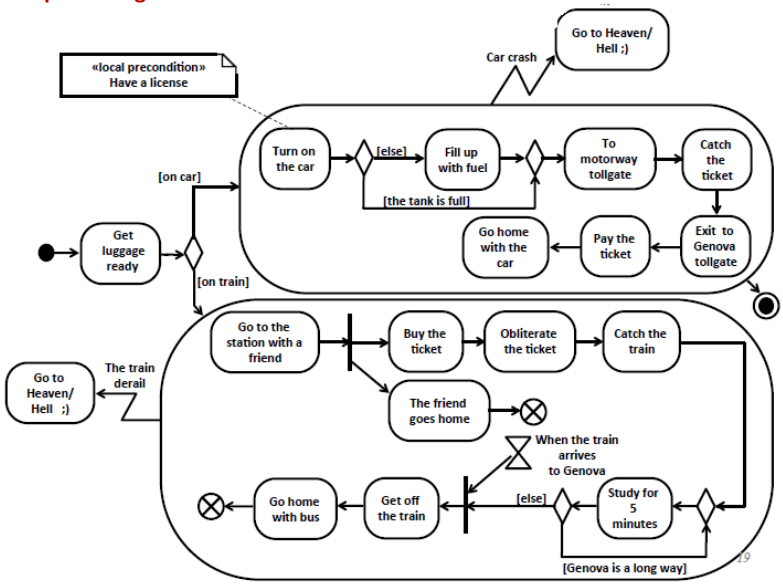
Esempio



Esempio: Orders



Esempio: Going to Genova



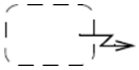
FlowFinal

• A volte serve segnalare che un ramo concorrente della computazione va a chiudersi.

• Per fare ciò si usa il simbolo di terminazione di un ramo concorrente: FlowFinal 

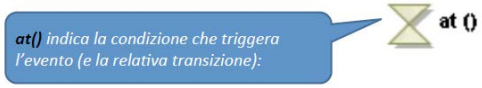
Costrutti per gli eventi

• Regione interrompibile
  – Raggruppamento di attività e transizioni che supporta la terminazione del flusso di esecuzione
  – Se la terminazione avviene, tutto il flusso ed il comportamento dovuto alle attività interne alla regione è terminato



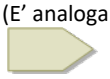
• Evento

– genera un evento per sincronizzarsi con una condizione esterna (es temporale)



Input/output verso l'esterno

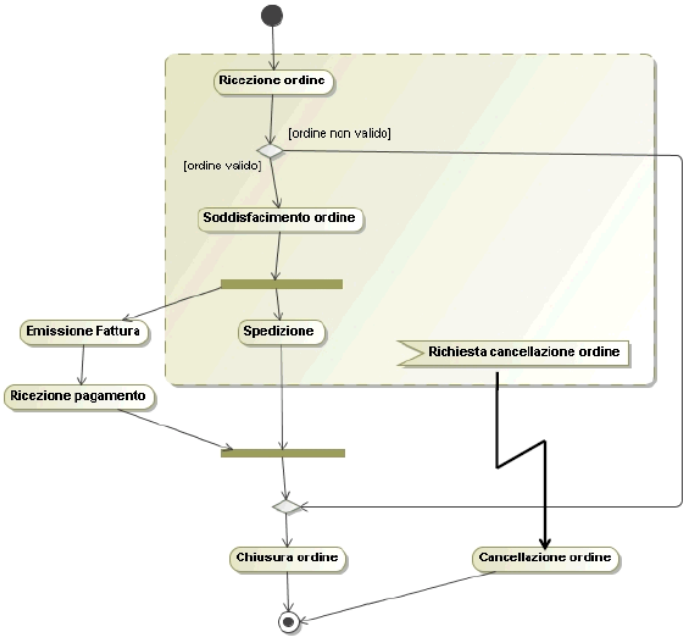
• Invia segnale (in output): si spedisce un segnale (con payload, cioè con dato!) all'esterno e si prosegue, senza aspettare risposte. (E' analoga ad una istruzione di write.)



•Ricevi segnale (in input): si rimane bloccatti sull'azione finché non arriva il segnale (con payload!) poi si prosegue (E' analoga ad una istruzione di read.)

Esempio – “ordini”

• Rappresentare il processo di gestione degli ordini. Dopo la ricezione di un ordine, se esso viene accettato si procede al soddisfacimento dello stesso, e quindi si attivano in parallelo la spedizione dell’ordine e la preparazione della fattura. A pagamento ricevuto, l’ordine può essere chiuso. In un qualsiasi momento prima dell’emissione della fattura, l’ordine può essere cancellato, e quindi chiuso.



LEZIONE 3 – Diagrammi Attivita Specifica

Specifica di classi e attività

Il diagramma delle classi e il diagramma delle attvità vengono corredati da:

• una specifica per ogni classe

• una specifica per ogni attività

La specifica di una classe ha lo scopo di definire precisamente il comportamento di ogni operazione della classe

La specifica di un’attività ha lo scopo di definire precisamente il comportamento di ogni operazione di cui l’attvità è costituito, e dell’attività nel suo complesso (variabili e flusso di controllo)

Specifica di una classe

InizioSpecificaClasse C

    Specifica della operazione 1

    …

    Specifica della operazione N

FineSpecifica

Specifica di una operazione

La specifica di una operazione di classe ha la seguente forma:

        alfa (X1: T1, … , Xn: Tn): T

            pre: condizione

            post: condizione

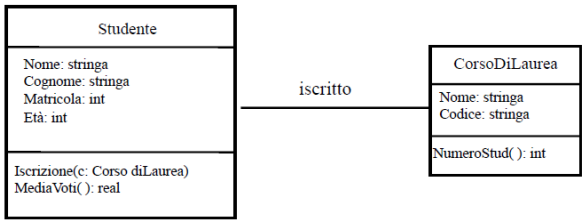
• alfa (X1: T1, … , Xn: Tn): T è la segnatura dell’operazione (T può mancare),

• pre rappresenta la precondizione dell’operazione, cioè l’insieme delle condizioni (a parte quelle già stabilite dalla segnatura) che devono valere prima di ogni esecuzione della operazione

• post rappresenta le postcondizioni della operazione, cioè l’insieme delle condizioni che devono valere alla fine di ogni esecuzione della operazione

Esempio di specifica di una operazione

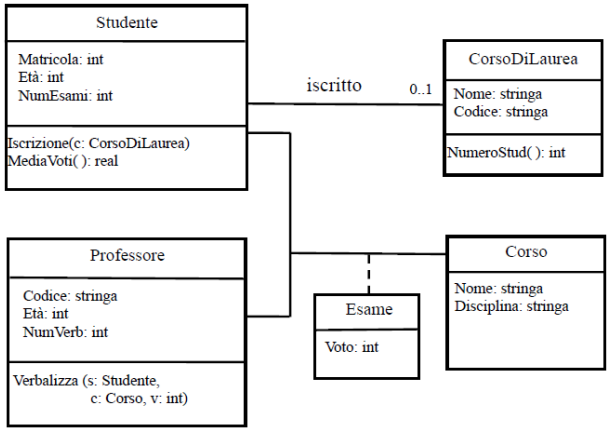
InizioSpecificaClasse CorsoDiLaurea
NumeroStud() : int
pre : nessuna
post : result è uguale al numero di studenti iscritti nel corso di laurea this
FineSpecifica



Precondizioni e postcondizioni

• Nella specifica di una operazione, nella precondizione si usa "this" per riferirsi all'oggetto di invocazione della operazione
• Nella specifica di una operazione, nella postcondizione si usa
– "this" per riferirsi all'oggetto di invocazione della operazione nello stato corrispondente alla fine della esecuzione della operazione
– "result" per riferirsi al risultato restituito dalla esecuzione della operazione
– pre(alfa) per riferirsi al valore della espressione alfa nello stato corrispondente alla precondizione

Esempio di diagramma delle classi



Specifica di attvità

Distinguiamo la specifica di
• attvità atomiche
• attvità complesse
Per una attività atomica abbiamo necessità di specificare l'operazione che svolge in termini di segnatura, precondizione e postcondizioni.
Per le attività complesse invece una una specifica in termini di precondizioni e postcondizioni non è sufficiente.

Specifica di attività atomiche

Per quanto riguarda le attività atomiche la specifica è del tutto analoga alla specifica delle classi: Una attività atomica è costituita da una singola operazione.
InizioSpecificaAttivitàAtomica A
……
FineSpecifica

Specifica di una attività atomica

La specifica della operazione di un'attività atomica ha la seguente forma (analoga alla specifica di una operazione di classe):
• alfa (X1: T1, … , Xn: Tn): (Y1: T1, …, Yn: Tn) è la segnatura dell'operazione:
- alfa è il nome dell'operazione (tipicamente quello dell'attività atomica)
- X1: T1, … , Xn: Tn sono i parametri dell'operazione
- Y1: T1, …, Yn: Tn sono i risultati dell'operazione (questi possono essere 0, 1, o più di uno, se il risultato è uno solo allora si può omettere il nome denotandolo convenzionalmente con ``result'').
• **pre** rappresenta la precondizione dell'operazione, cioè l'insieme delle condizioni (a parte quelle già stabilite dalla segnatura) che devono valere prima di ogni esecuzione della operazione
• **post** rappresenta le postcondizioni della operazione, cioè l'insieme delle condizioni che devono valere alla fine di ogni esecuzione dell'operazione.

alfa (X1: T1, … , Xn: Tn): (Y1:T1, …, Yn:Tn)

pre: *condizione*

post: *condizione*

Precondizioni e postcondizioni

• Nella specifica delle precondizioni e postcondizioni di una attività atomica non si può usare “this” non essendoci l’oggetto di invocazione.

• Si usano invece (analogamente al caso delle operazioni di classe):

– “Y1, …, Yn” per riferirsi ai risultati restituiti dall’esecuzione dell’operazione

– pre(alfa) per riferirsi al valore della espressione alfa nello stato corrispondente alla precondizione

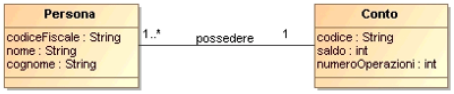
Specifica di attività complesse

Per quanto riguarda le attività complesse la specifica ha una natura profondamente diversa.

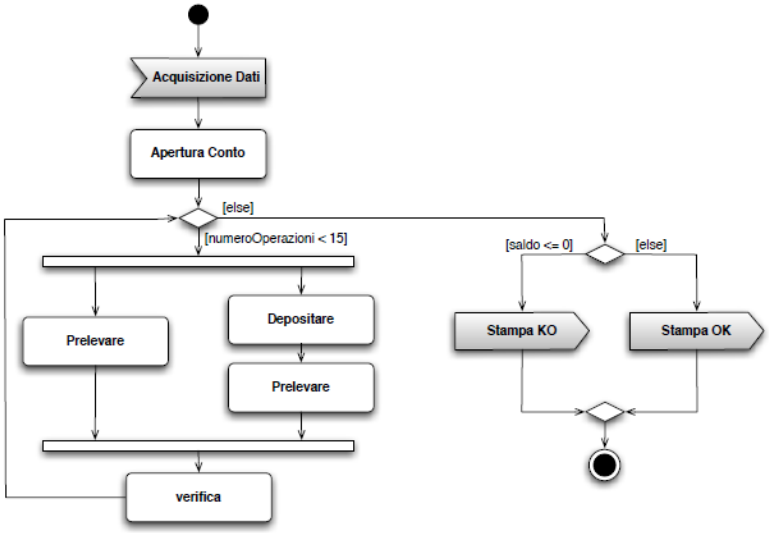
Infatti, in tali attività il flusso stesso di controllo è di interesse.

Naturalmente il processo descritto da tali attività insiste sulla estensione del diagramma delle classi ed infatti le attività atomiche accedono/modificano tale diagramma.

Esempio: diagramma delle classi



Esempio: diagramma delle attività



Analisi del diagramma delle attività

Un diagramma delle attività si compone di

– Attività atomiche che lavorano sulle classi, che chiamiamo task

– Segnali di I/O (interfacciamento della applicazione con i clienti)

– Strutture di controllo

Attività atomiche & I/0

Task:	I/O:
• Apertura Conto	• Acquisizione Dati
• Prelevare	• StampaOK
• Depositare	• StampaKO
• Verifica	

Specifica di attività complesse

Oltre a definire le attività atomiche, l’attività complessa si deve occupare di legare gli input e output delle varie operazioni delle stesse tra loro, oltre che del flusso di controllo già specificato nel diagramma delle attività.

Per fare ciò, la specifica delle attività complesse è costituita da:

– Segnatura

– Variabili di processo

– Specifica del flusso di controllo che tenga conto dei dei dati

Segnatura è formata da:
– Nome dell'attività
– Parametri di ingresso
– Parametri di uscita
La segnatura è del tutto analoga alla segnatura delle attività atomiche e che quindi descrive i parametri di input e di output della attività stessa.
• Si noti che l'attività oltre ad accedere ai sui parametri ha sempre accesso a tutto il diagramma delle classi
– In lettura: per leggere l'istanziazione del diagramma delle classi
– In scrittura: per fare side-effect sull'istanziazione del diagramma delle classi!
• I parametri di ingresso servono tipicamente per passare gli id degli oggetti che costituiscono i punti di ingresso all'istanziazione del diagramma delle classi.
• I parametri di uscita servono a restituire eventuali risultati (oltre quelli memorizzati con side-effect sul diagramma delle classi), per esempio l'id di nuovi oggetti creati o selezionati.

Variabili di processo:
- Sono variabili ausiliarie private del processo.
- Si aggiungono ai parametri di ingresso e uscita.
- Formano una sorta di memoria di lavoro locale della attività.
- Possono essere scritte e lette più volte.
- Servono a memorizzare risultati intermedi per passarli come parametri a sottoattività successive.
- Le sottoattività NON hanno accesso a queste variabili (che sono private) se non sono passate come parametro. Questo favorisce l'interfacciamento esplicito:
- I dati in input e output ad una attività /sottoattivita sono solo i suoi parametri
+ l'istanziazione del diagramma delle classi

Flusso di controllo
• Il flusso di controllo del processo è già stabilito dal diagramma delle attività.
• Tuttavia, tale descrizione va dettagliata per tenere conto dei dati.
• Serve una specifica del processo messo in atto dall'attività, secondo il diagramma delle attività ma con in più l'esatta descrizione di quali dati sono scambiati tra le sottoattività. Noi descriveremo quest'ultimo in pseudocodice Java.
• Adottamo una specifica procedurale in pseudo codice (simile a Java ma senza alcuni dettagli specifici)

Specifica di una attività complessa
InizioSpecificaAttivita NomeAttivita
Segnatura Attività Complessa
con nome parametri di input e parametri di output
VariabiliProcesso
Definizione delle variabili di processo con nome e tipo
InizioProcesso
Descrizione procedurale del processo stesso in pseudocodice
FineProcesso
FineSpecifica

Esempio

```
InizioSpecificaAttivita Principale

Principale()

VariabiliProcesso

    marito: Persona
    moglie: Persona
    conto: Conto
    numeroOperazioni : Int
    saldo: Real


InizioProcesso

   AquisizioneDati():(marito,moglie);
   AperturaConto(marito,moglie): (conto, numeroOperazioni, saldo);
   while (numeroOperazioni < 15) {
     fork {
       thread t1: {
          Prelevare(marito,conto);
       }
       thread t2: {
          Deposita(moglie,conto);
          Prelevare(moglie,conto);
       }
     }
     join t1, t2;
   }
   verifica(conto): (numeroOperazioni, saldo)

   if (saldo <= 0)
      StampaKO();
   else
      StampaOK();

FineProcesso

FineSpecifica
```

Pseudo codice per rappresentare processi

Come pseudocodice usiamo un formalismo simile a Java.

• Per rappresentare cicli e decisioni usiamo le strutture di controllo Java (if-else, while, do-while).

Esempio:

```
while (numeroOperazioni < 15) { … }
if (saldo <= 0) StampaKO(); else StampaOK();
```

• Per invocare sottoattività (atomiche e non) usiamo la sua segnatura come fosse un metodo statico, passandogli in input e in output i parametri attuali

Esempio:

```
Prelevare(marito,conto);
```

• Per quanto riguarda il passaggio dei parametri in input, si assume il solito passaggio di parametri per valore di Java: la variabile parametro attuale viene valutata e il riferimento corrispondente diviene il valore iniziale del parametro attuale di input dell'attività.

• Per invocare sottoattività (atomiche e non) usiamo la sua segnatura come fosse un metodo statico, passandogli in input e in output i parametri attuali

Per quanto riguarda il passaggio dei parametri in output delle sottoattività (atomiche e non) , si assume un passaggio di parametri per risultato: la variabile parametro attuale viene assegnata con il nuovo valore al termine dell'operazione, durante l'operazione stessa tale variabile non può essere ne letta ne modificata.

Esempio:

```
AquisizioneDati():(moglie,marito);   //NB la segnatura è AquisizioneDati():(pe1: Persona, pe2: Persona);
```

• In pratica per quanto riguarda il passaggio dei parametri di output possiamo pensare l'istruzione:
AquisizioneDati():(moglie,marito); come una **abbreviazione** per le seguenti istruzioni:

```
AquisizioneDati();
marito = AcquisizioneDati.pe1;
moglie = AcquisizioneDati.pe2;
```

dove, attraverso la notazione basata sul punto, usiamo i parametri di uscita come fossero campi dati di una classe.

• Per invocare sottoattività (atomiche e non) usiamo la sua segnatura come fosse un metodo statico.

Esempio:

```
Prelevare(marito,conto); AquisizioneDati();  //NB la segnatura è AquisizioneDati():(pe1: Persona, pe2: Persona, co: Conto);
```

• Per riferirci ai risultati delle sottoattività (atomiche e non) useremo la notazione basata sul punto, dove usiamo i parametri di uscita come fossero campi dati di una classe.

Esempio:

```
marito = AcquisizioneDati.pe1;
```

• Per introdurre ed eliminare thread di controllo concorrenti facciamo uso di nuovi costrutti speciali fork e join

• A ciascun thread generato associamo un identificatore che può poi essere usato nei join

Esempio fork:

```
fork {
  thread t1: {
  …
  }
  thread t2: {
  …
  }
}
```

Esempio join:

```
join t1, t2;
```

• E' naturalmente possibile fare uso di sottoattività complesse nel definire un processo di una attività complessa, ed è anche possibile fare uso della ricorsione.

• Inoltre se necessario, si può usare liberamente l'assegnazione sulle le variabili di processo

LEZIONE 3 – SLIDE 4

Classi come funzioni

Nei linguaggi orientati agli oggetti, ed in particolare in JAVA è spesso necessario realizzare classi che rappresentano funzioni. Chiamiamo nel seguito **funtore** una classe che rappresenta una funzione, cio`e le cui istanze rappresentano attivazioni (computazioni) di una funzione.

Il loro utilizzo in JAVA è comune. Per esempio gli oggetti che implementano l'interfaccia Runnable sono oggetti funtore che rappresentano (oltre eventualmente ad altro) la computazione descritta in run(). Essi vanno eseguiti costruendo un oggetto Thread che funge appunto da esecutore e chiamando su tale oggetto il metodo start() che a sua volta chiama il metodo run() dell'oggetto Runnable associato al Thread.

Esempio Runnable

```
package esempioRunnerConcorrenti;
public class Runner implements Runnable {
   private String nome;
```

```
   public Runner(String n) { nome = n; }
   public void run() {
      for (int i = 0; i < 25; i++) {
         System.out.println(nome + " sta girando");
      }
   }
}

package esempioRunnerConcorrenti;
public class Main {
    public static void main(String[] args) {   // Crea attivazioni del Runner
        Runner r1 = new Runner("Alpha");
        Runner r2 = new Runner("Beta");
       // Attivazioni passata all'esecutore (cioe' Thread)
       Thread alpha = new Thread(r1);
       Thread beta = new Thread(r2);
       // Esecuzione dell'attivazione (start() chiama run()
       alpha.start();
       beta.start();
    }
}
```

Il pattern Funtore

Approfondiamo la nozione di funtore, cioè una classe le cui istanze rappresentano attivazioni di funzioni.
Lo facciamo trasformando una funzione statica in un funtore.

Consideriamo un esempio di funzione:

```
package funtore;
public class Funzione {
    public static double function(Object o, int i) {   // fa cose con o, i
    System.out.println("faccio cose con " + o + " e " + i);
    // restituendo ris
    return i;
   }
}
```

La corrispondente classe funtore avrà la seguente forma:

```
package funtore;
public class Funtore implements Runnable {
    private boolean eseguita = false;
    private Object o;
    private int i;
    private double ris;
    public Funtore(Object oo, int ii) {
        o = oo;
        i = ii;
   }

    public synchronized void run() {
        if (eseguita)
            return;
        eseguita = true;
        // fa cose con o, i
        System.out.println("faccio cose con " + o + " e " + i);   // restituendo ris ris = i;
   }

   public synchronized double getRis() {
       if (!eseguita)
           throw new RuntimeException();
       return ris;
}

public synchronized boolean estEseguita() { return eseguita; }
}
```

Discutiamone i vari aspetti.
• Il nome **della funzione** è ora diventato il nome **della classe**.
• I parametri della funzione sono ora passati in ingresso al costruttore della classe e vengono memorizzati in opportuni campi dati privati della classe stessa.
• Il risultato viene anche esso memorizzato in un campo privato della classe e reso disponibile attraverso getRis().
• Il corpo della funzione è adesso il corpo del metodo run. In questo caso genererà effettivamente la computazione voluta.

Vogliamo che ogni oggetto della classe Funtore corrisponda esattamente ad una attivazione della funzione funzione().
Per fare ciò procediamo come segue:
• Introduciamo una variabile booleana eseguita che indica se l'attivazione corrente del funtore è stata eseguita o meno e che inizialmente è posta a false.
• Il metodo responsabile dell'esecuzione (run() in questo caso) verifica se la variabile eseguita è true,
– se lo è esce senza fare altro, poichè la computazione è già stata eseguita (o potremmo dire che l'attivazione corrente della funzione è stata "consumata") e non deve essere eseguita una seconda volta.
N.B. Si noti però che ovviamente è possibile generare una nuova attivazione del funtore costruendo un nuovo oggetto.
– altrimenti pone la variabile eseguita a true ed esegue il metodo.
• Il metodo getRis(), che restituisce il risultato al cliente, può essere chiamato solo se eseguita è true, cioè se il metodo responsabile dell'esecuzione (run()) è stato effettivamente eseguito.
• Infine diamo al cliente la possibilità di leggere la variabile eseguita, attraverso il metodo estEseguita(), per evitare un uso scorretto del funtore.
C'è un ultimo aspetto da commentare: se assumiamo, come in questo caso, che il funtore venga usato in un programma multithread, allora dobbiamo renderlo "thread safe", cioè dobbiamo evitare che esecuzioni multiple dei metodi di un dato oggetto corrompano la correttezza delle informazioni riportate nei suoi campi dati. Per fare ciò rendiamo tutti i metodi pubblici del funtore synchronized.
In questo modo, per ogni oggetto della classe Funtore, sarà in ogni momento in esecuzione un solo metodo tra quelli pubblici messi a disposizione dall'oggetto stesso (cioè tra run(), getRis(), estEseguita()), rispecchiando pienamente il comportamento di una attivazione di funzione.
Concludiamo questa introduzione al pattern funtore mostrando come un cliente deve invocare una funzione rappresentata da un funtore.
In particolare, confrontiamo la chiamata ad una funzione con la chiamata al corrispondente funtore, facendo uso del nostro esempio:

```
package funtore;
public class MainFunzione {
    public static void main(String[] args) {
    // usando FUNZIONI
    Object  o  =  new Object();
    double ris = Funzione.function(o, 10);
    System.out.println(ris);
   }
}
```

```
package funtore;
public class MainFuntore {
    public static void main(String[] args) {
    // usando FUNTORI
    Object o = new  Object();
    Funtore f = new Funtore(o, 10);
    Thread t = new Thread(f);
    t.start();
    try {
        t.join();
    }  catch (InterruptedException  e) { e.printStackTrace(); }
    System.out.println(f.getRis());
    }
}
```

Discutiamo i vari aspetti della chiamata ad un funtore
• Prima di tutto deve essere costruito un oggetto Funtore corrispondente alla attivazione della funzione Funzione.
• Questo oggetto va passato all'esecutore (Thread in nostro caso).
• L'esecutore richiama il metodo del funtore che effettivamente computa la funzione (run() chiamato da start(), nel nostro caso).
• Infine una volta terminata l'esecuzione di questo metodo, il risultato è reso disponibile attraverso getRis().
L'uso del join del thread generato con il thread del main() in questo esempio è solo contingente*, l'importante è dare la possibilità al metodo run() di iniziare (e quindi completare visto che tutti i metodi del funtore sono synchronized) prima di richiedere il risultato.*

Si noti che complessivamente il cliente risulta più complesso che nel caso della chiamata ad una funzione, ma anche più flessibile. Per esempio, è possibile far restituire più di in risultato ad un funtore, cosa impossibile nel caso di funzioni.

Il pattern Singleton

Prima di poter realizzare il diagramma delle attività abbiamo bisogno di un ulteriore pattern realizzativo: il **pattern Singleton**.
Lo scopo del pattern singleton è realizzare una classe con una sola istanza. lo scopo è avere un singolo oggetto con proprietà speciali di interesse.
Per fare ciò dobbiamo impedire l'uso dell'operatore new, rendendo il costruttore privato e usando invece un metodo statico per farci restituire il singolo oggetto, ogni volta che serve.
Questa è la realizzazione tipica dove l'oggetto singolo viene costruito solo se richiesto lazy construction.

```
package singleton;
public class MySingleton {
    private static MySingleton uniqueInstance;    //"lazy construction"
    // altre variabili di istanza
    private MySingleton() { //inizializza a campi significativi }
    //synchronized se per deve essere thread-safe
    public static synchronized MySingleton getInstance() {
        if (uniqueIntance == null)
                uniqueIntance = new MySingleton();
        return uniqueInstance;
        }
        // altri metodi
}
```

Questa è la realizzazione tipica dove l'oggetto singolo viene construito in ogni caso eager construction.

```
package singleton;
public class MySingleton {
    private static MySingleton uniqueInstance = new MySingleton(); //"eager construction"
    // altre variabili di istanza
    private MySingleton() { //inizializza a campi significativi }
    //synchronized se per deve essere thread-safe
    public static synchronized MySingleton getInstance() { return uniqueInstance; }
    // altri metodi
}
```

Realizzazione dei diagrammi delle attività UML

Un diagramma delle attività descrive una attività complessa, che potremmo in prima approssimazione considerare come una funzione, o meglio una operazione o un processo, in cui il flusso di controllo stesso che costituisce il corpo della funzione è di primario interesse.
Si noti che usiamo il termine funzione per semplicità, in realtà tipicamente delle funzioni interessa solo il risultato prodotto, mentre per le attività complesse interessa il flusso delle sotto attività coinvolte.
Infatti, è assolutamente possibile avere attività complesse che non producono alcun risultato, o che non terminano affatto (per esempio, il sistema operativo di un calcolatore è una siffatta attività).

Tipi di attività

In un diagramma delle attività associato ad una attività complessa (inclusa l'attività principale) troviamo varie sotto attività.
I tipi di sotto attività sono fondamentalmente i seguenti:
• Attività atomiche atte ad interagire con la realizzazione del diagramma delle classi UML, chiamiamo tali attività **task**;
• **Segnali di ingresso/uscita**, cioè segnali di ingresso o di uscita, o che comunque si rivolgono all'esterno del sistema realizzato.
• **Segnali per la gestione degli eventi**, cioè segnali atti a gestire gli eventi degli oggetti dotati di diagramma di transizione.
• Altre sotto attività complesse.
Tali attività sono organizzate secondo un flusso di controllo anche ramificato in diversi flussi concorrenti.

Attività atomiche: task

I task sono le attività atomiche fondamentali. Esse servono ad accedere e a modificare le classi JAVA che corrispondono alle classi e alle associazioni del diagramma delle classi.
Tali classi saranno in generale accedute concorrentemente da diverse attività. Quindi dobbiamo garantirne l'integrità a fronte di accessi concorrenti.
Per regolare questo accesso concorrente, non possiamo semplicemente aggiungere synchronized a tutti i metodi delle classi realizzate. Infatti, è immediato accorgersi che questo porterebbe a deadlock.
Risolviamo il problema dell'accesso concorrente facendo uso del pattern funtore e richiedendo che l'esecutore di questi funtori si prenda carico della sincronizzazione.

• Richiediamo che le attività atomiche che accedono al diagramma delle classi (o meglio alla sua realizzazione in JAVA) implementano una interfaccia specifica Task:

```
package _framework;
public interface Task {
    public void esegui(); //va utilizzata SOLO da TaskExecutor
}
```

• I task sono eseguiti da un esecutore specifico che appartiene alla classe Singleton TaskExecutor:

```
package _framework;
//Implementa il pattern Singleton
//Nota e' importantissimo che perform sia synchronized in TaskExecutor!
//Serve per avere un accesso controllato agli oggetti del diagramma delle classi
public final class TaskExecutor {
    private static TaskExecutor theTaskExecutor = new TaskExecutor();
    private TaskExecutor(){}
    public synchronized static TaskExecutor getInstance() { return theTaskExecutor; }
    public synchronized void perform(Task t) { t.esegui(); }
}
```

Commentiamo il framework presentato.
• Le attività atomiche che accedono al diagramma delle classi implementano una interfaccia specifica Task che include un solo metodo esegui().
• Tale metodo è "pseudo privato" (anche se per semplicità non lo stiamo forzando come abbiamo fatto per lo schema realizzativo per associazioni con responsabilità doppia) perchè deve essere attivato solo attraverso il metodo perform() di TaskExecutor.
• TaskExecutor è una classe che implementa il pattern Singleton.
• L'unica istanza di TaskExecutor è il solo esecutore effettivo dei task che esegue attraverso il suo metodo perform().
• Il metodo perform() è synchronized il che implica che se una attivazione di perform() è in esecuzione in un thread, tutte le altre eventuali attivazioni perform() su altri thread rimangono in attesa.
In altre parole: Un solo Task alla volta può essere in esecuzione e quindi modificare il diagramma delle classi.
In questo modo garantiamo di non poter corrompere il diagramma delle classi a causa di accessi concorrenti, cioè rendiamo il diagramma delle classi thread safe.

Esempio di attività atomica

```
package attivita_atomiche;
import numeriprogressivi.GeneratoreNumeriProgressivi;
import _framework.*;
import conto.*;
import persona.*;
import possedere.*;

public class AperturaConto implements Task {
    private boolean eseguita = false;
    private Persona personaInput_1;
    private Persona personaInput_2;
    private Conto contoOutput;
    private int numOperazioni;
    private int saldo;
    public AperturaConto(Persona p1, Persona p2) {
        personaInput_1 = p1;
        personaInput_2  =  p2;
        contoOutput = new Conto(getID(),0);
    }
    private String getID() { return GeneratoreNumeriProgressivi.getInstance().nuovoNumeroProgressivo(); }
    public synchronized void esegui() {
        if (eseguita == true)   return;
        eseguita = true;
        personaInput_1.inserisciLinkPossedere(new TipoLinkPossedere(personaInput_1,contoOutput));
        personaInput_2.inserisciLinkPossedere(new TipoLinkPossedere(personaInput_2,contoOutput));
    }
    public synchronized Conto getConto() {
        if (!eseguita) throw new RuntimeException("Attivita' non ancora eseguita");
```

```
    return contoOutput;
    }
    public synchronized int getNumeroOperazioni() {
        if (!eseguita) throw new RuntimeException("Attivita' non ancora eseguita");
        return contoOutput.getNumeroOperazioni();
    }
    public synchronized int getSaldo() {
        if (!eseguita) throw new RuntimeException("Attivita' non ancora eseguita");
        return contoOutput.getSaldo();
    }
    public  boolean estEseguita() { return eseguita; }
}
```

Commentiamo l'esempio:

- Segue il pattern funtore implementando l'interfaccia Task.
- Prende tre parametri di ingresso e non restituisce nulla, ma fa side-effect sul diagramma delle classi.
- Il corpo del funtore è il corpo di esegui() che appunto effettua l'accesso al diagramma delle classi
- Va utilizzata attraverso la singola istanza di TaskExecutor:

```
TaskExecutor executor = TaskExecutor.getInstance();
...
AperturaConto ac = new AperturaConto(marito, moglie);
executor.perform(ac);
```

Segnali di ingresso/uscita

Per i segnali di ingresso/uscita presenti in una attività complessa, non diamo alcuna strategia implementativa specifica. Semplicemente le realizzeremo come funzioni o funtori con opportuni esecutori.

- Ovviamente rimangono validi i principi base della modularità: basso accoppiamento tra i moduli, alta coesione all'interno di ciascun modulo, alta information hiding, e alto interfacciamento esplicito tra moduli.
- Questo suggerisce di racchiudere in uno o più moduli separati i segnali di ingresso/uscita rendendo così (la realizzazione in JAVA de) il diagramma delle attività indipendente dall'interfaccia utente scelta per l'ingresso/uscita.
- Per esempio si può raccogliere in un'unica classe un insieme di metodi statici che servono per i vari segnali di ingresso/uscita, di modo che (la realizzazione in JAVA de) il diagramma delle attività faccia uso di chiamate a questi metodi per l'ingresso/uscita, e il loro corpo si occupi dell'interfacciamento con l'utente.

Esempio

```
package segnali_io;
import persona.Persona;
public class SegnaliIO {
    public static RecordPersonaPersona aquisizioneDati() {
        RecordPersonaPersona result = new RecordPersonaPersona();
        result.persona1 = new Persona("GIONIONH17H501F", "Pippo", "De Pippis");
        result.persona2 = new Persona("EINOINMS60G230T", "Clarabella", "De Claris");
        return result;
    }
    public static void aperturaOK() { System.out.println("Apertura del conto avvenuta con successo"); }
    public static void stampaOK() { System.out.println("Il conto per fortuna non è in rosso :-)"); }
    public static void stampaKO() { System.out.println("ALLARME: non ci sono piu' soldi nel conto"); }
    //Metodo aggiunto per mostrare il funzianamento: da rimuovere prima del rilascio
    public static void mandaMessaggio(String messaggio) { System.out.println(messaggio); }
}
```

Segnali per la gestione eventi

Questi sono segnali che per l'inizializzazione del meccanismo per la gestione gestione degli eventi, l'attivazione e disattivazione di tale meccanismo, e l'introduzione di eventi da inviare ai vari oggetti dotati di (una realizzazione in JAVA del) diagramma degli stati e delle transizioni. Tutte queste operazioni hanno a che fare con il meccanismo di gestione degli eventi che è chiaramente completamente esterno al diagramma delle attività.

Come per i segnali di ingresso/uscita, anche per i segnali per la gestione eventi non diamo alcuna strategia realizzativa specifica. Semplicemente le realizzeremo come funzioni o funtori con opportuni esecutori.

Assumeremo però che gli oggetti dotati di (una realizzazione in JAVA del) diagramma degli stati e delle transizioni vengano eseguiti su thread separati dai thread delle attività e di avere un meccanismo di gestione degli eventi per lo scambio dei messaggi che sia adatto alla concorrenza. Naturalmente assumeremo che le azioni associate alle transizioni dei diagrammi degli stati e delle transizioni avvengano attraverso il meccanismo dei task introdotto sopra per garantire l'integrità dell'informazione (degli oggetti JAVA che realizzano l'istanziazione corrente) del diagramma delle classi.

Attività complesse

Realizziamo le attività complesse (sia l'attività principale che le sotto attività complesse) come dei funtori il cui esecutore è la classe Thread di JAVA.
Più precisamente:
• il funtore stesso implementa Runnable;
• il codice da eseguire corrispondente al flusso di controllo dell'attività complessa è nel corpo del metodo run();
• l'esecutore del funtore è la classe Thread predefinita in JAVA che a sua volta chiama il codice da eseguire (run()) attraverso start().
Si noti che in questo modo tutte le attività complesse verranno eseguite su thread separati. Useremo join() (metodo definito in Thread) per correlarne l'esecuzione con l'esecuzione dell'attività che le ha invocate. In altre parole attiveremo attività complesse attraverso un fork (start()) dopo di che aspetteremo l'esecuzione della stessa attraverso un join (join()). Naturalmente in questo modo possiamo attivare più sotto attività da eseguire concorrentemente se necessario.
Nota: per semplicità assumeremo che ogni volta che il diagramma delle attività richieda un fork su più attività concorrenti, queste siano sempre complesse (anche se poi all'interno contengono solo una attività atomica).

Costrutti sequenziali per il controllo di flusso

Per quanto riguarda i costrutti sequenziali del diagramma delle attività (cioè le sequenze, le guardie, i nodi condizionale e i nodi merge), faremo uso direttamente delle strutture di controllo sequenziali messe a disposizione da JAVA. Useremo quindi:
• sequenza di istruzioni
• if-else, while, do-while.

Costrutti concorrenti per il controllo di flusso

Per quanto riguarda i costrutti concorrenti del diagramma delle attività, useremo:
• Thread.start() per il fork
• Thread.join() per il join.
Per semplicità, e senza perdita di generalità, richiederemo che ciascun ramo di un fork sia incapsulato in una sotto attività complessa.

Esempio

```
package attivita_complesse;
import attivita_atomiche.*;
import persona.*;
import segnali_io.*;
import _framework.*;
import conto.*;
public class AttivitaPrincipale implements Runnable {
    private boolean eseguita = false;
    private TaskExecutor executor = TaskExecutor.getInstance();
    public synchronized void run() {
        if (eseguita == true) return;
        eseguita = true;
        Persona marito;
        Persona moglie;
        Conto conto;
        int numeroOperazioni;
        int saldo;
        //operazione IO
        RecordPersonaPersona ris = SegnaliIO.aquisizioneDati();
        marito = ris.persona1;
        moglie = ris.persona2;
        //task
        AperturaConto ac = new AperturaConto(marito, moglie);
        executor.perform(ac);
        conto = ac.getConto();
        numeroOperazioni = ac.getNumeroOperazioni();
        saldo = ac.getSaldo();
        //operazione IO
        SegnaliIO.aperturaOK(); //Per mostrare cosa succede: da rimuovere
        while (numeroOperazioni < 15) {
            //attività complessa
            Thread ramo1 = new Thread(new AttivitaSottoramo1(moglie, conto));
            ramo1.start(); //FORK: primo ramo
            //attività complessa
            Thread ramo2 = new Thread(new AttivitaSottoramo2(marito, conto));
            ramo2.start(); //FORK: secondo ramo
```

```
            try {
                ramo1.join(); //JOIN: primo ramo
                ramo2.join(); //JOIN: secondo ramo
            } catch (InterruptedException e) { e.printStackTrace(); }
            Verifica v = new Verifica(conto);
            executor.perform(v);
            numeroOperazioni = v.getNumOperazioni();
            saldo = v.getSaldo();
        }
        if (saldo > 0)
        //operazione IO
        SegnaliIO.stampaOK();
        else
        //operazione IO
        SegnaliIO.stampaKO();
        }
        public synchronized boolean estEseguita() { return eseguita; }
}
```

Commentiamo brevemente l'esempio:

• Innanzitutto, si nota immediatamente che AttivitaPrincipale realizza come richiesto il pattern funtore, avendo come esecutore la classe Thread.

• Il corpo del funtore costituito dal corpo di run contenente l'intero processo associato all'attività complessa.

• Vengono definite tre variabili per l'attività: moglie, marito, conto.

• Il flusso di controllo è inizialmente sequenziale:

- Legge i dati dal conto (si noti che SegnaliIO.aquisizioneDati() restituisce un record ris formato da tre campi dati pubblici: marito, moglie, conto. Questa attività di ingresso/uscita prenderà in qualche modo i dati necessari dall'utente dell'applicazione.
- Apre il conto stesso immettendo i dati raccolti nel diagramma delle classi attraverso una opportuna attività atomica, il task AperturaConto.
- Manda in output l'ok sulla apertura conto (segnale di ingresso/uscita).

• A questo punto troviamo un while. Si noti che il test stesso del while è basato su un task (qui incapsulato in un metodo privato numeroOperazioni()) poiché richiede l'accesso al diagramma delle classi.

• Nel corpo del while abbiamo un fork in due sottoattività complesse concorrenti, AttivitaSottoramo1 e AttivitaSottoramo2 e poi il join delle stesse.

• All'uscita del while troviamo un task per recuperare l'ammontare del conto e un if-else utilizzato per selezionare il giusto segnale di uscita da effettuare. (codice delle altre attività complesse sulle slide complete)

LEZIONE 4 – SLIDE 1

Il diagramma degli stati e delle transizioni

Il diagramma degli stati e delle transizioni viene definito per una classe, ed intende descrivere l'evoluzione di un generico oggetto di quella classe.

Il diagramma rappresenta le sequenze di stati, le risposte e le azioni, che un oggetto attraversa durante la sua vita in risposta agli stimoli ricevuti.

Uno stato rappresenta una situazione in cui un oggetto ha un insieme di proprietà considerate stabili

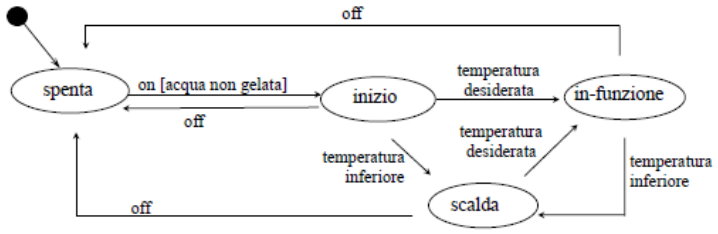
Una transizione modella un cambiamento di stato ed è denotata da: **Evento [Condizione] / Azione**

Il significato di una transizione del tipo di quella qui mostrata è:

Se l'oggetto si trova nello stato ***S₁*** e riceve l'evento ***E*** e la condizione ***C*** è verificata, allora attiva l'esecuzione dell'azione ***A*** e passa nello stato ***S₂***.

Esempio di diagramma degli stati e delle transizioni per la classe Caldaia

Descriviamo il diagramma degli stati e delle transizioni relativa ad una classe "Caldaia". In questo diagramma ogni transizione è caratterizzata solamente da eventi e condizioni (i cambiamenti di stato non hanno bisogno di azioni perché sono automatici)

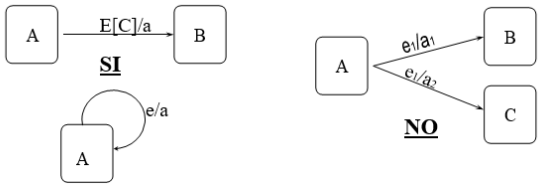


Stato

• Lo stato di un oggetto racchiude le proprietà (di solito statiche) dell'oggetto, più i valori correnti (di solito dinamici) di tali proprietà
• Una freccia non etichettata che parte dal "vuoto" ed entra in uno stato indica che lo stato è iniziale
• Una freccia non etichettata che esce da uno stato e finisce nel "vuoto" indica che lo stato è finale
• Stato iniziale e finale possono anche essere denotati da appositi simboli

Transizione

• Ogni transizione connette due stati
• Il diagramma corrisponde ad un automa deterministico (transizioni dallo stesso stato hanno eventi diversi), in cui un evento è un input, mentre un'azione è un output
• La condizione è detta anche **"guardia" (guard)**
• L'evento è (quasi) sempre presente (condizione e azione sono opzionali)



Esempio di diagramma degli stati e delle transizioni per la classe Motore

L'analisi dei requisiti ha evidenziato l'esistenza, nel diagramma delle classi, di una classe "Motore". Tracciare il diagramma degli stati e delle transizioni a partire da questi requisiti.
Un motore di automobile può essere spento o acceso, ma può essere avviato o spento solo se la marcia è in folle

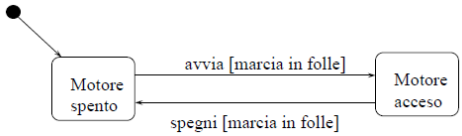
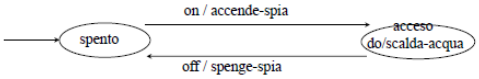


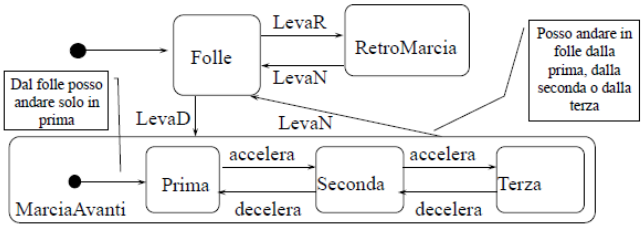
Diagramma degli stati e delle transizioni

Alcune volte vogliamo rappresentare dei processi che l'oggetto esegue senza cambiare stato. Questi processi si chiamano attività, e si mostrano negli stati con la notazione: **do / attività**
Esempio (scaldabagno):



Stato composto

• Uno **stato composto (o macro-stato)** è uno stato che ha un nome, e che contiene a sua volta un diagramma
• Esiste uno stato iniziale del macro-stato
• I **sottostati ereditano** le transizioni in uscita del macro-stato



Aspetti metodologici nella costruzione del diagramma degli stati e delle transizioni

Un metodo comunemente usato per costruire il diagramma degli stati e delle transizioni prevede i seguenti passi
• Individua gli stati di interesse
• Individua le transizioni
• Individua le attività
• Determina gli stati iniziali
• **Controllo di qualità**
• **Correggi, modifica, estendi**

Controllo di qualità del diagramma degli stati e delle transizioni

Sono stati colti tutti gli aspetti insiti nei requisiti? Ci sono ridondanze nel diagramma? Ogni stato può essere caratterizzato da proprietà dell'oggetto? Ogni azione e ogni attività possono corrispondere ad una operazione della classe? Ogni evento e ogni condizione può corrispondere ad un evento o condizione verificabile per l'oggetto? (Ulteriori esercizi sulle slide complete)

Diagramma degli stati e delle transizioni di oggetti reattivi

Principi generali

• Assumiamo di avere diversi **oggetti reattivi**, cioè con associato un diagramma stati-transizioni.
• Assumiamo che l'interazione sia basata su scambio esplicito di **eventi**
• Assumiamo che gli eventi abbiano un mittente ed un destinatario
In particolare ammettiamo che

- **Messaggi punto-punto**: un oggetto manda un messaggio ad un altro oggetto
- **Messaggi in broadcasting**: un oggetto manda un messaggio a tutti gli altri oggetti.

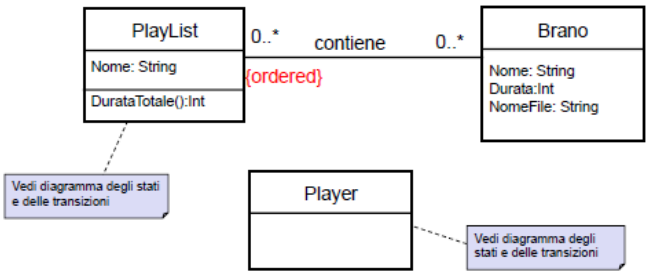
• Inoltre gli eventi possono avere **parametri** con specifico **contenuto informativo** (il cosiddetto **payload** del messaggio)
• Una **azione** può a sua volta lanciare un **evento** (tipicamente uno solo per semplicità) per un altro oggetto o in broadcasting.

Osservazioni

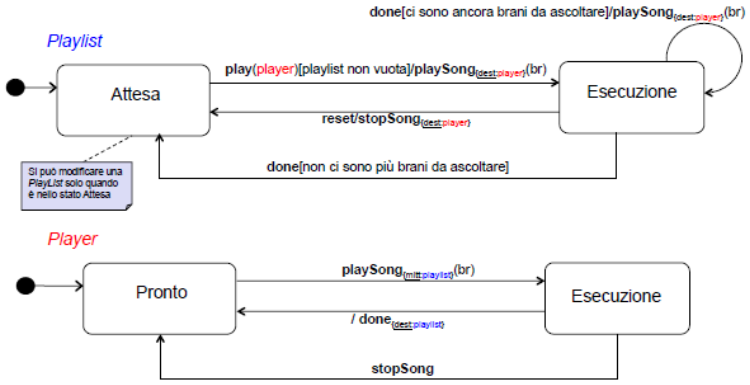
• Nel diagramma degli stati e transizioni per semplicità identifichiamo **l'azione** stessa con **l'evento lanciato.**
• Diamo una specifica dettagliata di ciò che avviene ad ogni transizione:
– Quali **eventi** sono **recepiti e lanciati** (e a chi)
– Come cambiano eventuali **variabili di stato ausiliarie** associate allo stato dell'oggetto
– Come **cambia l'istanziazione del diagramma delle classi**
• Le **variabili di stato ausiliarie** non sono di interesse per il cliente servono solo alla corretta realizzazione delle azioni associate alle varie transizioni. Quindi non vanno confuse con gli attributi dell'oggetto stesso.
• Il **diagramma degli stati e transizioni** è sempre corredato da detta **specifica** che ne chiarisce in dettaglio la semantica.

Diagramma delle classi

• Consideriamo il seguente diagramma delle classi:
• PlayList e Brano li abbiamo già incontrati in precedenza
• Player è una classe che non contiene alcun dato (ma a cui è associato un diagramma stati e transizioni)



Diagrammi degli stati e delle transizioni



Per le specifiche degli stati consultare slide complete

LEZIONE 4 – SLIDE 2

Progetto di classi con associato diagramma degli stati e delle transizioni

Decisioni preliminari sulla gestione degli eventi

• Per realizzare gli oggetti "reattivi", cioè con associato un diagramma degli stati e delle transizioni, sono possibili varie scelte. In particolare:
– Gli **eventi sono chiamate a funzioni che modificano lo stato dell'oggetto reattivo**:
• È uno schema realizzativo idoneo soprattutto quando l'interazione con gli oggetti è pilotata da un cliente esterno al sistema.
• È stato utilizzato in vecchie edizioni di questo corso
– Gli **eventi sono messaggi** che oggetti reattivi si scambiano:
• È uno schema realizzativo che permette una forte interazione tra i vari oggetti del sistema.
• Porta a realizzare parti del programma **orientate agli eventi**
(**event-based programming** – molto usato per interfacce grafiche, videogiochi, realizzazione di dispositivi reattivi)

Eventi come messaggi

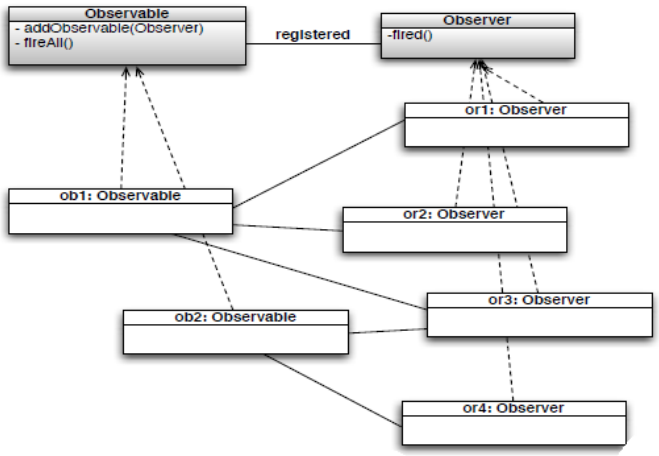
• In questa edizione del corso noi ci focalizzeremo su **eventi come messaggi**, mettendo in piedi un "framework" opportuno per la gestione degli eventi.
• Tale gestione degli eventi verrà inizialmente proposta considerando **un unico flusso di controllo** (programmazione basata su eventi, realizzata con programmi sequenziali).
• **Successivamente renderemo i flussi di controllo dei singoli oggetti indipendenti e concorrenti** (programmazione basata su eventi, realizzato con programmi multi-thread).
• Assumeremo che ogni **evento** o messaggio abbia un **mittente** ed un **destinatario** esplicito,realizzando connessioni **point-to-point.**
• Inoltre, permetteremo di mandare **messaggi in broadcasting** a tutti gli oggetti reattivi del sistema, cioè connessioni broadcasting
• Per semplicità non affronteremo il caso in cui i messaggi abbiano più di un destinatario, ma non siano in broadcasting, cioè connessioni multicasting. Va comunque osservato che quanto proposto può facilmente essere esteso a questo caso.
• Ammetteremo che un mittente possa non dichiararsi quando manda un messaggio (perché conoscere il mittente è irrilevante, o per altri motivi)

Supporto per lo scambio degli eventi:

• Lo scambio degli eventi segue sostanzialmente il **pattern Observable - Observer.**

Pattern Observable – Observer

• Un **Observable** rappresenta un oggetto osservabile da altri oggetti legati ad esso ("registrati") detti **Observer.**
• Ogni **Observer** implementa una speciale funzione per reaggire alle notifiche dell'**Observable**, qui chiamata **fired().**
• Un **Observable** registra, attraverso la funzione **addObserver(),** i suoi **Observer.**
• Quando l'**Observable** vuole notificare qualcosa, attraverso **fireAll()** chiama su ciascun **Observer** registrato il suo metodo **fired().**
NB: la comunicazione **Observable-Observer** è **unidirezionale** (responsabilità singola di **Observable**): L'**Observable** comunica ai suoi **Observer** l'avvenimento di qualcosa



• In Java esistono le classi Observable e Observer ma essendo classi e non intefacce sono di fatto deprecate, perché costringono all'uso della derivazione tra classi per riuso invece che per ISA.
• Invece il pattern è implementato in molte librerie, eg Java Swing, utlizzando il nome di **Listener** invece di **Observer**

Environment: supporto per lo scambio degli eventi

• Nel nostro caso l'idea generale del pattern **Observable-Observer** va adattata in modo opportuno, visto che tutti gli oggetti reattivi ricevono eventi ma anche lanciano eventi (**comunicazione bidirezionale**).
• Per realizzare tale comunicazione bidirezionale faremo uso di un particolare oggetto **Environment**, **che agisce da canale di comunicazione:**
– **Tutti gli oggetti reattivi manderanno all'Environment i propri eventi**
– **L'Environment si occuperà di inoltrare ciascun evento al gusto destinatario** (che, si ricorda, è scritto sull'evento stesso).

LEZIONE 4 – SLIDE 3

Realizzazione di classi con associato diagramma degli stati e delle transizioni: implementazione sequenziale

Realizzazione di oggetti reattivi in Java

• Innanzitutto, una classe con associato un diagramma degli stati e delle transizioni è una classe legata allo ad altre classi secondo il diagramma delle classi.
• Quindi vale tutto ciò che è stato detto in precedenza, relativamente alla rappresentazione degli attributi, alla partecipazione ad associazioni, alla responsabilità sulle associazioni stesse, ecc.
• In più ci si dovrà occupare del suo aspetto "reattivo" come modellato dal diagramma degli stati e delle transizioni.
• Per rappresentare tale l'aspetto reattivo secondo il diagramma degli stati e delle transizioni dobbiamo fare in modo che gli oggetti reattivi si scambino eventi.
• Per fare ciò utilizzeremo il pattern **Observable-Observer** dove l'**Observable** è un oggetto di supporto speciale chiamato **Environment.**
• Gli oggetti reattivi per tanto implementeranno tutti una speciale interfaccia Java che chiameremo semplicemente **Listener**.
•Iniziamo a descrivere tali oggetti reattivi approfondendo nell'ordine:
– Come rappresentare in Java gli **eventi**
– Come rappresentare in Java gli **stati**
– Come rappresentare in Java le **transizioni**
• Poi illustreremo la realizzazione dell'**Environment**.
– Inizialmente assumeremo ci concentreremo su una **realizzazione sequenziale dell'Environment** in cui tutti gli oggetti reattivi si scambiano eventi all'interno di un singolo thread.
– Dopo vedremo una realizzazione **concorrente dell'Environment (presente nell'esame)**

Realizzazione degli eventi

• Gli eventi sono rappresentati da oggetti di una classe Java Evento.
• La classe **Evento** rappresenta eventi generici, dotati di mittente e destinatario.
– Quando il **destinatario** è **null** allora l'evento è in **broadcasting**;
– Quando il **mittente** è **null** allora il **mittente non si è dichiarato** (rimanendo nascosto).
• Tutti gli eventi (dei diagrammi degli stati e delle transizioni) specifici per una data applicazione sono istanze di **classi derivate da Evento.**
• Gli oggetti (la cui classe più specifica è) **Evento** vengono usati direttamente solo per abilitare transizioni in casi particolari (esempio quando le transizioni non hanno un evento scatenate).
• Dal punto di vista tecnico la classe **Evento** e le sue derivate rappresentano **valori** (gli eventi / i messaggi scambiati): di fatto sono record che contengono riferimenti al mittente ed al destinatario, più (nelle classi derivate) eventuali parametri.
• Realizziamo quindi gli eventi come oggetti Java **immutabili** (*astrazione di valore realizzati con schema funzionale*) i cui metodi pubblici non fanno side-effect.
• Definiamo un opportuno **costruttore** che inizializza l'oggetto con tutte le informazioni necessarie
• Definiamo i metodi **get** per restituire dette informazioni
• Ridefiniamo **equals()** e **hashCode()** in modo che oggetti con le stesse informazioni risultino uguali
• **Non ridefiniamo** clone() visto che sono oggetti immutabili e quindi possiamo avere condivisione di memoria.

La classe Evento

```
public class Evento {
   private Listener mittente;
   private Listener destinatario;

   public Evento(Listener m, Listener d) {
     mittente = m;    //null se non rilevante
     destinatario = d;  //null se in broadcasting
   }

   public Listener getMittente() { return mittente; }
   public Listener getDestinatario() { return destinatario; }

   public boolean equals(Object o) {
      if (o != null && getClass().equals(o.getClass())) {
         Evento e = (Evento) o;
         return mittente == e.mittente &&
                destinatario == e.destinatario;
      }
      else return false;
   }

  public int hashCode() { return mittente.hashCode() + destinatario.hashCode(); }
}
```

Chiamiamo listener gli oggetti che si scambino messaggi (vedi dopo)

mitt == null se non rilevante
dest == null se evento in broadcasting

Esempio di ridefinizione di evento

```
public class MioEvento extends Evento {

   private Object info;

   public MioEvento (Listener m, Listener d, Object i) {
     super(m,d);
     if (i == null) throw RuntimeException("Payload del messaggio mancante.");
     info = i;
   }
   public Object getInfo() { return info; }
   public boolean equals(Object o) {
     if (super.equals(o)) {
        MioEvento e = (MioEvento ) o;
        return info == e.info;
     }
     else return false;
   }
  public int hashCode() { return super.hashCode() + info.hashCode(); }

}
```

Gli eventi del diagramma degli stati e delle transizioni possono avere parametri, es info

Si noti la definizione dell'equals() che rimanda alla classe padre i controlli di base: oggetto o diverso da null e della stessa classe più specifica dell'oggetto di invocazione

Si noti l'uso di super

Realizzazione degli stati

• Tipicamente rappresenteremo lo stato di un oggetto reattivo (con associato diagramma degli stati e delle transizioni) facendo uso di una specifica **rappresentazione degli stati** del diagramma.
• Scelte tipiche sono:
– una **enumerazione Java**: per costruire una costante per ogni stato del diagramma associando alle stesse un tipo (l'enumerazione stessa).
– Una serie di **constanti intere** individuali, una per ciascuno stato del diagramma (questa soluzione è peggiore della prima perché Java non associa a queste costanti un tipo specifico ma solo il tipo intero).
• Altre scelte sono possibili:
– L'uso diretto dei valori assunti dagli attributi dell'oggetto (ma è raro che questo sia possibile).
– Una rappresentazione booleana degli stati (come attraverso flip-flop) – utile per esempio, quando gli stati sono costituiti da variabili associate a specifici dispositivi.
Noi faremo praticamente sempre uso di enumerazioni.
• Per rappresentare **lo stato corrente** avremo una specifica **variabile di stato** nell'oggetto reattivo.
• Tale variabile sarà del tipo scelto per rappresentare gli stati del diagramma, quindi:
– una variabile di tipo enumerazione, se gli stati sono rappresentati da una enumerazione;
– una variabile intera, i cui valori ammissibili sono solo quelli associati alle costanti corrispondenti agli stati, nel caso gli stati sono rappresentati da costanti intere.
• Accanto alla **variabile di stato** per rappresentare **lo stato corrente**, se necessario si farà uso di eventuali **variabili di stato ausiliarie** per memorizzare dati necessari durante le transizioni
Nota, le variabili di stato ausiliarie servono a rappresentare informazioni necessarie alle transizioni che non siano già rappresentate nei campi dato dell'oggetto corrispondenti agli attributi del diagramma delle classi

Rappresentazione dello stato in Java: codice

```
public class MioOggettoConStato  implements Listener {

   //Tutto l' oggetto secondo metodologia
   //piu' :

   //Gestione dello stato

   public static enum Stato {STATO_0, STATO_1, STATO_2, /*...*/ STATO_n}

   private Stato statocorrente = Stato.STATO_0;

   private Object varAux = null;

   public Stato getStato() {
      return statocorrente
   }

   //Gestione delle transizioni
   ...
}
```

*Chiamiamo **Listener** gli oggetti che si scambino messaggi (vedi dopo)*

***Rappresentazione degli stati** come enumerazione "MioOggettoConStato.Stato"*

***Variabile di stato** per denotare lo stato corrente.*
Si noti l'inizializzazione con lo stato iniziale
(qui fatta a tempo di compilazione, poteva anche essere fatta dal costruttore)

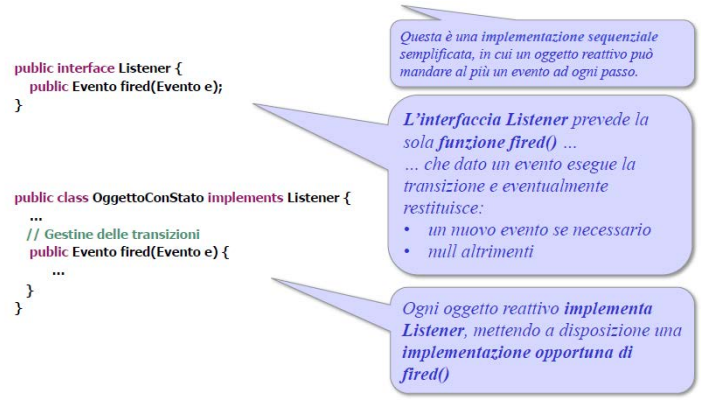
***Variabili di stato ausiliarie** (private) da usare nella gestione delle transizioni, se necessarie*

***Funzione per conoscere lo stato corrente** secondo il diagramma degli stati e delle transizioni*

Gestione delle transizioni

• La gestione delle transizioni avviene in una funzione specifica **fired():**
– Questa prende come **parametro l'evento scatenante** della transizione e **restituisce in uscita il nuovo evento** lanciato dalla azione della transizione, oppure **null** in caso l'azione non lanci eventi.
– La funzione **fired()** è specificata dall'interfaccia **Listener.**

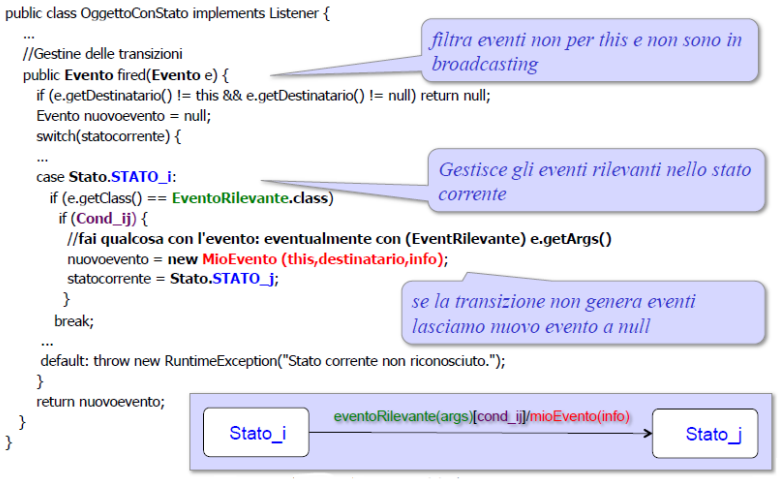
Interfaccia Listener



Gestione delle transizioni

• Il corpo della funzione **fired()** è costituito da un **case** sullo stato corrente che definisce come si risponde all'evento passato come parametro di ingresso:
• In particolare, per ogni **stato** (condizione nel case)
– Controlla la **rilevanza dell'evento**;
– Prende gli eventuali **parametri dell'evento**;
– Controlla la **condizione** che seleziona la transizione tra quelle disponibili in nello stato stesso;
– Esegue l'azione associata alla transizione:
• Fa eventualmente **side-effect** sulle proprietà dell'oggetto e in generale di tutto la realizzazione del diagramma delle classi;
• Crea e il **nuovo evento** da mandare e lo restituisce come risultato di **fired().**

Gestione delle transizioni: codice



Supporto per lo scambio degli eventi

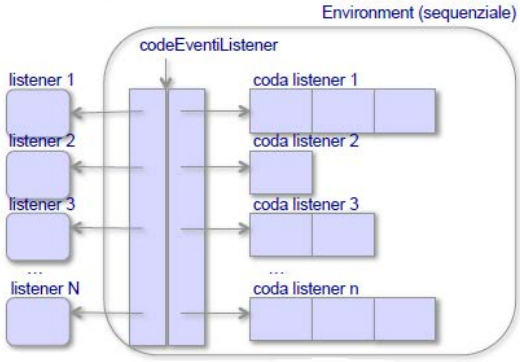
• Relativamente all'environment faremo le seguenti assunzioni:
– L'**Environment** (l'observable) lancia, ad ogni turno, un evento per ciascun **Listener** (l'observer);
– L'Environment garantisce che **l'ordine di inoltro dei messaggi**, per ciascun Listener, è l'ordine di arrivo degli stessi.
• Per fare ciò l'environment deve essere dotato **di una coda di eventi separata per ciascun Listener:**
– Avere una **struttura dati separata per ciascun Listener** garantisce la gestione indipendente di ciascun Listener e la possibilià di lanciare un evento per ciascun Listener ad ogni passo.
– Il fatto che tale **struttura dati sia una coda** garantisce l'ordinamento giusto dei messaggi.

Struttura dati Environment sequenziale

Un Map (o dizionario) con

- **Chiave**: il ***listener*** e
- **Informazione**: la ***coda degli eventi*** indirizzati al listener

```
HashMap<Listener, LinkedList<Evento>> codeEventiDeiListener;
```



Environment

```
public class Environment {
 private HashMap<Listener, LinkedList<Evento>> codeEventiDeiL

 public Environment() {
  codeEventiDeiListener = new HashMap<Listener, LinkedList<Evento>>();
 }

 public void addListener(Listener lr) {
  codeEventiDeiListener.put(lr, new LinkedList<Evento>());
 }

 public void aggiungiEvento(Evento e) {
  // aggiunge evento e nella coda del destinatario di e
  // Pre: e <> null e.getDestinatario ha una coda associata in codeEventiDeiListener
   ...
 }

public void eseguiEnvironment() {
  // finche' ci almeno un evento da processare in una delle code
  // prendi tutti i primi elementi e mandali ai rispettivi Listener
   ...
       Evento ne = listener.fired(e); // chiamata all'esecuzione del listener
   ...
 }
}
```

*Map che associa ad **ogni Listener** registrato **una coda di eventi** specifica*

Funzione per registrare Listener

Funzione per aggiungere eventi anche esogeni (esterni)
(Serve anche inserire un evento iniziale per fare partire tutto il sistema)

Manda in esecuzione il sistema

Invoca l'esecuzione di fired() dei Listener

```
public class Environment {
 private HashMap<Listener, LinkedList<Evento>> codeEventiDeiListener;

 public Environment() {
  codeEventiDeiListener = new HashMap<Listener, LinkedList<Evento>>();
 }
 public void addListener(Listener lr) {
  codeEventiDeiListener.put(lr, new LinkedList<Evento>());
 }
 public void aggiungiEvento(Evento e) {
  // aggiunge evento e nella coda del destinatario di e
  // Pre: e <> null e.getDestinatario ha una coda associata in codeEventiDeiListener
  Listener destinatario = e.getDestinatario();
  if (destinatario != null)
   // il messaggio e' per un destinatario specifico
   codeEventiDeiListener.get(destinatario).add(e);
  else {
   // destinatario == null significa che il messaggio e' in broadcasting
   Iterator<Listener> itn = codeEventiDeiListener.keySet().iterator();
   while (itn.hasNext()) {
    Listener lr = itn.next();
    codeEventiDeiListener.get(lr).add(e);
   }
  }
 }
 ...
```

*Map che associa ad **ogni Listener** registrato una coda di eventi specifica*

Funzione per registrare Listener

Funzione per aggiungere eventi anche esogeni (esterni)
Serve anche creare un evento iniziale per fare partire tutto il sistema

```
public class Environment {
 ...
 public void eseguiEnvironment() {
  boolean eventoProcessato;
  do {
   // finche' ci almeno un evento da processare in una delle code
   // prendi tutti i primi elementi e mandali ai rispettivi Listener
   eventoProcessato = false;

   // Inizio parte di codice corrispondente al fireAll()
   Iterator<Listener> it = codeEventiDeiListener.keySet().iterator();
   while (it.hasNext()) {
    Listener listener = it.next();
    LinkedList<Evento> coda = codeEventiDeiListener.get(listener);
    if (coda.isEmpty()) continue;
    Evento e = coda.remove(0);
    eventoProcessato = true;
    Evento ne = listener.fired(e); // chiamata all'esecuzione del listener
    if (ne == null) continue;
    aggiungiEvento(ne); // aggiunge evento ne nella coda del destinatario di ne
   }
   // Fine parte di codice corrispondente al fireAll()

  } while (eventoProcessato);
 }
}
```

Manda in esecuzione il sistema

Questa parte di codice corrisponde al fireAll() dell'observable, ma con in più la ricezione di eventi e la registrazione che qualche evento è stato processato, per abilitare la terminazione di eseguiEnvironmet()

Chiamata all'esecuzione del Listener: aggiunge evento ne nella coda del destinatario di ne (se ne != null)

Realizzazione di classi con associato diagramma degli stati e delle transizioni: implementazione concorrente

Progetto e realizzazione di diagrammi stati e transizioni in presenza di concorrenza

• Ora andiamo a studiare classi con associato un diagramma degli stati e delle transizioni in presenza di attività concorrenti.
• La classe naturalmente è una classe legata ad altre classi secondo il diagramma delle classi. Quindi vale tutto ciò che è stato detto in precedenza, relativamente alla rappresentazione degli attributi, alla partecipazione ad associazioni, alla responsabilità sulle associazioni stesse, ecc.
• In più ci si dovrà occupare del suo aspetto "reattivo" come modellato dal diagramma degli stati e delle transizioni, tenendo presente che l'attività di ricezione elaborazione e invio di eventi è solo una delle attività della applicazione che agiscono sul diagramma delle classi.
• Di fatto andremo ad associare a ciascun oggetto reattivo un thread separato per la gestione degli eventi.
• Avremo quindi un'applicazione in cui conviveranno thread dedicati alle attività del diagramma delle attività e thread dedicati alla gestione degli eventi.
• Questo richiederà da una parte la realizzazione di un environment più sofisticato.
• Dall'altra una gestione degli stati e delle transizioni che gestisca la concorrenza in modo opportuno (come già facciamo per le attività del diagramma delle attività).

Realizzazione di oggetti reattivi concorrenti

• Come precedentemente avremo una rappresentazione in Java di:
– **Eventi e stati** che sarà identica a prima.
– **Transizioni** che sarà invece diversa per sfruttare la concorrenza.
• **La realizzazione dell'Environment sarà completamente concorrente**.
– Ogni **oggetto reattivo** esegue il suo diagramma stati transizioni su un **thread separato.**
– Anche **l'Environment** lavora su un **thread separato.**
– L'Environment è dotato di una **struttura dati condivisa thread-safe** per lo scambio degli eventi. Tale struttura dati è accessibile dai thread degli oggetti reattivi oltre che dal thread dell'environment stesso.

Interfaccia Listener (modificata per concorrenza)

Per prima cosa modificare l'interfaccia **Listener**, rimuovendo l'evento restituito.
Gli eventi generati (che ora potranno essere 0, uno, o più) saranno inseriti direttamente nella struttura dati thread safe dell'Environment concorrente.

```
public interface Listener {
   public void fired(Evento e);
}
```

L'interfaccia Listener prevede la sola funzione fired() ... ... che dato un evento esegue la transizione e eventualmente restituisce un nuovo evento o null se il nuvo evento non c'e'

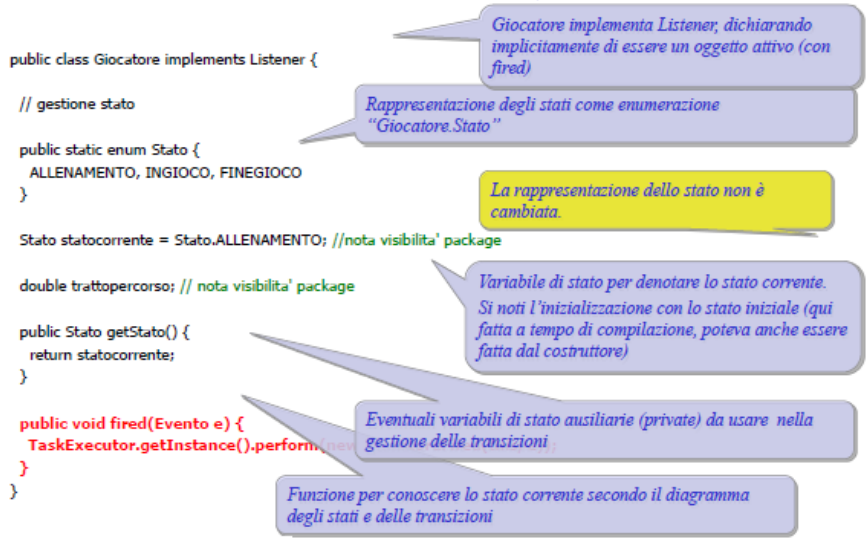
```
public class OggettoConStato implements Listener {
   ...
  // Gestine delle transizioni
  public void fired(Evento e) {
      ...
  }
}
```

Ogni oggetto reattivo implementa Listener, mettendo a disposizione una implementazione opportuna di fired()

Realizzazione di oggetti reattivi concorrenti

Un oggetto reattivo, cioè che implementa l'interfaccia **Listener** sarà analogo a prima, tranne per la funzione **fired().**

Gestione degli stati e delle transizioni con concorrenza



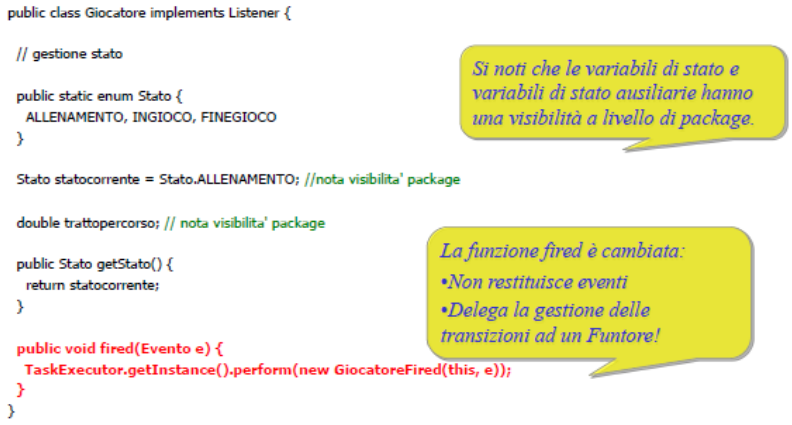
Realizzazione di oggetti reattivi concorrenti

La funzione **fired()** deve ora lavorare con il diagramma delle classi condiviso da tutti i thread. Quindi deve diventare analogo ad un **Task** dei diagrammi delle attività.

Per fare ciò la funzione **fired()** invocherà semplicemente sull'istanza (singleton) di **TaskExecutor** l'esecuzione di un **funtore** che conterrà tutta la logica delle transizioni (ciò che prima stava nel corpo del **fired()** nel caso non concorrente).

Il nuovo funtore NON sarà pubblico e sarà accessibile solo dall'oggetto reattivo corrispondente.

Per fare ciò faremo usa della visibilità di **Package.**



Gestione delle transizioni

• La gestione delle transizioni avviene nella funzione **fired().**

• Questa prende come parametro **l'evento scatenante** della transizione e restituisce in uscita il **nuovo evento** lanciato dalla azione della transizione, oppure **null** in caso l'azione non lanci eventi

• **Eventi restituiti:**

– Si noti: se fired() restituisce un evento come output della funzione a quale thread lo rende disponibile? A quello corrente soltanto!!!

– Invece noi dobbiamo renderlo disponibile a oggetti che lavorano su thread separati.

Per fare ciò faremo uso di una esplicita istruzione di **inserimento dell'evento nell'environment:**

Environment.aggiungiEvento(new Evento(…));

• Il corpo della funzione fired() deve essere eseguito concorrentemente agli altri thread e può accedere al diagramma delle classi! Quindi deve essere costituito da una **chiamata** all'esecuzione di un **funtore di tipo Task.**

• Il funtore che realizza **fired()** associato ad una classe NomeClasse lo chiameremo sempre **NomeClasseFired.**

• Questo funtore non deve essere acceduto dai clienti della classe perché contiene codice di interesse solo per la classe stessa: per fare ciò gli diamo visibilità a livello di package e lo metteremo nel package della classe (che invece è pubblica).

• Il codice del funtore deve poter accedere alle variabili di stato in generale, ecco perché ora queste hanno visibilità a livello di package (invece che private).

• Il corpo funtore (cioè la funzione **esegui())** fa quello che nel caso non concorrente fa direttamente **fired().** Cioè è costituito da un case sullo stato corrente che definisce come si risponde all'evento in ingresso:
– Controlla la **rilevanza dell'evento;**
– Controlla la **condizione** che seleziona la transizione;
– Prende gli eventuali **parametri dell'evento,**
– Fa eventualmente **side-effect** sulle proprietà (campi dati) dell'oggetto;
– Crea e il **nuovo evento** da mandare e lo restituisce lo inserisce nell'environment con un'istruzione del tipo:
Environment.aggiungiEvento(new Evento(…));

Codice

```
class GiocatoreFired implements Task {
 private boolean eseguita = false;
 private Giocatore g;
 private Evento e;

 public GiocatoreFired(Giocatore g, Evento e) {  this.g = g; this.e = e; }

public synchronized void esegui() {
  if (eseguita || (e.getDestinatario() != g && e.getDestinatario() != null))
   return;
  eseguita = true;
  switch (g.getStato()) {
  ...
  case INGIOCO:
   if (e.getClass() == Bastone.class)
    if (g.trattopercorso < 100) {
      ...
      Environment.aggiungiEvento(new Bastone(g, g));
      g.statocorrente = Stato.INGIOCO;
    } else { // trattopercorso >= 100
      ...
    }
   break;
  ...
  default:
   throw new RuntimeException("Stato corrente non riconosciuto.");
  }
 }

public synchronized boolean estEseguita() { return eseguita; }
}
```

Implementa il pattern funtore ed in particolare l'interfaccia Task.

Gestisce gli eventi rilevanti nello stato corrente con esattamente la stessa logica del caso non concorrente.

Mette l'evento nell'environment.

Environment nel caso concorrente

• Nel nostro caso l'idea generale del pattern **observable-observer** dove l'observable è un **Environment** e gli observer sono **Listener** (di tipo concorrente).
• L'Environment questa volta è diviso in due classi che trattano aspetti separati:
– Una classe detta **Environment** che conterrà una struttura dati thread safe formata da una **coda bloccante** per ciascun Listener dove memorizzare gli eventi che gli arrivano e sono in attesa di essere processati.
– Una classe detta **EsecuzioneEnvironment** che si occuperà degli aspetti dinamici relativi all'environment: la sua inizializzazione, attivazione e spegnimento.
Osservazione: pattern singleton realizzato con classi con tutti metodi static
• Le classi
  – **Environment**
  – **EsecuzioneEnvironment**

devono realizzare rispettivamente **un unico oggetto Environment** a **un unico oggetto EsecuzioneEnvironment**.
• Quindi potrebbero essere realizzate con il **pattern singleton.**
• Tuttavia, le realizzeremo in modo diverso questa volta, come classi in cui tutti I metodi sono **static.**
• In questo modo I due oggetti che realizzano Environment e EsecuzioneEnvironment diventano semplicemente
  – **Environment.class**
  – **EsecuzioneEnvironment.class**

Environment (struttura dati thread safe di supporto per scambio eventi)

```
public final class Environment { // NB con final non si possono definire sottoclassi
        private Environment() { // NB non si possono costruire oggetti Environment
        }

        private static ConcurrentHashMap<Listener, LinkedBlockingQueue<Evento>> codeEventiDeiListener =
                new ConcurrentHashMap<Listener, LinkedBlockingQueue<Evento>>();

        public static void addListener(Listener lr, EsecuzioneEnvironment e) {...}

        public static Set<Listener> getInsiemeListener() {...}

        public static void aggiungiEvento(Evento e) {...}

        public static Evento prossimoEvento(Listener lr)  {...}
}
```

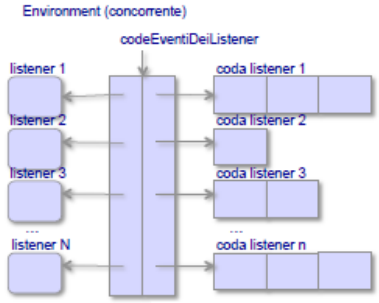
Nota!

Tutti devono avere accesso all'environment sempre! L'environment stesso gestisce accesso concorrente alle proprie strutture dati attraverso ConcurrentHashMap e soprattutto LinkedBlockingQueue

Struttura dati Environment concorrente

Una **ConcurrentMap** (o dizionario thread safe) con
- **Chiave**: il **listener** e
- **Informazione**: la **coda bloccante** degli eventi indirizzati al listener

```
ConcurrentHashMap<Listener, LinkedBlockingQueue<Evento>> codeEventiDeiListener
```



Environment (stuttura dati thread safe di supporto per scambio eventi)

```
public final class Environment { // NB con final non si possono definire sottoclassi
        private Environment() { // NB non si possono costruire oggetti Environment
        }

        private static ConcurrentHashMap<Listener, LinkedBlockingQueue<Evento>> codeEventiDeiListener =
                new ConcurrentHashMap<Listener, LinkedBlockingQueue<Evento>>();

        public static void addListener(Listener lr, EsecuzioneEnvironment e) {
                if (e == null) return;
                codeEventiDeiListener.put(lr, new LinkedBlockingQueue<Evento>());
                // Nota Listener inserito ma non attivo
        }

        public static Set<Listener> getInsiemeListener() {
                return codeEventiDeiListener.keySet();
        }

        public static void aggiungiEvento(Evento e) {...}

        public static Evento prossimoEvento(Listener lr)
                throws InterruptedException {
                // nota NON deve essere synchronized!!!
                return codeEventiDeiListener.get(lr).take();
        }
}
```

Nota!

Pseudoprivato!

Tutti devono avere accesso all'environment sempre! L'environment stesso gestisce accesso concorrente alle proprie strutture dati attraverso ConcurrentHashMap e soprattutto LinkedBlockingQueue

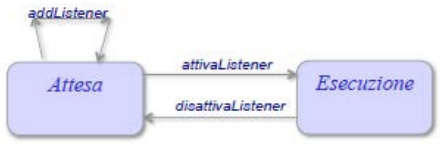
Ha la stessa logica di aggiungiEvento nel caso senza concorrenza…

```
...
        public static void aggiungiEvento(Evento e) {
                // unico meccanismo per aggiungere eventi
                if (e == null) return;
                Listener destinatario = e.getDestinatario();
                if (destinatario != null && codeEventiDeiListener.containsKey(destinatario)) {
                        // evento per un destinatario attivo
                        try {
                                codeEventiDeiListener.get(destinatario).put(e);
                        } catch (InterruptedException e1) {
                                e1.printStackTrace();
                        }
                } else if (destinatario == null) {
                        // evento in broadcasting
                        Iterator<Listener> itn = codeEventiDeiListener.keySet().iterator();
                        while (itn.hasNext()) {
                                Listener lr = itn.next();
                                try {
                                        codeEventiDeiListener.get(lr).put(e);
                                } catch (InterruptedException e1) {
                                        e1.printStackTrace();
                                }
                        }
                }
        }
...
```

… ma ora utilizza code bloccanti(!!!) e map che permettono l'accesso concorrente

EsecuzioneEnvironment

• La classe **EsecuzioneEnvironment** è equipaggiata con tre metodi, rispettivamente
– per **aggiungere Listener**,
– per **far partire tutti i Listener,**
– per **fermare tutti i Listener.**
• Queste funzioni però possono essere usate solo in certi stati, vedi diagramma degli stati (qui usato per limitare le invocazioni hai metodi e non per scambiare eventi).



EsecuzioneEnvironment

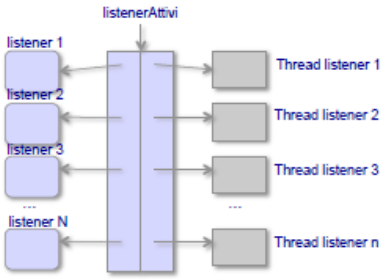
```
public final class EsecuzioneEnvironment { //NB con final non si possono definire sottoclassi
        private EsecuzioneEnvironment() {
        }

        private static ConcurrentHashMap<Listener, Thread> listenerAttivi = null;

        public static enum Stato {
                Attesa, Esecuzione
        };

        private static Stato statocorrente = Stato.Attesa;

        public static synchronized void addListener(Listener lr) {
                if (statocorrente == Stato.Attesa) {...}
        }

        public static synchronized void attivaListener() {
                if (statocorrente == Stato.Attesa) { statocorrente = Stato.Esecuzione; ...}
        }
        public static synchronized void disattivaListener() {
                if (statocorrente == Stato.Esecuzione) { statocorrente = Stato.Attesa; ...}
        }

        public static synchronized Stato getStato() {
                return statocorrente;
        }
```

I Listener sono eseguiti in thread separati

EsecuzioneEnvironment

- La classe **EsecuzioneEnvironment** svolge il suo lavoro attivando un thread per ogni oggetto Listener.
- Per fare ciò mantiene una struttura dati che associa ad ogni **Listener** un oggetto Java **Thread** corrispondente al proprio thread:

ConcurrentHashMap<Listener, Thread> listenerAttivi



EsecuzioneEnvironment

La struttura dati **private static ConcurrentHashMap<Listener, Thread> listenerAttivi** è manipolata dalle funzioni:
- **addListener()** che aggiunge una coppia **Listener, Thread** ad esso associato.
- **attivaListener()** che manda in esecuzione ciascun **Listener**, invocando **start()** sul **Thread** ad esso associato.
- **disattivaListener()** che:
- **avvelena** ciascun thread mettendo in coda agli eventi di ciascun **Listener** un evento speciale **stop;**
- Aspetta la fine del thread di ciascun **Listener,** con **join().**

EsecuzioneEnvironment

La classe EsecuzioneEnvironment è equipaggiata con tre metodi, rispettivamente per aggiungere Listener , per far partire tutti i listener, per fermare tutti i listener.
Queste funzioni però possono essere usate solo in certi stati, vedi diagramma degli stati (qui usato per limitare le invocazioni hai metodi e non per scambiare eventi.

```
/* Serve ad aggiungere i singoli listener, ed ad attivarli e disattivali (tutti insieme)
 * Nota: implementa il seguente diagramma degli stati e delle transizioni :
 *
 *   addListener
 *
 *     | |  --attivaListener->
 * --> Attesa              Esecuzione
 *       <--disattivaListener-
 */
public final class EsecuzioneEnvironment { //NB con final non si possono definire sottoclassi
        private EsecuzioneEnvironment() {
        }

        public static enum Stato {
                Attesa, Esecuzione
        };

        private static Stato statocorrente = Stato.Attesa;
        private static ConcurrentHashMap<Listener, Thread> listenerAttivi = null;
        ...
```

I Listener sono eseguiti in thread separati

Faremo uso di un evento speciale Stop che useremo per segnalare ai vari listener di terminare.

```
public static synchronized void addListener(Listener lr) {
        if (statocorrente == Stato.Attesa) {
                Environment.addListener(lr, new EsecuzioneEnvironment());
                //NB: Listener inserito ma non attivo
        }
}

public static synchronized void attivaListener() {
        if (statocorrente == Stato.Attesa) {
                statocorrente = Stato.Esecuzione;
                System.out.println("Ora attiviamo i listener");
                listenerAttivi = new ConcurrentHashMap<Listener, Thread>();
                Iterator<Listener> it = Environment.getInsiemeListener().iterator();
                while (it.hasNext()) {
                        Listener listener = it.next();
                        listenerAttivi.put(listener, new Thread(
                                        new EsecuzioneListener(listener)));
                }
                Iterator<Listener> i = listenerAttivi.keySet().iterator();
                while (i.hasNext()) {
                        Listener l = i.next();
                        listenerAttivi.get(l).start();
                }
        }
}
```

Attiva tutti i Listener presenti nell'environment.

EsecuzioneListener è l'eseguibile associato al Thread (vedi dopo).

Disattiva tutti i Listener presenti nell'environment, mandandogli l'evento stop.

```
        public static synchronized void disattivaListener() {
                if (statocorrente == Stato.Esecuzione) {
                        statocorrente = Stato.Attesa;
                        System.out.println("Ora fermiano i listener");
                        Environment.aggiungiEvento(new Stop(null, null));
                        // NB: a questo punto i listener non sono ancora fermi
                        // ma l'evento Stop e' stato inserito nella coda di ciascuno di loro
                        // e questo evento quando processato li disattivera'
                        Iterator<Listener> it = listenerAttivi.keySet().iterator();
                        while (it.hasNext()) {
                                Listener listener = it.next();
                                try {
                                        Thread thread = listenerAttivi.get(listener);
                                        thread.join();
                                } catch (InterruptedException e1) {
                                        e1.printStackTrace();
                                }
                        }
                }
        }

        public static synchronized Stato getStato() {
                return statocorrente;
        }
}
```

EsecuzioneListener

• Implementa il pattern **Funtore** con:
- **Thread** come **esecutore**
- **EsecuzioneListener** come **eseguibile** *(NB: esattamente come le attività complesse del diagramma delle attività)*
• Prende il **prossimo evento** dalla coda associata al suo **Listener** e lancia il **fired()** del suo **Listener.**
• Tratta in modo speciale l'evento **Stop** (predefinito nel framework) che serve a fare terminare il thread.
– **Non lo passa al fired()** dal suo Listener che deve rimanere associato al solo diagramma stati transizioni,
– Ma lo **processa direttamente.**

```
class EsecuzioneListener implements Runnable { //NB: non e' pubblica, serve solo nel package
        private boolean eseguita = false;
        private Listener listener;

        public EsecuzioneListener(Listener l) {
                listener = l;
        }

        public synchronized void run() {
                if (eseguita) return;
                eseguita = true;
                while (true) {
                        try {
                                Evento e = Environment.prossimoEvento(listener);
                                if (e.getClass() == Stop.class) return;
                                listener.fired(e);
                        } catch (InterruptedException ex) {
                                ex.printStackTrace();
                        }
                }
        }

        public synchronized boolean estEseguita() {
                return eseguita;
        }
}
```

Implementa il pattern funtore con:
- *Thread come esecutore*
- *EsecuzioneListener come eseguibile*

(NB: esattamente come le attività complesse del diagramma delle attività)

Stop è l'evento speciale che serve a fare terminare il thread.

(NB: è processato qui, e non in fired() che deve rimanere associato al solo diagramma stati transizioni.)