MAP - Metodi Avanzati di Programmazione

Marco Ferrara

Marzo 2024

Contents

1 Astrazione nella progettazione

Il termine **astrazione** deriva dal latino *asbstrahere* che vuol dire "*trascinare via*", dal verbo *trahere* che significa *trascinare*. Intendiamo trascinare un concetto, un'idea o un principio da una realtà concreta.
In ambito scientifico, astrarre significa cambiare la rappresentazione di un problema.
Qual è l'obiettivo?
L'obiettivo del cambio di rappresentazione è quello di concentrarsi su **aspetti rilevanti**, tralasciando gli elementi *incidentali*.

L'astrazione si focalizza sulle caratteristiche essenziali di un oggetto, rispetto alla *prospettiva* di colui che osserva.

Il termine astrazione sottointende:

- **un processo:** l'astrazione delle informazioni essenziali e rilevanti per un particolare scopo, ignorando il resto dell'informazione.
- **una entità:** una descrizione semplificata di un sistema che mette in risalto alcuni dei dettagli o proprietà, trascurando il resto.

Nota: entrambi le viste sono valide.

1.1 Astrazione e software

Nella **programmazione** il termine astrazione fa riferimento alla distinzione tra:

- **cosa:** *what* $\rightarrow$ cioè cosa fa un pezzo di codice.
- **come:** *how* $\rightarrow$ cioè come esso è implementato.

Per l'utente del codice l'essenziale è cosa fa il codice, non è interessato ai dettagli dell'implementazione.

Poichè i sistemi software diventano sempre più complessi, per padroneggiare la **complessità**, è neccessario che ci si concentri soltanto su pochi aspetti che interessano maggiormente il contesto.
L'astrazione permette ai progettisti di sistemi software di risolvere *problemi complessi* in maniera organizzata e facilmente gestibile.

1.2 Astrazione funzionale

L'astrazione funzionale si riferisce alla **progettazione del software**, in particolar modo alla possibilità di specificare un modulo software che vada a trasformare dei dati in input in dati di output, nascondendo quelli che sono i dettagli algoritmici della progettazione.

Sintetizzando:

- il modulo software deve trasformare un input in un output, cioè deve calcolare una funzione;
- i dettagli della trasformazione (cioè del calcolo) non sono visibili al consumatore (finale) del modulo;
- il consumatore conosce solo le convenzioni corrette di chiamate (chiamate **specifica sintattica**) e cosa fa il modulo (chiamata **specifica semantica**);
- il consumatore deve fidarsi del risultato.

Esempio: modulo che realizza un operatore per il calcolo del fattoriale.
La *specifica sintattica* indica il nome del modulo (es. fatt), il tipo di dato passato in input e il tipo di risultato che si ottiene, cosi da permettere la chiamata corretta della funzione.

Ad esempio: fatt(intero) $\rightarrow$ intero

La *specifica semantica* indica, invece, la trasformazione operata, cioè proprio la funzione calcolata:

$$fatt(n) = \begin{cases} n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1 & \text{se n >= 1} \\ 1 & \text{se n = 0} \end{cases}$$

Come specificare la semantica del modulo?
Un modo è quello di esprimere, attraverso due *predicati*, la relazione che lega i dati di ingresso ai dati di uscita: se il primo predicato (detto **precondizione**) è vero sui dati in ingresso e se il programma termina su quei dati, allora il secondo predicato (detto **postcondizione**) è vero sui dati in uscita.
Queste specifiche semnatiche sono chiamate **assiomatiche**.

Ritornando all'esempio del fattoriale:
Precondizione: $n \in \mathbb{N}$
Postcondizione:

$$fatt(n) = \begin{cases} n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1 & \text{se n >= 1} \\ 1 & \text{se n = 0} \end{cases}$$

Esercizio: Un modulo ha in input un intero x e un vettore A di n interi. Restituisce un intero p e un vettore di B interi.
Siano date le seguenti condizioni:

- Precondizione: $n > 0 \wedge \forall i \in [1, n] A[i] \in Z$
- Postcondizione: $\forall i \in [1, n] \exists j \in [1, n] B[i] = A[i] \wedge \forall i \in [1, p-1] B[i] <= x \wedge \forall i \in [p+1, n] B[i] > x$

Qual è la funzione in questo modulo?
Continua..

1.3 Limiti dell'astrazione funzionale

L'astrazione funzionale non permette di progettare (e quindi sviluppare) moduli software *invarianti ai cambiamenti nei dati*, in quanto sono invarianti solo ai cambiamenti nei processi di trasformazione che operano.
Tutto ciò rende difficoltosa la manutenzione delle soluzioni progettate e risulta inappropriata per lo sviluppo di soluzioni per problemi complessi.

1.4 Principio dell'astrazione dati

Alla base dell'astrazione dati c'è il principio che **non si può accedere direttamente alla rappresentazione di un dato** qualunque esso sia, ma si può accedere soltanto attraverso un insieme di operazioni che devono essere considerate lecite.
Il vantaggio è che nel momento in cui c'è un cambiamento nella rappresentazione di un dato, questo si ripercuoterà soltanto sulle operazione lecite, le quali potrebbero subire delle modifiche; ma non porterà nessuna modifica sul codice che utilizza il dato astratto.

Esempio: Se i moduli "Rimuovi duplicati" e "Ricerca elemento" accedono all'elenco attraverso un insieme di operazioni (ad esempio 'dammi il prossimo elemento', o 'non ci sono più elementi'), il cambiamento della rappresentazione dell'elenco richiederà una riformulazione delle operazioni lecite ma non andrà ad influenzare i due moduli.

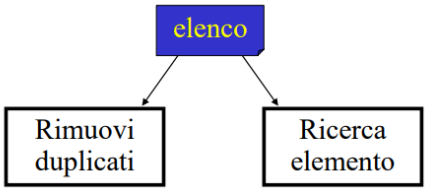


Figure 1: Esempio

1.5 Information Hiding

In generale, un principio di astrazione suggerisce di **occultare l'informazione** (*information hiding*) sulla rappresentazione del dato perchè:

- non è necessaria al consumatore dell'entità astratta

- la sua rivelazione creerebbe delle dipendenze inutili che andrebbero a compromettere l'invarianza ai cambiamenti

Il principio dell'astrazione funzionale suggerisce di occultare i dettagli del processo di trasformazione, cioè come esso è operato.
Il principio dell'astrazione dati identifica nella rappresentazione del dato l'informazione da nascondere.
Nota: in entrambi i casi non si dice COME farlo.

Figure 2: Information Hiding

1.6 Incapsulamento (encapsulation)

Cos'è? E' una tecnica di progettazione che consiste nell'**impacchettare** (o *racchiudere in capsule*) una collezione di entità, andando a creare una barriera concettuale.

Esso sottointende:

- **un processo:** l'impacchettamento
- **una entità:** il pacchetto ottenuto; ad essa corrispondono tecniche di programmazione che consentono l'incapsulamento dei dati.

Esempi:

- una procedura impachetta diversi comandi;
- una libreria incapsula diverse funzioni;
- un oggetto incapsula un dato e un insieme di operazioni sul dato.

Inoltre, l'incapsulamento non dice *come* devono essere le **pareti del pacchetto** o della capsula, le quali potranno essere:

1. **trasparenti:** permettono di vedere tutto quello che è stato impacchettato;
2. **traslucide:** permettono di vedere in modo parziale il contenuto;
3. **opache:** nascondono tutto il contenuto del pacchetto.

1.7 Astrazione dati AND Incapsulamento

Andando a combinare il principio di astrazione dati e la tecnica dell'incapsulamento, si ha che:

- la rappresentazione del dato va nascosta;
- L'accesso al dato deve passare solo attraverso operazioni lecite;
- Le operazioni lecite (che devono avere accesso all'informazione sulla rappresentazione del dato) vanno impacchettate con la rappresentazione del dato stesso.

Esempio: il dato "conto corrente" ha una sua rappresentazione interna che permette di memorizzare il *saldo* (balance), il *limite fido* (overcraft limit) e il *numero di conto* (account number). la rappresentazione dei dati, come già detto in precedente, è nascosta e l'accesso alla rappresentazione stessa passa per quattro operazioni lecite:

- creazione conto
- deposito
- prelievo
- stampa saldo

La rappresentazione e le operazioni lecite sono impacchettate in un modulo.

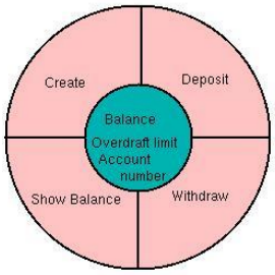


Figure 3: Esempio

Ovviamente, l'*isolamento dei moduli* **non** può essere totale.
In generale, la **specifica (o contratto)** descrive come si può interagire con un dato astratto.

1.8 Astrazione dati VS Astrazione funzionale

L'astrazione dati ricalca ed estende quella funzionale.
Attualmente la possibilità di effettuare astrazioni di dati è considerata importante almeno quanto quella di definire nuovi operatori con astrazioni funzionali.

La scelta delle strutture dati è il primo passo importante per avere un buon risultato dell'attività di programmazione.

L'**astrazione funzionale** stimola gli sforzi per evidenziare operazioni ricorrenti o comunque ben caratterizzate all'interno della soluzione del problema.

L'**astrazione di dati** stimola, invece, in più gli sforzi per andare ad individuare le organizzazioni di dati più consone alla soluzione del problema.

Si va quindi da una progettazione **function centered** a una **data centered**.

1.9 Punti di vista dell'astrazione

Solitamente, le astrazioni supportano la separazione dei diversi interessi di

- utenti: i quali sono interessati a cosa si astrae (what)
- implementatori: i quali sono interessati a come (how) si astrae
- specifica
- realizzazione

1.9.1 Specifica

Per descrivere una specifica bisogna ricorrere a *linguaggi di specifica*, i quali sono diversi dai linguaggi usati per descrivere le realizzazione delle astrazioni.

In generale, la specifica può essere:

- **sintattica:** la quale stabilisce quali identificatori sono associati all'astrazione;
- **semantica:** la quale definisce il risultato della computazione inclusa nell'astrazione.

1.9.2 Parametrizzazione di un'astrazione

Per rendere migliore l'efficacia di un'astrazione, si possono utilizzare i **parametri** per la comunicazione con l'ambiente esterno.
Quando un'astrazione viene chiamata, ogni parametro formale viene associato al corrispondente argomento.

Attenzione: Ritornando all'astrazione dati, anche quest'ultima (come qualunque altra astrazione), è formata da una **specifica** e una **realizzazione**, dove:

- la specifica consente di descrivere un nuovo dato e gli operatori che si possono applicare ad esso;
- la realizzazione stabilisce come il nuovo dato e i nuovi operatori vengono ricondotti ai dati e agli operatori che sono già disponibili.

1.10 Linguaggi di specifica

I **linguaggi di specifica** per astrazione dati più conosciuti sono due:

1. il *linguaggio logico-matematico* usato nelle asserzioni → **specifiche assiomatiche**
2. il *linguaggio dell'algebra* usato nelle equ azioni definite tra gli operatori che sono stati specificati nel dato astratto → **specifiche algebriche**

1.10.1 Specifiche assiomatiche

Parliamo di specifica assiomatica quando un linguaggio formale per la specifica di un tipo di dato astratto è formato dalla notazione logico-matematica delle *asserzioni*.
Una specifica assiomatica è formata da:

1. una *specifica sintattica* (chiamata segnatura), la quale fornisce
   - l'elenco dei nomi dei domini e delle operazioni specifiche del tipo
   - i domini sia di partenza che di arrivo per ogni nome di operatore
2. una *specifica semantica* che associa:
   - un insieme ad ogni nome di tipo introdotto nella specifica sintattica
   - una funzione ad ogni nome di operatore, esplicitando sui domini di arrivo e partenza sia la precondizione (che definisce quando un operatore è applicabile) che la postcondizione (che stabilisce la relazione tra argomenti e risultato)

L'esempio più elementare di una specifica assiomatica è quella di un vettore.

1.10.2 Esempio: Il dato astratto Vettore

Definiamo la specifica sintattica e semantica.

- *Specifica sintattica:*

  Tipi: vettore, intero, tipoelem

  Operatori:
  CREA VETTORE() → vettore
  LEGGIVETTORE(vettore, intero) → tipoelem
  SCRIVIVETTORE(vettore, intero, tipoelem) → vettore

- *Specifica semantica:*

  Tipi:
  -intero: l'insieme dei numeri interi

-vettore: l'insieme delle sequenze di n elementi di tipo *element*

Operatori:

CREA VETTORE = v
→ Pre: non ci sono precondizioni o più precisamente il predicato che definisce la precondizione di CREA VETTORE è il valore vero
→ Post: $\forall i \in \{0, 1, 2, \ldots, n-1\}$, l'$i - esimo$ elemento del vettore, $v(i)$, è uguale ad un prefissato elemento di tipo tipoelem

LEGGIVETTORE(v,i) = e
→ Pre: $0 <= i <= n-1$
→ Post: e=v(i)

SCRIVIVETTORE(v,i,e) = v'
→ Pre: $0 <= i <= n-1$
→ Post: $\forall j \in \{0, 1, \ldots, n-1\}, j \neq i, v'(j) = v(j), v'(i) = e$

- *Realizzazione:*

  In Java il vettore (chiamato *array*) è un tipo di dato concreto. Vediamo la corrispondenza tra la specifica introdotta a quella del Java:

  CREA VETTORE ↔ tipoelem v[n]
  LEGGIVETTORE(v,i) ↔ v[i]
  SCRIVIVETTORE(v,i,e) ↔ v[i]=e

dove i può essere anche un'espressione di tipo intero.

1.10.3 Limiti delle specifiche assiomatiche

Notiamo che il metodo di specifiche assiomatica risulta preciso nella definizione della specifica sintattica ma risulta piuttosto informale per altri aspetti, tanto che delle volte si ricorre al linguaggio naturale per semplicità.
In particolare, non consente di definire i valori che possono essere generati mediante l'applicazione di operatori e non consente di stabilire quando l'applicazione di diverse sequenze di operatori porta allo stesso valore.

Questo problema è superato dalle **specifiche algebriche**.

1.10.4 Specifiche Algebriche

Le specifiche algebriche si basano sull'algebra, piuttosto che sulla logica. Essenzialmente definiscono un **dato astratto** come un'**algebra eterogenea**, ovvero come una collezione di diversi insiemi su cui sono definite diverse operazioni. Le algebre tradizionali sono omogenee. Un'algebra omogenea consiste in un unico insieme e diverse operazioni.

Esempio: $\mathbb{Z}$ con le operazioni di addizione e moltiplicazione è un'algebra omogenea.

Esempio: Dato un alfabeto Σ, indichiamo con Σ^* l'insieme di tutte le stringhe, incluso quella vuota, costruite con i simboli di Σ. $\Sigma*$ con le operazioni di concatenaione e calcolo della lunghezza non sono un'algebra omogenea, visto che il codominio dell'operazione di calcolo della lunghezza è $\mathbb{N}$ e non Σ^*. Quindi quest'algebra consiste in due insiemi, stringhe e interi, su cui sono definite le operazioni di concatenamento e lunghezza delle stringhe.

Specifica algebrica Una specifica algebrica consiste in tre parti:

1. **Sintattica**: elenca i nomi del tipo, le sue operazioni e il tipo degli argomenti delle operazioni. Se un'operazione è una funzione allora è specificato il codominio (range) della funzione.
2. **Semantica**: consiste in un insieme di equazioni algebriche che descrivono in modo indipendente dalla rappresentazione delle proprietà delle operazioni.
3. **Di restrizione**: stabilisce varie condizioni che devono essere soddisfatte o prima che siano applicate le operazioni o dopo che esse siano state completate.

Alcuni autori inglobano le specifiche di restrizione in quelle semantiche.

Uno degli aspetti più interessanti delle specifiche algebriche è la semplicità del linguaggio di specifica rispetto ai liinguaggi di programmazione procedurale. Infatti, il linguaggio di specifica consiste in solo cinque primitive:

1. composizione funzionale:
2. relazione di eguaglianza
3. costante true
4. costante false
5. un numero illimitato di variabili libere (a differenza della specifica semantica assiomatica che definisce le variabili tramite quantificatori).

La funzione matematica ***if then else***, che ha come dominio il prodotto cartesiano fra un boolean e due statement (composizioni di funzioni) e come codominio uno statement, può essere facilmente descritta dalle seguenti equazioni:

$$\textit{if then else (true, q, r)} = q$$
$$\textit{if then else (false, q, r)} = r$$

Questa funzione è così importante che si assume già data come operatore infisso:

$$\textit{if } p \textit{ then } q \textit{ else } r$$

Inoltre si assume che sono predefiniti i valori interi e booleani.

1.11 Specifiche Algebriche

1.11.1 Specifica algebrica di Pila

Una **pila** è una struttura dati lineare che segue il principio '*LIFO*' (Last In, First Out), il che significa che l'ultimo elemento inserito è il primo ad essere rimosso. Puoi immaginare una pila come un insieme di oggetti impilati uno sopra l'altro, dove l'unico modo per accedere agli elementi è rimuovere l'elemento più recentemente inserito. Ogni insieme è detto **sort** (letteralmente 'tipo') dell'algebra eterogenea. I sort item e boolean sono ausiliari alla definizione di stack.

- **Specifica sintattica**:
  - *sorts*: stack, item, boolen
  - *operations*:
    - newstack() → stack
    - push(stack, item) → stack
    - pop(stack) → stack
    - top(stack) → item
    - isnew(stack) → boolean
- **Specifica semantica**: Nella specifica semantica occorre dichiarare i parametri su cui lavorano gli operatori. *declare* **stk**: stack, **i**: item
  1. pop(push(stk, i)) = stk
  2. top(push(stk,i)) = i
  3. isnew(newstack()) = true
     newstack = newstack() in caso di assenza di parametri
  4. isnew(push(stk, i)) = false
- **Specifica di restrizione**
  restrictions
  - pop(newstack) = error
  - top(newstack) = error

  dove 'error' è un elemento speciale indefinito.

Ovviamente avremmo potuto scrivere molte altre equazioni come:

$$\text{isnew(pop(push(newstack, i)))} = \text{true}$$

che evidenzia come il predicato *isnew* sia vero anche per quegli stack ottenuti inserendo un generico elemento i in un **nuovo** stack e poi rimuovendolo.
Tuttavia questa equazione è **ridondante**, poichè è ricavabile dalle altre scritte in precedenza. Infatti, grazie alla 1. e alla 4. equazione possiamo scrivere:

$$\text{isnew(pop(push(newstack, i)))} = \text{isnew(newstack)} = \text{true}$$

Nelle specifiche semantiche è importante indicare l'insieme minimale di equazioni (dette **assiomi**) a partire dalle quali possiamo derivare tutte le altre.
Le specifiche si diranno:

- **incomplete** se non permetteranno di derivare tutte le equazioni desiderate dell'algebra specificate.
- **inconsistenti** se permetteranno di derivare delle equazioni indesiderate.
- **ridondanti** se alcune delle equazioni sono ricavabili dalle altre.

Costruttori e Osservazioni Scrivere delle specifiche semantiche complete, consistenti e non ridondanti può non essere un compito semplice. Per questo conviene introdurre una **metodologia**, che si basa sulla distinzione degli operatori di un dato astratto in:

- **Costruttori**, che creano o istanziano il dato astratto
- **Osservazioni**, che ritrovano informazioni sul dato astratto

Il comportamento di una astrazione dati può essere specificata riportando il valore di ciascuna osservazione applicata a ciascun costruttore.
Questa informazione è organizzata in modo naturale nella seguente matrice:

Osservazioni	**Costruttore di stk'**	
	newstack	push(stk, i)
pop(stk')	error	stk
top(stk')	error	i
isnew(stk')	true	false

Tutte le osservazioni viste finora sono unarie, nel senso che esse osservano un singolo valore del dato astratto. Spesso è necessario disporre di osservazioni più complesse.
Per confrontare due valori è necessario osservare due istanze del dato astratto. Questo complica la specifica, perchè il valore dell'osservazione dev'essere definito per tutte le **combinazioni di costruttori** possibili per i valori astratti che si devono confrontare.

Esempio: predicato *equal(l,m)* che è vero solo se le due pile contengono gli stessi elementi nello stesso ordine. Anche in questo caso possiamo organizzare il predicato in modo naturale nella seguente matrice che può essere vista come l'aggiunta di una terza dimensione alla tabella di base per le osservazioni unarie.
Bisogna assumere and$\hat{()}$: Boolean x Boolean $\rightarrow$ Boolean.

Costruttore di m	**Costruttore di l**	
	newstack	push(stk, i)
newstack	true	false
push(stk', i')	false	i=i' and equal(stk, stk')

Questa terza dimensione può essere rimossa andando a classificare esplicitamente solo il primo argomento dell'osservazione e refereniando in modo astratto il secondo argomento mediante una variabile libera.

Osservazione	**Costruttore di stk'**	
	newstack	push(stk, i)
equal(stk', m)	isnew(m)	not isnew(m) and i=top(m) and equal(stk, pop(m))

1.11.2 Specifica algebrica di Coda

Una **coda** è una struttura dati che segue il principio '*FIFO*' (First In, First Out). Gli elementi vengono aggiunti alla fine della coda (operazione di "enqueue") e rimossi dall'inizio (operazione di "dequeue").

- **Specifica sintattica**:
  - *sorts*: queue, item, boolen
  - *operations*:
    - newq() → queue
    - addq(queue, item) → queue
    - deleteq(queue) → queue
    - frontq(queue) → item
    - isnewq(queue) → boolean

Osservazioni	**Costruttore di q'**	
	newq	addq(q, i)
isneq(q')	true	false
frontq(q')	error	if isnewq(q) then i else frontq(q)
delete(q')	error	if isnewq(q) then newq else addq(deleteq(q), i)

- **Specifica semantica**: *declare* **q**: queue, **i**: item
  1. isnewq(newq) = true
  2. isnewq(addq(q,i)) = false
  3. deleteq(addq(q,i)) = if isnewq(q) then newq else addq(deleteq(q),i)
  4. frontq(addq(q, i)) = if isnewq(q) then i else frontq(q)
- **Specifica di restrizione**
  restrictions
  - frontq(newq) = error
  - deleteq(newq) = error

1.12 Esercizi Svolti

Esercizio 1: dato astratto Stringa

Fornire una specifica algebrica semantica completa, consistente e minimale per il dato astratto Stringa supposto che le specifiche sintattiche siano le seguenti:

- sorts: string, char, integer, boolean
- operations:
  - new() → string //crea nuove stringhe
  - append(string,string) → string //concatena due stringhe
  - add(string, char) → string //aggiungere un carattere a fine stringa
  - lenght(string) → integer //calcola la lunghezza di una stringa
  - isEmpty(string) → boolean //predica se la stringa è vuota
  - equal(string, string) → boolean //predica se due stringhe sono uguali

Nello scegliere i costruttori adotteremo il **criterio di minimalità**, cioè l'insieme dei costruttori dev'essere il più piccolo insieme di operatori necessario a costruire tutti i possibili valori per un certo dato astratto. Ci poniamo 3 quesiti:

- Produce come output il sort principale?
- effettivamente crea un dato astratto di sort principale o lo modifica/distrugge?
- è atomico o è esprimibile mediante altri costruttori?

I candidati sono:

- new()
- append(string, string)
- add(string, char)

Ora, è evidente che per costruire una stringa abbiamo bisogno di:

- new() → string
- add(string, char) → string

Il terzo operatore, append(string,string) → string, non è necessario poichè semplificabile mediante add, quindi non lo scegliamo come costruttore.

Per determinare il numero di espressioni che dovremo generare secondo la specifica semantica, possiamo calcolare il prodotto tra il numero di occorrenze del sort principale come parametro e il numero di costruttori del medesimo sort principale. In questo caso avremo 2*6 espressioni.

- Specifica Semantica
  - Declare: s,s':string; c,c':char
    - length(new) = 0
    - lenght(add(s,c)) = length(s) + 1
      Assumo +(Integer, Integer) → Integer
    - isEmpty(new) = true
    - isEmpty(add(s,c)) = false
    - equal(new, new) = true
    - equal(new, add(s',c')) = false
    - equal(add(s,c), new) = false
    - equal(add(s,c), add(s',c')) = c=c' and equal(s,s')
      Assumo and(Boolean, Boolean) → Booleand
      Assumo =(Character, Character) → Boolean
    - append(new, new) = new
    - append(new, add(s'.c')) = add(s',c')
    - append(add(s,c), add(s',c')) = add(append(add(s,c), s'), c')

Esercizio 2: dato astratto Conto con Fido

Progettare il dato astratto Conto Con Fido (account with overcraft) che consente di rappresentare dei conti correnti bancari per i quali è permesso uno scoperto. Si deve poter limitare lo scoperto tramite concessione del fido.
Le operazioni ammesse per questo dato astratto sono:

- conto(account): apre un nuovo conto corrente bancario definendo saldo iniziale e massimo scoperto ammesso
- saldo(balance): riporta il saldo del conto
- deposita(deposit): deposita una somma sul conto
- preleva(withdraw): preleva denaro dal conto
- concediFido(setOverdraftLimit): definisce il limite massimo dello scoperto
- fidoConcesso(getOverdraftLimit): restituisce il limite massimo dello scoperto

A partire da queste operazioni ammesse definiamo le specifiche sintattiche:

- sorts: contocorrente, saldo, denaro, scoperto
  - conto(saldo,scoperto) → contocorrente
  - saldo(contocorrente) → saldo
  - deposita(contocorrente, denaro) → contocorrente
  - preleva(contocorrente, denaro) → contocorrente
  - concediFido(contocorrente, scoperto) → contocorrente
  - fidoConcesso(contocorrente) → scoperto

L'unico costruttore che definiamo è *conto*. Gli altri operatori sono tutte osservazioni.

Osservazioni	**Costruttore di c**
	conto(s,l)
saldo(c)	s
deposita(c,d)	conto(s+d, l)
preleva(c,d)	if s-d<-1 then error else conto(s-d, l)
concediFido(c,l')	if s<-l' then error else conto(s,l')
fidoConcesso(c)	l

Assumo:

- declare s: Saldo;l,l':scoperto; d:denaro
- +(Saldo, Denaro) → Saldo

- -(Saldo, Denaro) → Saldo
- -(Scoperto) → Scoperto
- <(Saldo, Scoperto) → Boolean

Da qui possiamo produrre le espressioni della specifica semantica:

- declare s: Saldo;l,l':scoperto; d:denaro
  - saldo(conto(s,l)) = s
  - deposita(conto(s,l), d) = conto(s+d, l)
  - preleva(conto(s,l), d) = if s-d<-l then error else conto(s-d, l)
  - concediFido(conto(s,l), l') = if s<-l' then error else conto(s,l')
  - fidoConcesso(conto(s,l)) = l

Esercizio 3: dato astratto Dizionario

Progettare il dato astratto Dizionario che consente di gestire un insieme di coppie chiave-valore.
Le operazioni ammesse per questo dato astratto sono:

- creaDizionario: restituisce un dizionario vuoto
- aggiungiCoppia: aggiunge una coppia composta da chiave e valore al dizionario
- cancella: cancella tutte le occorrenze di un dato valore, restituisce errore in assenza di occorrenze da rimuovere

Iniziamo a definire la specifica sintattica:

- sorts: Dizionario, Chiave, Valore, Integer, Boolean
  - creaDizionario() → Dizionario
  - aggiungiCoppia(Dizionario, Chiave, Valore) → Dizionario
  - cancella(Dizionario, Valore) → Dizionario
  - conta(Dizionario, Valore) → Integer

Come costruttore identifichiamo *creaDizionario* e *aggiungiCoppia*.
Passiamo alla specifica semantica:

declare d:Dizionario; k:Chiave; v,v':Valore

Osservazioni	**Costruttore di d'**	
	creaDizionario()	aggiungiCoppia(d,k,v)
cancella(d',v')	error	if(v=v') then if conta(d,v')>=1 then cancella(d,v') else d else aggiungiCoppia(cancella(d,v'),k,v)
conta(d',v')	0	if(v=v') then conta(d,v')+1 else conta(d,v')

Assumo +(Intero, Intero) → Intero

1.13 Esercizi

Esercizio 1: dato astratto Libretto

Fornire le specifiche algebriche del tipo astratto Libretto di cui si forniscono le specifiche sintattiche:

- sorts: Libretto, Esame, Voto, Boolean, Intero
- operators:
  - crea() → Libretto //crea un libretto vuoto
  - inserisci(Libretto, Esame, Voto) → Libretto //registra la coppia Esame-Voto sul libretto
  - vuoto(Libretto) → Boolean //verifica che il libretto sia vuoto
  - contieneEsame(Libretto, Esame) → Boolean //verifica se Esame è registato nel libretto
  - leggiVoto(Libretto, Esame) → Voto //restituisce il voto di Esame presente in Libretto
  - contaEsami(Libretto) → Intero //restituisce il numero di esami presenti nel libretto
  - mediaVoti(Libretto) → Voto //restituisce la media dei voti
  - contaVoti(Libretto, Voto) → Intero //conta il numero di esami registrati nel libretto con voto uguale a Voto
  - stessoVoto(Libretto, Libretto, Esame) → Boolean //verifica se nei due libretti è registrato lo stesso esame con lo stesso voto

Stabiliamo come costruttori crea e inserisci.
Procediamo con la specifica semantica:

- declare: l,l':Libretto; e,e',e":Esame; v,v':Voto
- assumo +(Intero, Intero) → Intero
- assumo +(Voto, Voto) → Voto
- assumo /(Voto, Intero) → Voto
  - vuoto(crea) = true
  - vuoto(inserisci(l,e,v)) = false
  - contieneEsame(crea,e) = false
  - contieneEsame(inserisci(l,e,v),e') = if e'=e then true else contieneEsame(l,e')
  - leggiVoto(crea,e) = error
  - leggiVoto(inserisci(l,e,v), e') = if e=e' then v else leggiVoto(l, e')

- contaEsami(crea) = 0
- contaEsami(inserisci(l,e,v)) = contaEsami(l)+1
- mediaVoti(crea) = error

Inserisco nella specifica sintattica: sommaVoti(Libretto) → Voto

- sommaVoti(crea) = error
- sommaVoti(inserisci(l,e,v)) = if vuoto(l) then v else sommaVoti(l)+v
- mediaVoti(inserisci(l,e,v)) = sommaVoti(inserisci(l,e,v))/contaEsami(inserisci(l,e,v))
- contaVoti(crea, v) = 0
- contaVoti(inserisci(l,e,v),v') = if v=v' then contaVoti(l,v')+1 else contaVoti(l,v')
- stessoVoto(crea, crea, e) = error
- stessoVoto(crea, inserisci(l,e,v),e') = error
- stessoVoto(inserisci(l,e,v), crea, e') = error
- stessoVoto(inserisci(l,e,v), inserisci(l',e',v'), e") =

```
leggiVoto(inserisci(l,e,v),e'') = leggiVoto(inserisci(l',e'v'),e'')
```

Esercizio 2: dato astratto SocialBook

Fornire le specifiche algebriche del tipo astratto SocialBook di cui si forniscono le specifiche sintattiche:

- sorts: SocialBook, Contatto, Intero, Boolean
- operators:
  - crea() → SocialBook //crea un SocialBook vuoto
  - aggiungiContatto(SocialBook,Contatto) → SocialBook //aggiunge il Contatto al SocialBook
  - aggiungiLegame(SocialBook,Contatto,Contatto) → SocialBook //crea un legame tra i due Contatti nel SocialBook
  - vuoto(SocialBook) → Boolean //verifica che il SocialBook sia vuoto ossia privo di contatti
  - contieneContatto(SocialBook,Contatto) → Boolean //verifica che il Contatto appartenda al SocialBook
  - contieneLegame(SocialBook,Contatto,Contatto) → Boolean //verifical'esistenza di un legame tra i due Contatti del SocialBook
  - contaLegami(SocialBook,Contatto) → Intero //conta i legami in cui è coinvolto Contatto nel SocialBook, se il Contattonon appartiene al SocialBook genera errore
  - rimuoviContatto(SocialBook,Contatto) → SocialBook //rimuove il Contatto dal SocialBook (qualora vi appartenga) e i suoi legami, se non appartiene lascia il SocialBookinalterato
  - rimuoviLegame(SocialBook,Contatto,Contatto) → SocialBook //rimuove il legame tra due Contatti nel SocialBook, se uno dei due Contatti non appartiene al SocialBook o non esiste un legame tra i due Contatti genera errore
  - contiene(SocialBook,SocialBook) → Boolean //restituisce vero se tutti i contatti del primo SocialBook appartengono anche al secondo SocialBook

Definiamo come costruttori *crea*, *aggiungiContatto* e *aggiungiLegame* Procediamo con la specifica semantica:

- declare: s,s':SocialBook; c,c',c",c"':Contatto
- Assumo l'esistenza dei seguenti operatori sui sort secondari:
  - and(Boolean,Boolean) → Boolean
  - or(Boolean,Boolean) → Boolean
  - +(Intero,Intero) → Intero
- vuoto(crea) = true

- vuoto(aggiungiContatto(s,c)) = false
- vuoto(aggiungiLegame(s,c,c')) = false
- contieneContatto(crea,c) = false
- contieneContatto(aggiungiContatto(s,c),c') = if c=c' then true else contieneContatto(s,c')
- contieneContatto(aggiungiLegame(s,c,c'),c") = contieneContatto(s,c") //perchè aggiungiLegame non va a variare i contatti presenti
- contieneLegame(crea,c,c') = false
- contieneLegame(aggiungiContatto(s,c),c',c") = contieneLegame(s,c',c") //perchè c appena aggiunto non può avere legami
- contieneLegame(aggiungiLegame(s,c,c'), c",c"') =

```
if((c=c'' and c'=c''')or(c=c''' and c'=c''))
then true
else contieneLegame(s,c'',c''')
```

- rimuoviContatto(crea,c) = crea
- rimuoviContatto(aggiungiContatto(s,c), c') =

```
if c=c' then s
else aggiungiContatto(rimuoviContatto(s,c'),c)
```

- rimuoviContatto(aggiungiLegame(s,c,c'),c") =

```
if(c=c'' or c'=c'') then rimuoviContatto(s,c'')
else aggiungiLegame(rimuoviContatto(s,c''),c,c')
```

- contaLegami(crea, c)= error
- contaLegami(aggiungiContatto(s,c),c') = if c=c' then 0 else contaLegami(s,c')
- contaLegami(aggiungiLegame(s,c,c'),c") =

```
if(c=c'' or c'=c'')
then contaLegami(s,c'')+1
else contaLegami(s,c'')
```

- rimuoviLegame(crea,c,c') = error

- rimuoviLegame(aggiungiContatto(s,c),c',c") =

```
if(c=c' or c=c'')
then error
else aggiungiContatto(rimuoviLegame(s,c',c''),c)
```

  oppure

```
aggiungiContatto(rimuoviLegame(s,c',c''),c)
```

- rimuoviLegame(aggiungiLegame(s,c,c'),c",c"') =

```
if((c=c'' and c'=c''')or(c=''' and c'=c''))
then s
else aggiungiLegame(rimuoviLegame(s,c'',c'''),c,c')
```

- contiene(crea,crea) = true
- contiene(crea,aggiungiContatto(s,c)) = true
- contiene(crea,aggiungiLegame(s,c,c')) = true
- contiene(aggiungiContatto(s,c),crea) = false
- contiene(aggiungiContatto(s,c), aggiungiContatto(s',c')) =

```
contieneContatto(aggiungiContatto(s',c'),c) and contiene(s,aggiungiContatto(s',c'))
```

- contiene(aggiungiContatto(s,c), aggiungiLegame(s',c',c")) = contiene(aggiungiContatto(s,c),s')
- contiene(aggiungiLegame(s,c,c'),crea) = false
- contiene(aggiungiLegame(s,c,c'),aggiungiContatto(s',c")) = contiene(s,aggiungiContatto(s',c"))
- contiene(aggiungiLegame(s,c,c'),aggiungiLegame(s',c",c"')) = contiene(s,s')

Esercizio 3: dato astratto Albero

Fornire le specifiche algebriche del tipo astratto SocialBook di cui si forniscono le specifiche sintattiche:

- sorts: Albero, Nodo, Intero, Boolean
- operators:
- +(Intero,Intero) → Intero
- and(Boolean,Boolean) → Boolean
  - crea() → Albero //crea un albero binario vuoto
  - aggiungiR(Albero, Nodo) → Albero // aggiunge la radice all'albero
  - aggiungiS(Albero, Nodo, Nodo)→ Albero // aggiunge il secondo nodo come figlio sinistro del primo nodo nell'albero
  - aggiungiD(Albero, Nodo, Nodo)→ Albero // aggiunge il secondo nodo come figlio destro del primo nodo nell'albero
  - conta(Albero) → Intero // conta il numero di nodi nell'albero
  - max(Albero) → Intero // restituisce la profondità massima dell'albero
  - pota(Albero, Nodo) → Albero // cancella il sottoalbero radicato nel nodo trasformandolo in foglia; solleva errore se il nodo è assente
  - equal(Albero, Albero) → Boolean // restituisce vero se i due alberi hanno uguale profondità massima, falso altrimenti

Definiamo come costruttori *crea*, *aggiungiR*, *aggiungiS* e *aggiungiD* Procediamo con la specifica semantica:

- declare: a,a':Albero; n,n',n",n"':Nodo
- Assumo l'esistenza dei seguenti operatori sui sort secondari:
  - +(Intero,Intero) → Intero
  - and(Boolean,Boolean) → Boolean
- conta(crea) = 0
- conta(aggiungiR(a,n)) = 1
- conta(aggiungiS(a,n,n')) = conta(a)+1
- conta(aggiungiD(a,n,n')) = conta(a)+1
- max(crea)=0
- max(aggiungiR(a,n)) = 1

Aggiungiamo nella specifica sintattica: profondita(Albero,Nodo) → Intero

- profondita(crea,n) = 0
- profondita(aggiungiR(a,n), n') = if n=n' then 1 else 0
- profondita(aggiungiS(a,n,n'), n") =

```
if n'=n''
then 1+profondita(a,n)
else profondita(a,n'')
```

- profondita(aggiungiD(a,n,n'), n") =

```
if n'=n''
then 1+profondita(a,n)
else profondita(a,n'')
```

- max(aggiungiS(a,n,n')) =

```
if(max(a)=profondita(a,n))
then max(a)+1
else max(a)
```

- max(aggiungiD(a,n,n')) =

```
if(max(a)=profondita(a,n))
then max(a)+1
else max(a)
```

- pota(crea,n) = error
- pota(aggiungiR(a,n),n') = if n=n' then aggiungiR(a,n) else error
- pota(aggiungiS(a,n,n'),n") =

```
if n'=n''
then aggiungiS(a,n,n')
else
 if n=n''
 then pota(a,n'')
 else
  if antenato(a,n'',n)
  then pota(a,n'')
  else aggiungiS(pota(a,n''),n,n')
```

- pota(aggiungiD(a,n,n'),n") =

```
if n'=n''
then aggiungiD(a,n,n')
else
 if n=n''
 then pota(a,n'')
 else
  if antenato(a,n'',n)
  then pota(a,n'')
  else aggiungiD(pota(a,n''),n,n')
```

Aggiungiamo nella specifica sintattica: antenato(Albero,Nodo,Nodo) → Boolean

- antenato(crea,n,n') = error
- antenato(aggiungiR(a,n),n',n") = error
- antenato(aggiungiS(a,n,n'),n",n"') =

```
if n=n'' and n'=n'''
then true
else
 if(appartiene(a,n'') and appartiene(a,n'''))
 then antenatoBis(a,n'', padre(a,n'''))
 else error
```

- antenato(aggiungiD(a,n,n'),n",n"') =

```
if n=n'' and n'=n'''
then true
else
 if(appartiene(a,n'') and appartiene(a,n'''))
 then antenatoBis(a,n'', padre(a,n'''))
 else error
```

Aggiungiamo nella specifica sintattica: appartiene(Albero,Nodo) → Boolean

- appartiene(crea(), n) = false
- appartiene(aggiungiR(a,n),n')= n=n'
- appartine(aggiungiS(a,n,n'),n")=if n'=n" then true else appartiene(a,n")
- appartine(aggiungiD(a,n,n'),n")=if n'=n" then true else appartiene(a,n")

Aggiungiamo nella specifica sintattica: antenatoBis(Albero,Nodo,Nodo) → Boolean

- antenatoBis(crea(),n,n') = false
- antenatoBis(aggiungiR(a,n),n',n") = false
- antenatoBis(aggiungiS(a,n,n'),n",n"') =

```
if n=n'' and n'=n'''
then true
else antenatoBis(a,n'', padre(a,n'''))
```

- antenatoBis(aggiungiD(a,n,n'),n",n"') =

```
if n=n'' and n'=n'''
then true
else antenatoBis(a,n'', padre(a,n'''))
```

Aggiungiamo nella specifica sintattica: padre(Albero,Nodo) → Boolean

- padre(crea,n,n') = error
- padre(aggiungiR(a, n), n', n") = error
- padre(aggiungiS(a, n, n'), n", n"') =

```
if(n=n'' and n'=n''')
then true
else padre(a, n'', n''')
```

- padre(aggiungiD(a, n, n'), n", n"') =

```
if(n=n'' and n'=n''')
then true
else padre(a, n'', n''')
```

- equal(crea,a)= (max(a)=0)
- equal(aggiungiR(a,n),a') = (max(a')=1)
- equal(aggiungiS(a,n,n'),a') = (max(aggiungiS(a,n,n')) = max(a'))
- equal(aggiungiD(a,n,n'),a') = (max(aggiungiD(a,n,n')) = max(a'))

Esercizio 4: dato astratto Matrice

Fornire le specifiche algebriche del tipo astratto Matrice di cui si forniscono le specifiche sintattiche:

- sorts: Matrice, Intero, Boolean, Reale
- operators:
  - creaMatrice(Intero) → Matrice // crea una matrice quadrata di reali (inizialmente pari a zero), la dimensione della matrice è specificata come parametro
  - assegna(Matrice, Intero, Intero, Reale) → Matrice // assegna in valore specificato (quarto parametro) alla posizione della matrice (primo parametro) con indice di riga e colonna specificati come secondo e terzo parametro, rispettivamente
  - leggi(Matrice, Intero, Intero) → Reale // legge il valore nella posizione della matrice con indice di riga e colonna specificati come secondo e terzo parametro, rispettivamente
  - max(Matrice) → Reale // restituisce il valore massimo attualmente memorizzato nella matrice
  - somma(Matrice, Matrice) → Matrice // restituisce la matrice computata tramite somma, solleva errore se le due matrici non hanno uguale dimensione
  - uguale(Matrice, Matrice) → Booleano // vero se le due matrici contengono uguali valori nelle medesimi posizioni, falso altrimenti (solleva errore se le due matrici non hanno uguale dimensione)

Definiamo come costruttori gli operatori *creaMatrice* e *assegna*.
Passiamo alla specifica semantica:

- declare: m,m':Matrice; r,r',c,c',d,d':Intero; v,v':Reale
- Assumo che siano definite le seguenti operazioni:
  - $\leq$(Intero,Intero) → Boolean
  - <(Intero,Intero) → Boolean
  - and(Boolean,Boolean) → Boolean
  - >(Reale,Reale) → Boolean
  - +(Reale,Reale) → Reale
- leggi(creaMatrice(d),r,c) =

```
if r<=d and c<=d and r>0 and c>0
then 0
else error
```

- leggi(assegna(m,r,c,v),r',c') =

```
if r=r' and c=c'
then v
else leggi(m,r',c')
```

- max(creaMatrice(d)) = 0.0
- max(assegna(m,r,c,v)) = if max(m) > v then max(m) else v
- somma(creaMatrice(d), creaMatrice(d')) =

```
if d = d'
then creaMatrice(d)
else error
```

- somma(creaMatrice(d), assegna(m,r,c,v)) =

```
if d = dimensione(m)
then assegna(m,r,c,v)
else error
```

- somma(assegna(m,r,c,v), creaMatrice(d)) =

```
if d = dimensione(m)
then assegna(m,r,c,v)
else error
```

- somma(assegna(m,r,c,v), assegna(m',r',c',v')) =

```
if dimensione(m) =dimensione(m')
then if (r=r' and c=c')
  then assegna(somma(m,m'),r,c,v+v')
  else modifica(somma(assegna(m,r,c,v),m'),r',c',v'+leggi(m,r,c))
else error
```

Aggiungiamo nella specifica sintattica: dimensione(Matrice) → Intero

- dimensione(creaMatrice(d)) = d
- dimensione(assegna(m,r,c,v)) = dimensione(m)

Aggiungiamo nella specifica sintattica: modifica(Matrice,Intero,Intero,Reale) → Matrice

- modifica(creaMatrice(d),r,c,v) =

```
if r<=d and c<=d and r>0 and c>0
then assegna(creaMatrice(d),r,c,v)
else error
```

- modifica(assegna(m,r,c,v),r',c',v') =

```
if r=r' and c=c'
then assegna(m,r,c,v')
else assegna(modifica(m,r',c',v'),r,c,v)
```

- uguale(creaMatrice(d), creaMatrice(d')) = if d = d' then true else error
- uguale(creaMatrice(d), assegna(m,r,c,v)) =

```
if d = dimensione(m)
then else (m'=crea(d) and v=0)
else error
```

- uguale(assegna(m,r,c,v), creaMatrice(d')) =

```
if d' = dimensione(m)
then (m=crea(d') and v=0)
else error
```

- uguale(assegna(m,r,c,v), assegna(m',r',c',v')) =

```
if (dimensione(m) != dimensione(m'))
then error
else
 if (r=r' and c=c') then
  if (v=v')
  then uguale(m,m')
  else false
 else
  if(leggi(m,r',c')=v' and leggi(m',r,c)=v)
  then uguale(modifica(m,r',c',0),modifica(m',r,c,0))
  else false
```

2 Astrazione nella programmazione

2.1 Introduzione

Negli anni '50 e '60 si verificarono importanti sviluppi nei linguaggi di programmazione che introdussero diverse forme di astrazione. Queste astrazioni comprendevano la creazione di strutture di controllo, operatori e dati, che consentivano ai programmatori di gestire in modo più efficiente e chiaro la complessità dei loro programmi. In questo contesto, si consolidò la convinzione che fosse possibile costruire astrazioni su una vasta gamma di costrutti sintattici, a condizione che questi costrutti specificassero un qualche tipo di computazione. Nel corso di questo capitolo, esamineremo l'applicazione di questo principio di astrazione a sei classi sintattiche fondamentali:

- **Espressione** → Astrazione di funzione: Un'espressione rappresenta un calcolo che restituisce un valore. L'astrazione di funzione consiste nell'organizzare e incapsulare un insieme di operazioni in una singola entità, la funzione, che può essere chiamata con uno o più argomenti per ottenere un risultato specifico.
- **Comando** → Astrazione di procedura: Un comando rappresenta un'azione che deve essere eseguita. L'astrazione di procedura consiste nell'incapsulare una sequenza di comandi in una procedura, che può essere chiamata per eseguire l'insieme di operazioni definite.
- **Controllo di sequenza** → Astrazione di controllo: Il controllo di sequenza determina l'ordine in cui vengono eseguite le istruzioni. L'astrazione di controllo coinvolge la creazione di strutture di controllo più complesse, come cicli e condizioni, che consentono di gestire il flusso di esecuzione del programma in modo più flessibile e dinamico.
- **Accesso a un'area di memoria** → Astrazione di selettore: L'accesso a un'area di memoria implica l'individuazione e la manipolazione di dati memorizzati in variabili o strutture dati. L'astrazione di selettore coinvolge la creazione di meccanismi per accedere e manipolare queste aree di memoria in modo strutturato e controllato.
- **Definizione di un dato** → Astrazione di tipo: La definizione di un dato specifica la struttura e il comportamento di un tipo di dato. L'astrazione di tipo coinvolge la definizione di nuovi tipi di dato e le operazioni associate a essi, consentendo di organizzare e manipolare i dati in modi significativi e consistenti.
- **Dichiarazione** → Astrazione generica: Una dichiarazione introduce un'entità nel contesto del programma, come una variabile, una costante o una funzione. L'astrazione generica coinvolge la creazione di meccanismi per associare un nome a un'entità e specificarne le proprietà e il comportamento, contribuendo così alla gestione e all'organizzazione del codice.

È importante notare che tutte queste classi sintattiche sottintendono una forma di computazione, in quanto ogni astrazione implica l'esecuzione di operazioni o l'elaborazione di dati per ottenere un risultato. Tuttavia, esiste un controesempio: la classe sintattica "letterale", che non può essere astratta poiché rappresenta semplicemente un valore senza specificare alcuna operazione o computazione associata.

2.2 Astrazione di funzione

Un'**astrazione di funzione** rappresenta un concetto fondamentale nella programmazione, permettendo di organizzare e riutilizzare logiche di calcolo complesse in maniera efficiente e chiara. Questa astrazione è formalizzata attraverso la definizione di una funzione, che specifica come una determinata espressione deve essere valutata in base a un insieme di parametri forniti.
La definizione di una funzione segue uno schema ben definito, del tipo:

$$\text{function } I(FP1, ..., FPn) \text{ is } E$$

dove:

- I è l'identificatore della funzione;
- $FP1, ..., FPn$ sono i parametri formali della funzione, ossia i valori che devono essere forniti quando la funzione viene chiamata;
- E è l'espressione da valutare, che rappresenta l'algoritmo o la logica di calcolo che la funzione deve eseguire.

In questo modo, si crea un legame tra l'identificatore della funzione e un'entità astratta che, una volta chiamata con i parametri appropriati, restituisce un valore come risultato.
Una chiamata di funzione, del tipo $I(AP1, ..., APn)$, in cui $AP1, ..., APn$ sono i parametri effettivi che determinano gli argomenti, può essere considerata da due punti di vista:

- Dal punto di vista dell'utente, la chiamata di una funzione trasforma gli argomenti forniti in un risultato utile per il contesto del programma.
- Dal punto di vista dell'implementatore, la chiamata valuta l'espressione E, avendo precedentemente associato i parametri formali agli argomenti forniti.

L'algoritmo codificato nell'espressione E è di particolare interesse per l'implementatore, poiché rappresenta la logica di calcolo specifica che viene eseguita quando la funzione viene chiamata. Questo algoritmo può essere complesso o semplice a seconda delle necessità, ma il suo corretto funzionamento è fondamentale per garantire il comportamento desiderato della funzione nel contesto del programma.

Le funzioni, come astrazioni di espressioni, possono comparire ovunque si richieda un'espressione. Pertanto, possono essere utilizzate alla destra di operazioni di assegnazione, ma anche come parametri effettivi nelle chiamate di altre funzioni o procedure, dove un valore può essere calcolato mediante un'espressione. Ad esempio:

$$y := f(y, \text{power}(x, 2))$$

Qui, la funzione f ha due parametri passati per valore.

In molti linguaggi di programmazione è possibile definire dei parametri formali che sono un riferimento a una funzione. Ad esempio, in Pascal:

```
function Sommatoria(F:function(R:real,M:integer):real;X:real;N:integer):real;
var
 I: integer;
 Sum: real;
begin
 Sum := 0;
 for I := 1 to N do
  Sum := Sum + F(X, I);
 Sommatoria := Sum;
end;
```

Qui, la funzione `Sommatoria` prende come primo parametro una funzione F, e la utilizza per calcolare una sommatoria.

In alcuni linguaggi di programmazione, il concetto di astrazione di funzione è separato dal concetto di binding a un identificatore. Ciò permette di passare come parametri delle funzioni anonime, prive di identificatore.

Infine, le funzioni possono essere il risultato di valutazioni di espressioni o possono essere assegnate a variabili. Ad esempio, in ML:

$$\text{val cube} = \text{fn}(x : \text{real}) \rightarrow x \times x \times x$$

Qui, è definita una funzione anonima, `cube`, che calcola il cubo di un numero.

2.2.1 Anomalie di progetto dei linguaggi

Se un'astrazione di funzione include un'espressione E da valutare, ci si aspetterebbe naturalmente che il corpo della funzione non contenga comandi che modificano lo stato di un sistema, come assegnazioni, istruzioni di salto o iterazioni, poiché il ruolo principale di una funzione è quello di produrre valori.
Tuttavia, non sempre ciò avviene, come evidenziato in questo esempio di funzione scritta in Pascal:

```
function power(x: real; n: integer): real
begin
 if n = 1 then
  power := x
 else
  power := x * power(x, n - 1)
end
```

Nel corpo della funzione Pascal compaiono comandi di assegnazione. Questo avviene perché in Pascal, per restituire un valore, è necessario assegnare quel valore a una pseudo-variabile con lo stesso nome della funzione. In questo esempio, l'identificatore della funzione, `power`, funge sia da espressione da valutare che da pseudo-variabile dove verrà depositato il risultato. Anche se il corpo della funzione è sintatticamente un comando, semanticamente è considerato un'espressione, poiché la funzione può essere invocata solo alla destra di operazioni di assegnazione.

È possibile evitare comandi nel corpo delle funzioni. Ad esempio, nel linguaggio funzionale ML:

```
function power(x: real; n: int) is
 if n = 1 then x
 else x * power(x, n - 1)
```

Nel corpo della funzione definita in ML compare una semplice espressione condizionale, la cui valutazione non modifica lo stato del sistema.
Tuttavia, linguaggi come Pascal, Ada e altri permettono comandi nel corpo delle funzioni per sfruttare la potenza espressiva dell'assegnazione e dell'iterazione nella computazione dei risultati. Altrimenti, saremmo costretti a esprimere le espressioni da valutare in modo ricorsivo, il che può essere inefficiente nell'uso delle risorse di calcolo.

In conclusione, molti linguaggi consentono comandi nel corpo delle funzioni per motivi di efficienza, ma è compito del programmatore utilizzarli correttamente per evitare effetti collaterali oltre a quelli previsti per il calcolo del valore.

2.2.2 Riepilogo

In alcuni linguaggi di programmazione, come Fortran e Ada-83, le funzioni sono considerate di terza classe, il che significa che possono essere solo chiamate e non possono essere utilizzate in altri contesti.

In altri linguaggi, come Pascal, le funzioni sono considerate di seconda classe, il che significa che possono essere passate come argomenti a altre funzioni o procedure, ma non possono essere restituite come risultato di una chiamata di

funzione o assegnate come valore a una variabile.

Tuttavia, in alcuni linguaggi di programmazione avanzati, come Lisp, ML e alcuni linguaggi di scripting come Perl, le funzioni sono considerate di prima classe. Questo significa che possono essere trattate allo stesso modo delle altre entità del linguaggio, come numeri o stringhe. Possono essere restituite come risultato di una chiamata di funzione, assegnate come valore a una variabile, e persino generate al runtime, permettendo un'elevata flessibilità e capacità di astrazione nel codice.

La differenza tra i diversi livelli di "classe" delle funzioni nei linguaggi di programmazione riflette le diverse filosofie di design e le esigenze dei programmatori in termini di espressività e flessibilità nel linguaggio.

2.3 Cittadini di prima classe

In programmazione, un'entità è considerata cittadino di prima classe quando gode di piena libertà nel suo utilizzo, senza essere soggetta a restrizioni particolari. Ciò significa che l'entità può essere trattata allo stesso modo di altre entità del linguaggio, come numeri o stringhe, e può essere manipolata in modo flessibile e dinamico all'interno del programma. Questa libertà di utilizzo fornisce ai programmatori una maggiore espressività e capacità di astrazione nel codice, consentendo di scrivere software più conciso, modulare ed efficiente.

A tal proposito, i valori possono essere classificati in base a diverse proprietà:

- **Denotabili**: I valori sono denotabili se possono essere associati ad un nome, consentendo ai programmatori di fare riferimento a essi all'interno del codice.
- **Esprimibili**: I valori sono esprimibili se possono essere il risultato di un'espressione complessa, diversa da un semplice nome. Questo significa che possono essere calcolati o generati mediante operazioni o combinazioni di altri valori.
- **Memorizzabili**: I valori sono memorizzabili se possono essere assegnati e conservati in una variabile, consentendo ai programmatori di accedervi e modificarli durante l'esecuzione del programma.

Ad esempio, nei linguaggi imperativi, i valori di tipo intero sono generalmente denotabili, esprimibili e memorizzabili.
Al contrario, i valori del tipo delle funzioni da Integer a Integer sono denotabili in quasi tutti i linguaggi, perché possiamo dare loro un nome con una dichiarazione:

int succ(int x) return x+1

ma non sono esprimibili o memorizzabili nei linguaggi imperativi, poiché non possono essere restituiti come risultato di un'espressione complessa o assegnati

a una variabile.

La situazione è diversa nei linguaggi funzionali, come Scheme, ML e Haskell, dove i valori funzionali sono denotabili, esprimibili e, in certi linguaggi, memorizzabili. Stessa cosa dicasi per gli oggetti ai quali si accennerà in seguito. Nei linguaggi imperativi essi sono denotabili ma non esprimibili e memorizzabili. In altri termini, gli oggetti non sono cittadini di prima classe. Al contrario, nei linguaggi orientati agli oggetti, gli oggetti sono denotabili, esprimibili e memorizzabili, cioè sono cittadini di prima classe.

2.4 Astrazione di procedura

Un'astrazione di procedura include un comando da eseguire e, quando chiamata, aggiorna le variabili che rappresentano lo stato del sistema. È specificata mediante una definizione di procedura, del tipo:

$$\text{procedure } I(FP_1; \ldots; FP_n) \text{ is } C$$

dove:

- I è l'identificatore della procedura;
- $FP_1; \ldots; FP_n$ sono i parametri formali della procedura, ossia i valori che devono essere forniti quando la procedura viene chiamata;
- C è il blocco di comandi da eseguire, che rappresenta l'algoritmo o la logica di calcolo che la procedura deve eseguire.

In questo modo, si lega I all'astrazione di procedura, che cambia lo stato del sistema quando chiamata con argomenti appropriati.

Data una chiamata di procedura $I(AP_1; \ldots; AP_n)$ dove $AP_1; \ldots; AP_n$ sono i parametri effettivi, il punto di vista dell'utente è che la chiamata aggiorni lo stato del sistema in modo dipendente dai parametri, mentre il punto di vista dell'implementatore è che la chiamata consenta l'esecuzione del corpo di procedura C, avendo precedentemente vincolato i parametri formali agli argomenti corrispondenti. L'algoritmo codificato in C è di interesse solo per l'implementatore.
In linguaggi come Pascal e C, le procedure sono considerate cittadini di seconda classe.

Esempio di puntatori a funzione in C

La funzione `bubble` ordina un array di interi sulla base di una funzione di ordinamento. L'argomento con puntatore a funzione:

```
void bubble(int a[], int n, int (*compare)(int, int, int *))
```

indica a `bubble` di aspettarsi un puntatore a una funzione, identificata da `compare`, che accetti tre argomenti in ingresso e restituisca un tipo `void`.

Nota: Se avessimo rimosso le parentesi:

```
void *compare(int, int, int *)
```

avremmo dichiarato semplicemente una funzione che accetta tre interi in input e restituisce un puntatore a `void`. Invece, con le parentesi, stiamo dichiarando un puntatore a funzione.

```
#include <stdio.h>
#define SIZE 10
void bubble(int [], const int, void (*)(int,int,int*));
//si noti il parametro di tipo puntatore a funzione
void ascending(int,int,int*);
void descending(int,int,int*);

int main(){
    int order,
    counter,
    a[SIZE] = {2,6,4,8,10,12,89,68,45,37};

    printf("Enter 1 to sort in ascending order,\n"
    "Enter 2 to sort in descending order: ");

    scanf("%d",&order);

    printf("\nData items in original order\n");

    for(counter=0;counter<SIZE;counter++){
        printf("%5d",a[counter]);
    }

    if(order==1){
        bubble(a,SIZE,ascending);
        printf("\nData items in ascending order\n");
    }else{
        bubble(a,SIZE,descending);
        printf("\nData items in descending order\n");
    }

    for(counter=0;counter<SIZE;counter++){
        printf("%5d",a[counter]);
    }

    printf("\n");
```

```
    return 0;
}

void bubble(int work[],const int size, void (*compare)(int,int,int*)){
    int pass,
    count,
    bool;

    void swap(int*,int*);

    for(pass=1;pass<size;pass++){
        for(count=0;count<size-1;count++){
            (*compare)(work[count],work[count+1],&bool);
            //ascending e descending restituiscono in bool il risultato
            //che puo' essere 0 o 1.

            //bubble chiama swap se bool e' 1.

            //nota come i puntatori a funzione vengano chiamati
            // con l'operatore di deferenziazione (*),
            //che, seppur non sia strettamente necessario,
            //enfatizza come compare sia un puntatore a funzione
            //e non una funzione vera e propria.
            if(bool){
                swap(&work[count],&work[count+1]);
            }
        }
    }
}

void swap(int *element1Ptr,int *element2Ptr){
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}

void ascending(int a,int b, int *c){
    *c = a > b;
}

void descending(int a,int b, int *c){
    *c = a < b;
}
```

2.5 Astrazione Funzionale

L'astrazione di funzione o di procedura sono due tecniche di programmazione di supporto all'astrazione funzionale, intesa come tecnica di progettazione del software secondo la quale occorre distinguere la specifica di un operatore (come esso è visto e manipolato dall'utente) dalla sua realizzazione.

La scelta della tecnica di programmazione più opportuna da adottare dipende da diversi fattori:

- Il tipo di operatore progettato:
  - Se l'operatore ha effetti collaterali, come modificare lo stato del sistema, è più opportuno utilizzare l'astrazione di procedura.
  - Se l'operatore non ha effetti collaterali e restituisce un risultato senza modificare lo stato del sistema, è preferibile utilizzare l'astrazione di funzione.
- I limiti imposti dal linguaggio di programmazione:
  - Ad esempio, se l'operatore bubblesort non modifica l'array passato in ingresso ma ne restituisce uno ordinato, sarebbe più opportuno implementarlo ricorrendo a un'astrazione di funzione, poiché restituisce un nuovo valore che è un intero array ordinato.
  - Tuttavia, non tutti i linguaggi di programmazione permettono di definire funzioni che restituiscono dati complessi. In alcuni linguaggi, come Pascal, gli array sono cittadini di seconda classe e in tal caso, l'operatore bubblesort dovrà essere necessariamente realizzato mediante astrazione di procedura.

2.6 Astrazione di controllo

L'astrazione di controllo si applica alla classe sintattica delle strutture di controllo. Queste definiscono l'ordine in cui le singole istruzioni o gruppi di istruzioni (unità di programma) devono essere eseguiti.

Nei linguaggi di basso livello, come i linguaggi assemblativi, le istruzioni sono eseguite in sequenza e i salti sono implementati tramite istruzioni di salto, ad esempio

jump to <indirizzo simbolico o label>

Tuttavia, l'indirizzo simbolico è di scarsa importanza per il programmatore; ciò che conta è poter indicare le prossime istruzioni da eseguire.

Per questa ragione i linguaggi di alto livello hanno introdotto strutture di controllo astratte per semplificare la programmazione.

Ad esempio, l'istruzione

if cond then S1 else S2

rappresenta una selezione astratta, mentre l'istruzione

jump on <cond> to A; S2; jump to B; A: S1; B: ...

rappresenta un salto astratto.

Similmente, sono state introdotte diverse strutture di controllo astratte per l'iterazione. Inoltre, l'utilizzo dello stack per conservare gli indirizzi di ritorno dalle chiamate di funzione/procedura ha reso possibili le chiamate ricorsive.

Di particolare interesse è l'attuale tendenza a offrire strutture di controllo iterative per collezioni omogenee di valori, come insiemi, multiinsiemi, liste e array. Ad esempio, in Java è disponibile l'astrazione di controllo "for-each", che permette di iterare su una collezione.

```
LinkedList<Integer> list = new LinkedList<Integer>();
... /* si inseriscono degli elementi
for (Integer n : list) {
 System.out.println(n);
}
```

L'operazione messa a disposizione di LinkedList per l'astrazione di controllo for-each è iterator().

Se consideriamo le strutture di controllo come espressioni che definiscono l'ordine di esecuzione dei comandi, possiamo specificare un'astrazione di controllo come segue:

$$\text{control } I(FP_1; \ldots; FP_n) \text{ is } S$$

dove:

- I è l'identificatore di una nuova struttura di controllo;
- $FP_1; \ldots; FP_n$ sono i parametri formali;
- S è un'espressione di controllo che definisce l'ordine di esecuzione.

Esempio: (in un ipotetico linguaggio di programmazione)

```
control swap(boolean:cond, statement:S1,S2) is
//argomento di tipo espressione, statement
 if cond then
  begin S1;S2 end
 else
  S2;S1
 endif
```

La chiamata swap(i¡j, i:=j, j:=i) porta i al valore di j se i è minore di j, altrimenti porta j al valore di i.

Esempio: (in un ipotetico linguaggio di programmazione)

```
control alt_execution(statement: S1,S2,S3) is
\\argomento di tipo statement
 if abnormal (S1) then
  S2
 else
  S3
 endif
```

abnormal è un predicato che ha in input un comando S1 e restituisce true se l'esecuzione di S1 è terminata in modo anomalo (è stata sollevata un'eccezione), false altrimenti.

I linguaggi di programmazione moderni sono dotati di meccanismi sofisticati per gestire situazioni eccezionali, tra cui errori aritmetici, errori di I/O, fallimento di precondizioni e condizioni imprevedibili. Quando si verifica un'eccezione, è necessario catturarla e gestirla in modo appropriato.

I metodi per individuare il gestore dell'eccezione possono includere la ricerca nel blocco corrente di codice, l'uscita dall'unità corrente o la navigazione verso l'alto nella gerarchia delle unità chiamanti. Una volta individuato il gestore e gestita l'eccezione, è possibile scegliere di ripartire dall'unità contenente il gestore (nel modello di terminazione, come in Ada) o di tornare all'esecuzione del comando che ha generato l'eccezione (nel modello di ripresa).

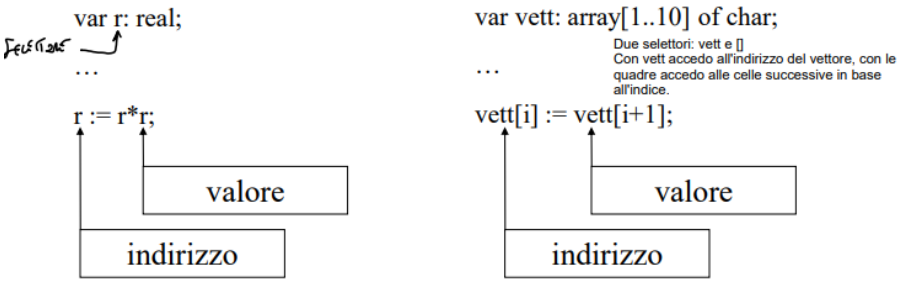
Nei linguaggi di programmazione sequenziale, la sequenza, la selezione e la ripetizione stabiliscono un ordinamento totale sulla sequenza di esecuzione dei comandi. Nei linguaggi paralleli, vengono impiegati ulteriori costrutti di controllo del flusso, come fork, cobegin e cicli for paralleli, per introdurre un ordinamento parziale sulla sequenza di esecuzione dei comandi. Dato che il parallelismo costituisce una forma di controllo della sequenza di esecuzione, l'astrazione di controllo assume una particolare importanza nella programmazione parallela.

2.7 Astrazione di selettore

Un linguaggio di programmazione dispone di costrutti per poter accedere in memoria ad una variabile (strutturata e non).
Il selettore variabile può essere utilizzato sia in espressioni che in comandi. Per esempio, in un assegnamento, quando è a sinistra è veramente un selettore perchè deve accedere ad un'area di memoria per scrivere il dato preso. Quando è a destra è un selettore perchè deve accedere ad un'area di memoria per leggere

il dato. Lo stesso identificatore può essere usato sia come nome di un valore (legame dinamico fra identificatore e valore) e sia come un indirizzo (legame statico fra identificatore e locazione di memoria).

Ad esempio in Pascal abbiamo:



(a)

```
type rec= record
            a: real;
            b:real
          end
```

Campi di record sono selettori

```
var r: rec;
...
r.b := r.a*3;
```

(b)

Figure 4: Due immagini affiancate.

Nel caso dell'array dell'immagine a abbiamo due selettori: la variabile vett che è un selettore classico che mi permette di accedere in memoria alla posizione dell'array; le parentesi quadre che mi permettono di accedere in memoria al valore dell'array in una determinata cella dipesa dall'indice di posizione.
Nel caso del record dell'immagine b abbiamo i campi dei record che sono selettori e che costituiscono un riferimento ad una variabile.

Tuttavia, se osserviamo il seguente codice Pascal:

```
type queue = ...;

var Aq: queue;

function first(q: queue): integer
```

```
  ...(* Restituisce il primo intero della coda *)

 i := first(Aq);
```

Ci accorgiamo che la chiamata a `first(Aq)` può comparire solo alla destra di un'assegnazione, perché le funzioni restituiscono valori, mentre a sinistra:

```
 first(Aq) := 0;
```

dovrebbe restituire riferimenti ad aree di memoria.

In Pascal, abbiamo dei selettori predefiniti dal progettista del linguaggio:

- `F^`: riferimento a un puntatore `F`
- `V[E]`: riferimento a un elemento di un array `V`
- `R.A`: riferimento a un elemento di un record

Tuttavia, il programmatore non ha modo di definire un nuovo selettore, come il riferimento a una lista di elementi indipendentemente da come la lista è realizzata. In altri termini, non c'è la possibilità di definire astrazioni di selettore che restituiscano l'accesso a un'area di memoria.

Per poter scrivere un'assegnazione del tipo:

```
 first(Aq) := 0;
```

dovremmo poter definire un tipo di astrazione che, quando chiamata, restituisce il riferimento a una variabile (astrazione di selettore). Supponiamo di estendere il Pascal con le astrazioni di selettore nel seguente modo:

```
 selector I(FP1; ...; FPn) is A
```

dove `A` è un'espressione che restituisce un accesso a una variabile (che denoteremo con & come in C).

Potremmo allora definire `first` come segue:

```
 type
  queue = record
   elementi: array[1..max] of integer;
   testa, fondo, lung: 0..max;
  end;

 selector first(q: queue) is &(q.elementi[q.testa]);
```

Questo ci consentirebbe di scrivere espressioni come:

```
first(Aq) := first(Aq) + 1;
```

dove l'invocazione di destra si riferisce alla funzione, mentre quella di sinistra si riferisce all'astrazione di selettore.

Esempio: Coda

```
class Queue{
 \dots
 public:
  static int & first(Queue*);
};
Queue q*;
\dots
int i=first(q);
first(q)*=first(q);
first(q)++;
```

dove & è l'operatore di indirizzamento, obbligatorio per rendere la funzione un selettore.

2.8 Flessibilità dei linguaggi

La potenza espressiva di un linguaggio di programmazione è determinata principalmente da tre fattori fondamentali: i meccanismi di composizione, i meccanismi di controllo di sequenza e la gestione dei dati.

I meccanismi di composizione si riferiscono alla capacità del linguaggio di definire operazioni complesse combinando quelle più semplici. Linguaggi che supportano l'astrazione funzionale offrono una grande flessibilità in questo ambito, consentendo ai programmatori di creare strutture complesse e modulari attraverso l'uso di funzioni e procedure.

I meccanismi di controllo di sequenza riguardano la capacità di stabilire l'ordine di esecuzione delle istruzioni nel programma. Linguaggi che supportano l'astrazione di controllo forniscono agli sviluppatori una vasta gamma di costrutti per gestire il flusso del programma in modo chiaro e intuitivo.

Infine, la gestione dei dati si riferisce alla capacità del linguaggio di rappresentare e manipolare i dati in modi diversi. Linguaggi che supportano l'astrazione dei dati consentono ai programmatori di definire nuovi tipi di dati e operazioni su di essi, offrendo così una maggiore flessibilità nell'organizzazione e nella manipolazione delle informazioni.

2.9 Tecniche di programmazione a supporto dell'astrazione dati

Le tecniche di programmazione che supportano l'astrazione dei dati nel paradigma imperativo, fondamentali per la progettazione del software, possono essere suddivise in due approcci principali:

1. **Definizione di Tipi Astratti**: Questo approccio consiste nell'astrazione della classe sintattica del tipo, cioè la creazione di tipi di dati personalizzati che nascondono la loro implementazione interna. Gli utenti possono interagire con questi tipi di dati attraverso un insieme di operazioni definite, senza avere conoscenza diretta della rappresentazione interna dei dati. Inoltre, i tipi astratti rendono visibili sia l'identificatore di tipo che gli operatori associati.

2. **Definizione di Classi di Oggetti**: In questo approccio, l'astrazione si basa sulla creazione di moduli dotati di stato locale, noti come classi di oggetti. Le classi di oggetti consentono di definire oggetti che incapsulano sia i dati che le operazioni che possono essere eseguite su di essi. Tuttavia, a differenza dei tipi astratti, le classi di oggetti rendono visibili solo gli operatori associati, nascondendo l'identificatore di tipo agli utenti.

In entrambi i casi, è fondamentale poter incapsulare la rappresentazione dei dati con le operazioni legittime, garantendo così un'interfaccia coerente e sicura per l'accesso e la manipolazione dei dati.

2.10 Tipo concreto e tipo astratto

Nei linguaggi di programmazione ad alto livello, viene fornito al programmatore un insieme diversificato di tipi predefiniti, noti come tipi concreti. Questi tipi possono essere suddivisi in due categorie principali: primitivi o semplici, i cui valori sono atomici e non possono essere ulteriormente scomposti, e composti o strutturati, i cui valori sono ottenuti dalla combinazione di valori più semplici.

Tuttavia, l'espressività di un linguaggio di programmazione dipende anche dalla sua capacità di consentire al programmatore di definire i propri tipi di dati personalizzati a partire dai tipi di dato concreti disponibili. Questi tipi definiti dall'utente, noti anche come user defined types (UDT), sono considerati astratti in quanto nascondono i dettagli implementativi sottostanti e forniscono un'interfaccia chiara e coesa per l'interazione con i dati.

2.11 Astrazione di tipo

L'espressione di tipo (spesso abbreviato con tipo) è il costrutto con cui alcuni linguaggi di programmazione consentono di definire un nuovo tipo.
Prendiamo ad esempio il linguaggio Pascal, dove possiamo creare un nuovo tipo chiamato "Person" attraverso la seguente dichiarazione:

```
type Person = record
name: packed array[1..20] of char;
age: integer;
height: real
end
```

In questa dichiarazione, definiamo esplicitamente una rappresentazione per i valori del tipo "Person", specificando i campi "name", "age", e "height". Tuttavia, gli operatori associati a questo tipo saranno necessariamente generici, come ad esempio l'assegnazione. Non abbiamo la possibilità di definire operatori specifici per il tipo "Person".

D'altra parte, l'astrazione di tipo, o tipo astratto di dato, ci permette di definire sia una rappresentazione per un insieme di valori, sia le operazioni applicabili a questi valori. Questo concetto può essere espresso attraverso una dichiarazione del tipo:

type I(FP1; ...; FPn) is T

dove:

- "I" è l'identificatore del nuovo tipo;
- "FP1; ...; FPn" sono i parametri formali;
- "T" è un'espressione di tipo che specifica la rappresentazione dei dati di tipo "I" e le operazioni ad esso applicabili.

Consideriamo ad esempio l'astrazione di tipo definita per un tipo "complex":

```
type complex = record
Re: real;
Im: real
end
```

Qui stabiliamo che "complex" è un identificatore di tipo, e associamo una rappresentazione attraverso tipi concreti già disponibili nel linguaggio. Le operazioni associate al tipo "complex" sono quelle definite per il tipo "record", come l'assegnazione e la selezione di campi.

Tuttavia, questa astrazione di tipo ha dei limiti evidenti:

1. Il programmatore non può definire nuovi operatori specifici da associare al tipo.
2. Si viola il requisito di protezione, in quanto l'utilizzatore è consapevole della rappresentazione del tipo "complex" e può operare su di esso con operatori non specifici del dato.
3. L'astrazione di tipo non è parametrizzata, quindi la comunicazione con il contesto esterno è limitata.

2.12 Package

Per superare questi limiti, possiamo introdurre un costrutto di programmazione che permette di incapsulare rappresentazioni del dato e operatori leciti: il package. Un package è un gruppo di componenti dichiarate, come tipi, costanti, variabili, funzioni e persino (sotto) moduli.
Introducendo un costrutto package, possiamo migliorare notevolmente questo concetto, consentendo una maggiore modularità, incapsulamento e flessibilità nel definire nuovi tipi di dati e le operazioni ad essi associate.

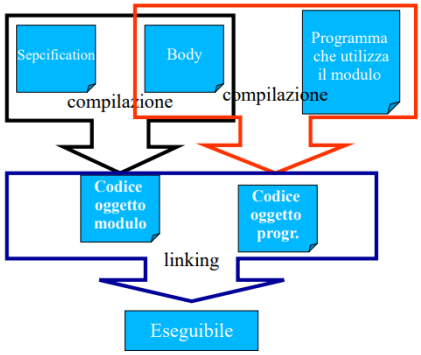


Figure 5

In Ada, il modulo è chiamato *package* e si compone di due parti:

- *specification*: contiene le dichiarazioni di tipi, costanti, procedure e funzioni, e interagisce con l'ambiente esterno.
- *body*: contiene l'implementazione di procedure e funzioni dichiarate nella *specification*, ed eventualmente una routine di inizializzazione del package.

La *specification* si articola in due sottoparti: *visible* e *private*.

- Le entità dichiarate nella parte *visible* possono essere rese note ad altre unità di programma mediante la clausola `use`.
- Le entità dichiarate nella parte *private* non possono essere né esportate né dichiarate nel corpo.

La parte di specifica inizia con la parola chiave package seguita dall'identificatore del package e da is; seguono poi le dichiarazioni delle entità visibili e private

Esempio: Package per il tipo astratto Complex

```
package Type_complex is
    type Complex is record
        RL, IM: Real;
```

```
    end record;
    I: constant Complex := (0.0, 1.0);
    function "+"(x, y: Complex) return Complex;
    ...
end Type_complex;
```

Questo è un esempio di specifica di un package che realizza il tipo astratto Complex pubblicando sia il tipo ma anche la sua implementazione poichè non c'è una parte privata nella specifica. Di conseguenza, un programma che fa uso di questo package può operare su variabili complesse mediante espressioni come la seguente:

$$\text{C.IM} := \text{C.IM} + 1.0;$$

invece di ricorrere alla forma più astratta:

$$\text{C:= C} + \text{I;}$$

Per evitare il problema possiamo nascondere la struttura del tipo Complex nella parte privata.

Esempio: Nascondere la Struttura di Complex nella Parte Privata

```
package Type_complex is
    type Complex is private;
    I: constant Complex;
    function "+"(x, y: Complex) return Complex;
    ...
private
    type Complex is record
        RL, IM: Real;
    end record;
    I: constant Complex := (0.0, 1.0);
end Type_complex;
```

Avendo dichiarato il tipo Complex nella parte privata non è più possibile inizializzare la costante I nella parte pubblica, in quanto non è ancora nota la rappresentazione di Complex.

Il corpo del package fornisce le implementazioni delle procedure e funzioni dichiarate nella *specification*.

```
package body Type_complex is
    function "+"(x, y: in Complex) return Complex is
    begin
        return (x.RL + y.RL, x.IM + y.IM);
    end "+";
```

```
    ...
end Type_complex;
```

Ovviamente la struttura del tipo Complex è visibile alla parte body del package, quindi si potrà accedere ai campi RL e IM dei record Complex.
Il package potrà essere utilizzato come segue:

```
with Type_complex; use Type_complex;
procedure main is
    cpx1, cpx2: Complex;
begin
    ...
    cpx1 := cpx2 + I;
    ...
end main;
```

Si osservi che sia nella specifica e sia nel corpo di *Type_complex* non c'e alcuna dichiarazione di variabili esterne a procedure e funzioni. Ciò vuol dire che questo package non è dotato di uno stato locale, cioè non definisce un oggetto. Per questa ragione il corpo del package non necessita di un "main": non si deve inizializzare un oggetto. Il package puo in ogni caso avere un proprio main. Esso verra specificato dopo le varie procedure e funzioni e sara compreso fra un begin e l'end del package.

```
with Simple_io; use Simple_io;
package body Type_complex is
function "+"(x,y: in Complex) return Complex is
 begin
  return(x.RL+y.RL, x.IM+y.IM);
end "+";
...
begin
put(\Main of package Type_complex ");
end Type_complex;
```

L'esecuzione del main del package avverrà al momento in cui si importa il package mediante la clausola use. Si vedrà quindi visualizzare la frase "Main of package *Type_complex*".

Esempio: Astrazione di Tipo per la Pila

```
package Type_stack is    //il nome del package può essere casuale
    type Stack is private; //il nome del dato astratto è da rispettare
    procedure push(s: in out Stack; x: in Integer);
    procedure pop(s: in out Stack);
```

```
    procedure top(s: in Stack; x: out Integer);
    function empty(s: in Stack) return Boolean;
private
    max: constant := 100;
    type Stack is limited record
  // limited perchè mancano gli operatori uguale e assegnamento
        st: array(1..max) of Integer;
        top: Integer range 0..max := 0;
    end record;
end Type_stack;

package body Type_stack is
    procedure push(s: in out Stack; x: in Integer) is
    begin
        s.top := s.top + 1;
        s.st(s.top) := x;
    end push;
    procedure pop(s: in out Stack) is
    begin
        s.top := s.top - 1;
    end pop;
    procedure top(s: in Stack; x: out Integer) is
    begin
        x := s.st(s.top);
    end top;
    function empty(s: in Stack) return Boolean is
    begin
        return (s.top = 0);
    end empty;
end Type_stack;
```

Le specifiche del tipo astratto Pila prevedono anche un costruttore CreaPila, che non ha un corrispondente nella definizione del tipo astratto Stack perchè si utilizzano costruttori impliciti forniti da Ada, come la dichiarazione di una variabile di tipo Stack. Tuttavia se un costruttore dovesse essere parametrizzato sarà necessario prevedere un metodo.

In Ada, definendo un tipo come private è possibile applicare su istanze di quel tipo tutti i metodi definiti nella parte pubblica della specifica, ma anche effettuare assegnazioni e confronti di (dis-)uguaglianza.
Queste operazioni che il compilatore offre 'gratuitamente' per un tipo privato devono necessariamente essere definite in modo generale, indipendentemente da come il tipo è poi definito. Quindi saranno implementate semplicemente copiano o confrontando byte a byte le aree di memoria riservate a due dati dello stesso

tipo dichiarato come privato.

Utilizzando le operazioni predefinite potrebbero causare dei problemi:

1. l'assegnazione (:=), il confronto di eguaglianza (=) e il confronto per diseguaglianza (/=) potrebbero non far parte della specifica di un dato astratto. Quindi potrebbe non essere corrette offrirle all'utilizzatore del tipo dichiarato come privati;

2. la semantica delle operazioni potrebbe essere diversa da quella stabilita dal compilatore

Per esempio:

```
with Type_stack, use Type_stack;
procedure main is
    s1, s2: Stack;
 cmp: Boolean;
begin
    push(s1, 1);
    push(s2, 1);
 push(s1, 2);
 pop(s1);
 cmp := s1 = s2;
end main;
```

Il valore di cmp è false. Infatti i corrispondenti campi top dei record s1 e s2 sarebbero identici, mentre non avrebbero gli stessi valori i corrispondenti campi. Eppure i due stack sarebbero identici secondo la specifica algebrica di *equal(l,m)* data nel progetto.

Per evitare tutto ciò, è sufficiente dichiarare il tipo come **limited private**, che inibisce l'uso delle operazioni di assegnazione e confronto offferte per default dal compilatore.

```
package Type_stack is
    type Stack is limited private;
    procedure push(s: in out Stack; x: in Integer);
 procedure pop(s: in out Stack);
 procedure top(s: in Stack; x: out Integer);
 function empty(s: in Stack) return Boolean;
...
```

In ADA è anche possibile definire **oggetti**, cioè moduli dotati di stato locale. Per esempio, possiamo definire un oggetto di tipo Stack come segue:

```
package Stack is
    procedure push(x: in Integer);
    procedure pop;
    procedure top(x: out Integer);
    function empty return Boolean;
end Stack;
package body Stack is
 max: constant := 100;
 type Table is array(1..max) of Integer;
 st: Table;
 top: Integer range 0..max := 0;

 procedure push(x: in Integer) is
 begin
     top := top + 1;
     st(top) := x;
 end push;

 procedure pop is
 begin
     top := top - 1;
 end pop;

 procedure top(x: out Integer) is
 begin
     x := st(top);
 end top;

 function empty return Boolean is
 begin
     return (top = 0);
 end empty;
end Stack;
```

Si potrà quiindi utilizzare l'oggetto Stack come segue:

```
with Stack; use Stack;
procedure main is
begin
...
    push(1);
    push(2);
    pop;
    if empty then push(1);
...
end main;
```

In generale, un oggetto è un insieme di variabili interne ad un modulo e manipolabili esternamente solo mediante gli operatori (pubblici) definiti nel modulo stesso.
Per poter definire più oggetti simili o dello stesso tipo, cioè con medesima rappresentazione e stesso insieme di operatori, si è costretti a definire tanti moduli quanti sono gli oggetti che si vogliono usare nel programma. Tutti questi moduli differiranno solo per l'identificatore del modulo (**identificatore dell'oggetto**).
Per evitare l'inconveniente di dover duplicare un modulo si può pensare di definire un **package generico** che identifica una **classe** di oggetti simili. I singoli oggetti sono poi ottenuti con il meccanismo della **istanziazione** della classe.

In ADA un package che specifica e implementa un singolo oggetto può essere facilmente trasformato in un **generic package**, che definisce una classe di oggetti, premettendo la parola **generic** alla dichiarazione del modulo.

```
generic
package Stack
 procedure push(x: in Integer);
```

In questo modo si definisce solo una matrice degli oggetti da creare. Per ottenere i singoli oggeti dobbiamo **istanziare** il generic package:

```
package Stack1 is new Stack
package Stack2 is new Stack
```

Queste due dichiarazioni sono processate in fase di precompilazione. In particolare, per ogni occorrenza di istanziazione:

- si sostituisce la stringa dell'istanziazione con il comando di importazione di un package avente come nome quello dell'istanza (with Stack1; use Stack1;)
- si genera un package (non generico) utilizzando il generic package come matrice. Esso si ottiene rimuovendo la parola generic e sostituendo il nome dell'istanza al posto del nome del package. Il package è poi compilato separatamente.

Nell'esempio specifico, la precompilazione genera due package distinti, che variano solo nel nome del package, cioè nell'*identificatore dell'oggetto*.

Possiamo, quindi, osservare che:

- il package generico non è compilabile separatamente in quanto il nome del package non identifica un oggetto

- la generazione di package distinti, uno per ogni istanziazione, comporta la creazione di molteplici copie dello stesso codice (inefficienza di spazio)
- la creazione dei legami (binding) al compile-time garantisce l'efficienza in tempo, poichè non è necessario effettuare computazioni di legami al run-time
- l'ambiguità dovuta alle molteplici occorrenze di metodi con stesso identificatore richiede il ricorso, nel programma utilizzatore, alla notazione <*identificatore di oggetto*>.<*nome metodo*> (ad esempio s1.push(10))

2.13 Astrazione della dichiarazione di modulo

La precedente definizione di una classe corrisponde ad una particolare forma di astrazione, quella della classe sintattica **dichiarazione di modulo**.
L'operazione di istanziazione corrisponde alla invocazione di questa astrazione ed ha l'effetto di 'creare legami' (binding). In particolare, si crea un legame tra l'identificatore dell'oggetto e il nome dell classe (del modulo generico).
Pertanto la definizione di una classe corrisponde a una particolare forma di **astrazione generica**, cioè di astrazione applicata alla classe sintattica dichiarazione.
Nell'astrazione generica è possibile astrarre anche su altre dichiarazioni (per esempio, funzioni e procedure), oltre a quella di modulo.

2.14 Tipi astratti o classi?

Se in fase di progettazione di identifica l'esigenza di disporre di un dato astratto, in fase realizzativa si può:

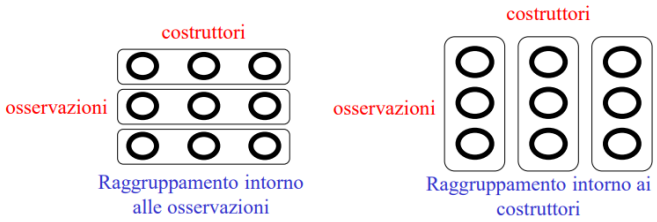
1. **definire un oggetto**: la scelta è appropriata nel caso in cui si necessita di una sola occorrenza del dato astratto
2. **definire un tipo astratto**: l'astrazione riguarda la classe sintattica tipo
3. **definire una classe**: l'astrazione riguarda la dichiarazione di un modulo dotato di stato locale

In tutti i casi la rappresentazione del dato astratto viene nascosta e la manipolazione dei valori è resa possibile solo mediante operazioni fornite allo scopo.
Tuttavia ci sono delle differenze fra tipo astratto e classe di oggetti:

- **Sintattica**: nel tipo astratto gli operatori hanno un parametro in più, relativo proprio al tipo che si sta definendo.
- **Realizzativa**: nel caso del tipo astratto gli operatori sono definiti una sola volta, mentre nel caso della classe gli operatori sono definiti tante volte quante sono le istanze. Le diverse copie degli operatori agiranno su diversi dati (gli oggetti) in memoria centrale.
- **Concettuale**: richiamando la suddivisione delle operazioni su un dato astratto in osservazioni e costruttori, si può dire che:

- il tipo astratto è organizzato intorno alle osservazioni. Ogni osservazione è implementata come una operazione su una rappresentazione concreta derivata dai costruttori. Anche i costruttori sono implementati come operazioni che creano valori. La rappresentazione è condivisa dalle operazioni, ma è nascosta ai fruitori del tipo astratto.
- la classe è organizzata intorno ai costruttori. Le osservazioni diventano metodi dei valori. Un oggetto è definito dalla combinazione di tutte le osservazioni possibili su di esso.

In breve, un tipo astratto di dato può essere inteso come un insieme con operazioni (come l'algebra) mentre le classi sono insiemi di operazioni.



Vantaggi del tipo astratto

Nei linguaggi imperativi, i valori di un tipo astratto vengono trattati alla stregua dei valori di un tipo concreto, cioè sono cittadini di prima classe. Al contrario i valori rappresentati mediante oggetti sono trattati come cittadini di terza classe in quanto:

- una procedura non può restituire l'istanza di un generic package
- non è possibile creare dinamicamente degli oggetti (le istanze sono stabilite al momento della compilazione)

I tipi astratti son utili in tutti i paradigmi di programmazione, mentre gli oggetti, essendo variabili aggiornabili, si adattano bene solo a un paradigma di programmazione side-effecting.
La notazione usata per chiamare un'operazione di un tipo astratto è più naturale perchè valori e variabili del tipo astratto sono argomenti espliciti.

Svantaggi del tipo astratto

Il tipo astratto presenta scarsa estensibilità: l'aggiunta di un nuovo costruttore comporta dei cambiamenti intrusivi nelle implementazioni esistenti degli operatori.
Ogni operatore dovrà essere opportunamente rivisto in modo da prevedere il trattamento di rappresentazioni ottenute con nuovi costruttori.

Esempio Si vuole implementare il dato astratto *geometricShape* la cui specifica algebrica è fornita di seguito:

Osservazioni	**Costruttore di g**	
	square(x)	circle(x)
area(g)	x^2	$\pi * r * 2$

```
package geometric_shape_type
type geometricShape is limited private;
 function square(x: in real) return geometricShape;
 function circle(x: in real) return geometricShape;
 function area(g: in geometricShape) return real;
private
 type geometricShape is record
  shape: char;
  value: real;
 end;
end geometric_shape_type;

package body geometric_shape_type
 function square(x: in real) return geometricShape is
  var g: geometricShape;
  begin
   g.shape := 's';
   g.value := x;
   return g;
  end;

 function circle(x: in real) return geometricShape is
  var g: geometricShape;
  begin
   g.shape := 'c';
   g.value := x;
   return g;
  end;

 function area(g: in geometricShape) return real is
  begin
   if g.shape = 's' then
    return g.value * g.value;
   else
    return 3.14159 * g.value * g.value;
   end if;
  end;
end geometric_shape_type;
```

Realizzando il dato astratto mediante classi si possono definire due moduli generici, uno per ogni forma geometrica (o costruttore):

```
generic package circle
 function area return real;
 procedure init(real);
end circle;
package body circle
 var raggio: real;
 procedure init(x: real) is
 begin
  raggio := x;
 end;

 function area return real is
 begin
  return 3.14159 * raggio * raggio;
 end;
end circle;


generic package square
 function area return real;
 procedure init(real);
end square;
package body square
 var lato: real;
 procedure init(x: real) is
 begin
  lato := x;
 end;

 function area return real is
 begin
  return lato * lato;
 end;
end square;
```

Per utilizzare una forma geometrica si deve istanziare una delle due classi e invocare il metodo *init*.

Se estendiamo la specifica del dato astratto in modo da considerare anche i rettangoli:

Osservazioni	**Costruttore di g**		
	square(x)	circle(x)	rectangle(l,m)
area(g)	xˆ2	pi*r*2	l*m

Se abbiamo specificato un tipo astratto, siamo costretti a cambiare la rappresentazione in modo da memorizzare due valori (pe ri due lati del rettangolo) e non uno:

```
type geometricShape is record
 shape: char;
 value, value2: real;
end;
```

Inoltre dobbiamo aggiungere l'opportuno costruttore e modificare anche la funzione area sempre supponendo di disporre del codice sorgente del modulo:

```
function rectangle(l,m: in real) return geometricShape is
 var g: geometricShape;
 begin
  g.shape := 'r';
  g.value := l;
  g.value2 := m;
  return g;
 end;

function area(g: in geometricShape) return real is
 begin
  if g.shape = 's' then
   return g.value * g.value;
  elsif g.shape = 'c' then
   return 3.14159 * g.value * g.value;
  else
   return g.value * g.value2;
  end if;
 end;
```

Diversamente, avendo realizzato il dato astratto mediante classi basta aggiungere un'altra classe:

```
generic package rectangle
 function area return real;
 procedure init(real, real);
end rectangle;
package body rectangle
 var lato1, lato2: real;
 procedure init(x, y: real) is
 begin
  lato1 := x;
  lato2 := y;
 end;
```

```
  function area return real is
  begin
   return lato1 * lato2;
  end;
 end rectangle;
```

Per utilizzare un rettangolo si dovrà istanziare questa classe e invocare il metodo *init*.
Nella programmazione orientata ad oggetti, per evitare la riscrittura di codice comune ad altre classi già definite è possibile ricorrere ai meccanismo di ereditarietà tra le classi.

2.15 Astrazione generica

Il concetto di astrazione generica è fondamentale nel contesto della programmazione orientata agli oggetti. L'astrazione generica si applica non solo ai concetti di tipo e funzione, ma anche alle dichiarazioni di oggetti e classi. In particolare, essa suggerisce che è possibile definire astrazioni su dichiarazioni, dove la valutazione di una dichiarazione comporta la creazione di legami (bindings). Un'astrazione generica è un'astrazione su una dichiarazione, pertanto il corpo della dichiarazione di una astrazione generica è a sua volta una dichiarazione. La chiamata di un'astrazione generica è detta istanziazione e produce dei legami elaborando la dichiarazione contenuta nel corpo dell'astrazione generica.

Un'astrazione generica può essere specificata come segue:

$$\text{generic } I(FP_1, \ldots, FP_n) \text{ is } D$$

dove:

- I è un identificatore dell'astrazione generica,
- $FP_1, \ldots, FP_n$ sono i parametri formali, e
- D è una dichiarazione che, quando elaborata, produrrà dei legami. D funge da matrice dalla quale ricavare le dichiarazioni per istanziazione.

Una dichiarazione D può essere di vario tipo, come ad esempio:

- La dichiarazione di un tipo,
- La dichiarazione di un modulo,
- La dichiarazione di una funzione,
- La dichiarazione di una procedura,
- ...

Per distinguere le diverse dichiarazioni ottenute per istanziazione, si può specificare un diverso identificatore, come nel seguente esempio:

A istantiation of I;

Così, la seguente dichiarazione generica

```
generic typeRxR is
 type RxR is record
  x: real;
  y: real;
end;
```

potrà essere utilizzata come matrice per generare le dichiarazioni per i tipi Point2D e Complex:

Point2D instantiation of typeRxR
Complex instantiation of typeRxR

L'astrazione generica trova un ampio utilizzo nella dichiarazione di moduli dotati di stato locale. Mediante l'operazione di istanziazione si ottengono diverse copie dell'oggetto che differiscono solo per il nome dell'identificatore.
In conclusione, l'astrazione generica di un oggetto corrisponde al concetto di classe.

Il generic package di Ada è una esemplificazione di astrazione generica. L'espressione:

package Stack is new Stack

è un esempio di istanziazione generica.

Le astrazioni generiche, come qualsiasi altra astrazione possono essere parametrizzate.

Esempio Definiamo la classe coda in ADA. La variabile *items* è un array di caratteri. Al fine di svincolare la definizione di classe da particolari costanti legate all'applicazione si dota l'astrazione generica del parametro formale *capacity*, che è utilizzato per dimensionare l'array. L'istanziazione deve consentire di specificare il parametro effettivo, che sarà un valore da associare al parametro formle. In ADA l'istanziazione sarà specificata come segue:

```
package line_buffer is new queue_class(100);

generic
 capacity: positive;
package queue_class is
 procedure append(newitem: in character);
 procedure remove(olditem: out character);
```

```
end queue_class;

package body queue_class is
  items: array(1..capacity) of character;
  size, front, rear: integer range 0..capacity;

  procedure append(newitem: in character) is
  begin
   ...
  end append;

  procedure remove(olditem: out character) is
  begin
   ...
  end remove;
end queue_class;
```

In principio si può applicare l'astrazione generica a qualunque dichiarazione, incluso le procedure e le funzioni.
Ad esempio, si potrebbe dichiarare una procedura T_swap per scambiare dati di tipo T predefinito:

```
generic
procedure T_swap(x, y: in out T);
proceure T_swap(x, y: in out T) is
 temp: T;
 begin
  temp := x;
  x := y;
  y := temp;
 end T_swap;
```

e ottenere diverse copie di essa per istanziazione:

```
procedure swap1 is new T_swap
procedure swap2 is new T_swap
```

In realtà è poco utile disporre di due funzioni identiche ma di nome diverso. Diversa sarebbe la situazione se potessimo dichiarare una generica procedure T_swap che opera su fati di tipo T qualunque, e potessimo specificare il tipo al momento dell'istanziazione. Per ottenere questo risultato necessitiamo di un particolare classe di parametri, quelli di tipo.
Per esempio:

```
generic
 type T is private;
```

```
procedure T_swap(x, y: in out T);
procedure T_swap(x, y: in out T) is
 temp: T;
 begin
  temp := x;
  x := y;
  y := temp;
end T_swap;
```

La clausola generic introduce un parametro di tipo e la dichiarazione che segue introduce la matrice di una procedura che scambia due dati di un tipo T generico. Le procedure effettive sono ottenute istanziando la procedura generica con i parametri di tipo effettivi da sostituire a T.
Per esempio:

procedute int_swap is new T_swap(integer); procedure str_swap is new T_swap(string);

Assumendo che i e j sono variabili di tipo integer e che s e t sono variabili di tipo string, allora:

int_swap(i, j); str_swap(s, t);

è corretto, mentre:

int_swap(s, t); str_swap(i, j); int_swap(s, j);

è errato.

In questo modo si è svincolato la definizione dello scambio di due elementi da un fattore marginale, come il tipo degli elementi da scambiare e si è garantito comunque il controllo statico dei tipo fra i parametri formali e parametri effettivi delle diverse procedure ottenute, e fra sorgente e destinazione di una assegnazione.
L'uso dei parametri di tipo in astrazioni generiche offre un buon compromesso fra necessità di dover effettuare il controllo statico dei tipi e desiderio di definire componenti software riutilizzabili.

I parametri di tipo possono essere utilizzati anche per definire astrazioni generiche di classi.
Per esempio:

```
generic
 max: positive;
 type ITEM is private;
package stack is
 procedure push(x: in ITEM);
 procedure pop;
```

```
  procedure top(x: out ITEM);
  function empty return boolean;
 end stack;
 package body stack is
  type stack is array(1..max) of ITEM;
  st: Table;
  top: integer range 0..max := 0;

  procedure push(x: in ITEM) is
  begin
   top := top + 1;
   st(top) := x;
  end push;

  procedure pop is
  begin
   top := top - 1;
  end pop;

  procedure top(x: out ITEM) is
  begin
   x := st(top);
  end top;

  function empty return boolean is
  begin
   return (top = 0);
  end empty;
```

In questo caso per creare i singoli oggetti scriveremo:

```
 declare
  package STACK_INT is new stack(100, integer);
  use STACK_INT;
  package STACK_REAL is new stack(100, real);
  use STACK_REAL;
  A: REAL; B: INTEGER;
 begin
  push(1);
  push(2.0);
  top(A);
  top(B);
 end;
```

Si osservi che non è necessario utilizzare la notazione puntata:

STACK_INT.push(1); STACK_REAL.push(2.0);

in quanto push e top sono differeniate dal contesto (tipo di parametro effettivo passato). Questo è un caso di *overloading* come si chiarirà meglio in seguito.

L'astrazione generica è quindi di supporto all'astrazione dati, in quanto permette di definire delle classi che sono invarianti ad alcuni tipi di dati neccessari per definirle.
Inoltre, essa, mediante i parametri di tipo, è applicabile a tipi astratti che possono essere così ugualmente svincolati dalla necessità di specificare il tipo degli elementi sui quali operare.

Esempio Si consideri il problema di definire dei tipi astratti per una applicazione che usa:

1. stack di interi
2. stack di reali
3. stack di un tipo astratto *point3d* utilizzato per rappresentare i punti in uno spazio tridimensionale

Una alternativa sarebbe quella di scrivere una definizione separata per ciascuno dei tre tipo. Così facendo, però, si produrrebbe codice duplicato in quanto plausibilmente simile per tutte le definizioni (differisce solo nelle parti in cui si fa riferimento ai singoli elementi dello stack). Inoltre ci sarebbe uno sforzo di programmazione ridondante accompagnato da una manutenzione complicata poichè le modifiche, come l'aggiunta di un nuovo operatore, dovrebbero essere effettuate in tutte le definizioni.
Allora si potrebbe ricorrere alla separazione delle proprietà di uno stack dalle proprietà dei suoi elementi mediante i parametri di tipo:

```
generic
 max: positive;
 type ITEM is private;
package stacks is
 type stack is limited private;
 procedure push(s: in out stack; x: in ITEM);
 procedure pop(s: in out stack, x: out ITEM);
private
 type stack is record
  st: array(1..max) of ITEM;
  top: integer range 0..max := 0;
 end record;
end stacks;

package body stacks is
 procedure push(s: in out stack; x: in ITEM) is
```

```
 begin
  s.top := s.top + 1;
  s.st(s.top) := x;
 end push;

 procedure pop(s: in out stack; x: out ITEM) is
 begin
  x := s.st(s.top);
  s.top := s.top - 1;
 end pop;
end stacks;
```

In questo modo di è definito un tipo astratto generico stack.
I diversi stack richiesti sono ottenuti per istanziazione:

```
declare
 package my_stack is new stacks(100, real);
 use my_stack;
 x: stack; y: real;
begin
 push(x, 1.0);
 pop(x, y);
end;
```

Se un'astrazione è parametrizzata rispetto a un valore, possiamo usare l'argomento valore anche se non sappiamo nulla al di fuori del suo tipo. Analogamente se un'astrazione è parametrizzata rispetto a una variabile, possiamo ispezionare e aggiornare la variabile anche se non sappiamo nulla al di fuori del suo tipo.
Ma quando si parametrizza rispetto al tipo la situazione cambi. Nell'esempio di T_swap avevamo le seguenti assegnazioni:

$$\begin{aligned} \text{temp} &:= \text{x}; \\ \text{x} &:= \text{y}; \\ \text{y} &:= \text{temp}; \end{aligned}$$

ma chi garantisce che il tipo T supporti l'operatore di assegnazione?
In ADA l'espressione

$$\text{type T is private;}$$

sottintende che l'assegnazione e i predicati $=$ e $\neq$ sono operazioni valide per il tipo effettivo denotato da T. Per questo tutti gli esempi visti finora non hanno generato errori.
Se T fosse stato definito come limited private avremmo dovuto specificare un'operazione di assegnazione e i predicati di uguaglianza e disuguaglianza.

2.16 Esercizi

Esercizio 1

Fornire una realizzazione di una tipo astratto Dizionario in Ada (si ipotizzi che Chiave e Valore siano di tipo Intero e Carattere). Per tale esercizio considerare la specifica sintattica fornita nel seguito:

- creaDizionario()→Dizionario
- aggiungi(Dizionario,Chiave,Valore)→Dizionario
- leggi(Dizionario,Chiave)→Valore
- cancella(Dizionario,Chiave)→Dizionario
- uguale(Dizionario,Dizionario)→Booleano

Mostrare l'uso del tipo scritto. Commentare il codice scritto.

Soluzione

```
package Dizionario
begin
Type Dizionario is limited private;
 procedure aggiungi(d: inout Dizionario, c: in Integer, v: in Character);
 function leggi(d: in Dizionario, c: in Integer) return Character;
 procedure cancella(d:inout Dizionario, c: in Integer);
 function uguale( d1: in Dizionario, d2: in Dizionario) return Boolean;
private:
 constant max: Positive=200;
 type Dizionario is record
  chiavi: array [1..max] of Integer;
  valori: array[1..max] of Character;
  top: Positive=0;
 end
end

package body Dizionario
begin
 function cerca(d: in Dizionario, c: in Integer) return Positive
 var i: Positive
 begin
  if D.top > 0 then
   for i=1 to D.top loop
    if c= D.chiavi[i] then
     return i;
  return 0;
 end
```

```
procedura aggiungi(d: inout Dizionario, c: in Integer, v: in Character)
begin
 if d.top < max then
  if cerca(D,c) =0 begin
   D.top=D.top+1;
   D.chiavi[D.top]=c;
   D.valori[D.top]=v
  end
 else raise Exception;
end

function leggi(d: in Dizionario, c: in Integer) return Character
 var x:Positive
 begin
  x=cerca(D,c);
  if x=0 then raise Exception
  else return D.valori[x];
 end

procedure cancella(d:inout Dizionario, c: in Integer)
 var x, i:Positive;
 begin
  x=cerca(D,c);
  if x=0 then raise Exception;
  else
  begin
   for i= x to D.top-1 loop
   begin
    D.chiavi[i]=D.chiavi[i+1]
    D.valori[i]=D.valori[i+1];
   end
   D.top=D.top-1;
  end
 end

function uguale( d1: in Dizionario, d2: in Dizionario) return Boolean
 var i:Positive
 begin
  if d1.top= d2.top then
   for i=1 to D1.top loop
   begin
    x=cerca(d2,d1.chiavi[i])
    if x=0 then return false;
    else
     if d2.valori[x] not = d1.valori[i] then return false;
```

```
    end
    return true;
   else return false;
  end
end

...
With Dizionario; use Dizionario;
var D1,D2:Dizionario;
var x:Positive:
inserisci(D1,1,'a')
inserisci(D1,2,'b')
inserisci(D2,1,'c')
```

Esercizio 2

Fornire una realizzazione di una tipo astratto Dizionario in Ada (si ipotizzi che Chiave e Valore siano di tipo Intero e Carattere). Per tale esercizio considerare la specifica sintattica fornita nel seguito:

- creaDizionario()→Dizionario
- aggiungi(Dizionario,Chiave,Valore)→Dizionario
- leggi(Dizionario,Chiave)→Valore
- cancella(Dizionario,Chiave)→Dizionario
- uguale(Dizionario,Dizionario)→Booleano

Mostrare l'uso della classe scritta. Commentare il codice scritto.

Soluzione

```
Generic
max:positive
Package Dizionario
Begin
 procedure aggiungi( c: in Integer, v:In Character);
 function leggi(c:in Integer) return Characger;
 procedure cancella(c: in Integer);
end

package body Dizionario
begin
 var top: Positive = 0;
 var chiavi: array [1...max] of Integer;
 var valori: array [1..max] of Character;

 //funzione creata internamente
 function cerca(c: in Integer) return Positive
 var i: Positive
 begin
  if top > 0 then
   for i = 1 to top loop
    if c = chiavi[i] then
     return i;
 return 0;
 end

 procedura aggiungi(c: in Integer, v: in Character)
 begin
  if top < max then
```

```
    if cerca(c) = 0 begin
     top = top + 1;
     chiavi[top] = c;
     valori[top] = v
    end
   else raise Exception;
  end

  function leggi(c: in Integer) return Character
  var x: Positive
  begin
   x = cerca(c);
   if x = 0 then raise Exception
   else return valori[x];
  end

  procedure cancella(c: in Integer)
  var x, i:Positive;
  begin
   x = cerca(c);
   if x = 0 then raise Exception;
   else
   begin
    for i = x to top-1 loop
    begin
     chiavi[i] = chiavi[i+1]
     valori[i] = valori[i+1];
    end
    top = top - 1;
   end
  end
 end

 ..
 with Dizionario;
 package d1 is new Dizionario(3);
 package d2 is new Dizionario(300);
 with d1; use d1;
 with d2, use d2;
 d1.aggiungi(1,'a');
```

3 Paradigma Object Oriented

Nella programmazione imperativa, gli oggetti possono essere definiti, ma il loro utilizzo non è rigidamente imposto e dipende dall'autodisciplina dei programmatori. Inoltre, gli oggetti non godono dello status di cittadini di prima classe. Ciò significa che non hanno la stessa importanza e trattamento dei dati primitivi come interi o stringhe.

D'altra parte, nel paradigma orientato agli oggetti, c'è una trasformazione significativa. Gli oggetti diventano cittadini di prima classe, il che significa che hanno un ruolo centrale e sono trattati su un piano paritario con altri elementi del linguaggio. Questa trasformazione può essere vista come una rivoluzione poiché gli oggetti assumono un ruolo fondamentale nella progettazione e nella programmazione.

Principi come l'information hiding e l'incapsulamento diventano cardini del paradigma orientato agli oggetti, permettendo una migliore organizzazione, manutenzione e scalabilità del codice. Inoltre, l'orientamento agli oggetti incoraggia una maggiore modularità e riusabilità del codice attraverso concetti come l'ereditarietà e il polimorfismo, consentendo agli sviluppatori di costruire sistemi complessi in modo più efficiente e intuitivo.

3.1 Oggetti

Gli oggetti sono strutture dati complesse che combinano sia lo stato, che rappresenta le informazioni associate all'oggetto, sia il comportamento, che definisce le azioni che l'oggetto può eseguire.

Lo stato di un oggetto è tipicamente identificato dal contenuto di una specifica area di memoria, mentre il comportamento è definito da una collezione di procedure e funzioni chiamate metodi, che operano sullo stato dell'oggetto stesso.

Da una prospettiva di progetto, gli oggetti sono utilizzati per modellare le entità presenti nel dominio dell'applicazione, consentendo agli sviluppatori di creare astrazioni efficaci e rappresentazioni del mondo reale all'interno del software. Questo approccio favorisce una progettazione modulare, manutenibile e scalabile, in cui gli oggetti possono essere istanziati, manipolati e interagiti tra loro per creare sistemi complessi e funzionali. Inoltre, l'incapsulamento degli stati e dei comportamenti all'interno degli oggetti favorisce la sicurezza e l'integrità del sistema, poiché limita l'accesso diretto ai dati e alle operazioni, consentendo un migliore controllo e gestione delle interazioni tra le varie parti del software.

Esempio In un gioco elettronico che us una palla, si può penare alla palla come un oggetto dotato di uno:

- **Stato**: posizione, velocità, colore, dimensione, ecc.

- **Comportamento**: muoversi, rimbalzare, cambiare colore, cambiare dimensione, apparire e sparire, ecc.

Le variabili e i metodi definiti nell'oggetto *palla* stabiliscono lo stato e il comportamento che sono rilevanti all'uso della palla nel gioco elettronico.

3.2 Identificatore di oggetto

Un oggetto ha la sua **identità**, cioè è riconoscibile indipendentemente dal suo stato corrente. Per questo ogni oggetto ha un **identificatore di oggetto** (object identifier, OID) che lo identifica univocamente. In alcuni contesti gli OID sono anche detti **riferimenti** (references). Un identificatore di oggetto è immutabile, cioè non può essere modificato da una qualche opzione di programmazione. Cambiare l'OID di un oggetto equivale alla cancellazione dell'oggetto e alla creazione di un altro oggetto con lo stesso stato.

Normalmente gli OID sono assegnati in modo automatico agli oggetti, sicchè non hanno un significato nel mondo reale. In molti ambienti di programmazione object-oriented, l'OID corrisponde all'indirizzo dell'area di memoria che conserva lo stato dell'oggetto. Quasi mai il programmatore utilizza esplicitamente i riferimenti. Generalmente questi vengono legati a delle varibili e si fa riferimento agli oggetti mediante gli identificatori di variabile.

Variabili distinte possono riferirsi al medesimo oggetto. In questo caso si hanno degli alias. La presenza di alias non significa che un oggetto non è identificato univocamente, ma semplicemente che diversi identificatori di variabile sono stati legati al medesimo riferimento di oggetto.

Lo stato di un oggetto può anche contenere il riferimento ad un altro oggetto. Si dice che un oggetto punta ad un altro. Il puntamento è asimmetrico. La simmetria si ottiene mediante la reciprocità di puntamento.

3.3 Classi

Una **classe** è la descrizione di una famiglia di oggetti che condividono la stessa struttura (gli attributi) e il medesimo comportamento (operazioni).
Nella programmazione OO **ogni oggetto è un'istanza di una classe**, cioè un oggetto non può essere ottenuto se non si definisce la sua classe di appartenenza. Analogamente nella modellazione OO le istanze esistono in quanto ci sono le loro astrazioni.
Idealmente una classe è una realizzazione di un dato astratto. Questo significa che i dettagli della realizzazione sono normalmente nascosti. Ogni classe ha una doppia componente:

1. **Componente statica**, i dati, costituita da campi o **attributi** dotati di nome, che contengono un valore. I campi caratterizzano lo stato degli

oggetti durante l'esecuzione del programma.

Gli attributi si distinguono in base al loro ambito d'azione (**scope**):

- **Attributi d'istanza**: sono associati ad una istanza e hanno un tempo di vita pari a quello dell'istanza alla quale sono associati.
- **Attributi di classe**: sono associati alla classe e non alle istanze. Sono condivisi da tutte le istanze della classe e sono definiti una sola volta. Il loro tempo di vita è lo stesso della classe.

Gli attributi d'istanza contribuiscono a caratterizzare lo stato di ogni singolo oggetto, mentre gli attributi di classe contribuiscono a definire il fattore comune allo stato di tutti gli oggetti di una classe.

2. **Componente dinamica**, i **metodi** (operazioni), che caratterizzano il comportamento comune degli oggetti appartenenti alla classe, cioè i servizi che possono essere richiesti a un oggetto di una classe durante l'esecuzione del programma. I metodi manipolano gli **attributi**.

I metodi possono essere classificati:

1. **Metodi costruttori**: sono invocati per creare (istanziare) gli oggetti e inizializzarli.
2. **Metodi distruttori**: sono invocati per distruggere gli oggetti rimuovendoli dalla memoria.
3. **Metodi di accesso**: restituiscono astrazioni significative dello stato di un oggetto.
4. **Metodi di trasformazione**: modificano lo stato di un oggetto.

I metodi di accetto e trasformazioni possono essere distinti in:

1. **Metodi di istanza**: operano su almeno un attributo di istanza, pertanto possono essere invocati solo specificando l'istanza.
2. **Metodi di classe**: operano esclusivamente su attributi di classe, pertanto possono essere invocati specificando la classe. Si possono invocare metodi di classe anche quando non è stato creato alcun oggetto per quella classe.

L'invocazione di un metodo di classe può avvenire anche specificando un oggetto (e non la classe), tuttavia ciò è sconsigliato perchè non evidenzia il fatto che si manipolano solo attributi di classe.
Non è possibile invocare un metodo di istanza sulla classe.

3.4 UML

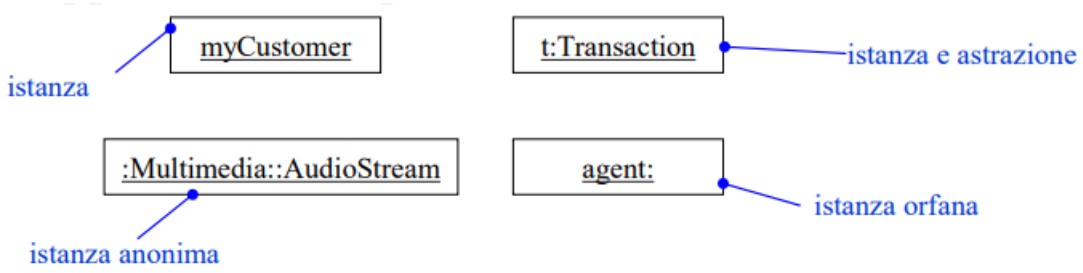
UML, acronimo di Unified Modeling Language, è un **linguaggio visuale** ampiamente utilizzato nell'ambito dello sviluppo software per definire, progettare, realizzare e documentare sistemi orientati agli oggetti. Questo linguaggio fornisce una serie di strumenti e convenzioni grafiche che consentono agli sviluppatori di rappresentare in modo chiaro e conciso i concetti e le relazioni all'interno di un sistema.

Uno dei principali vantaggi di UML è la sua **universalità**: può essere impiegato per modellare una vasta gamma di sistemi, indipendentemente dalla loro architettura, tecnologie utilizzate o tipologia applicativa. Questo rende UML uno strumento flessibile e adattabile alle esigenze di progettazione di diversi contesti, che vanno dai sistemi gestionali ai sistemi real-time e oltre.

Tra le sue funzionalità principali, UML supporta attivamente il processo di progettazione di nuovi sistemi, consentendo agli sviluppatori di visualizzare e analizzare in dettaglio i requisiti, le interazioni e le relazioni tra le varie componenti del sistema. Inoltre, UML è prezioso anche per la documentazione dei sistemi esistenti, offrendo un mezzo efficace per rappresentare e comunicare l'architettura e il funzionamento di un'applicazione software agli stakeholder.

3.4.1 Oggetto

In UML un **oggetto (o istanza)** è graficamente rappresentato in questo modo:

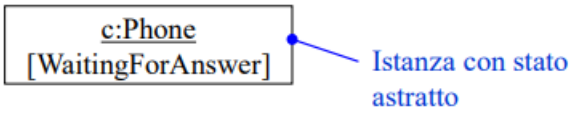


E' possibile indicare:

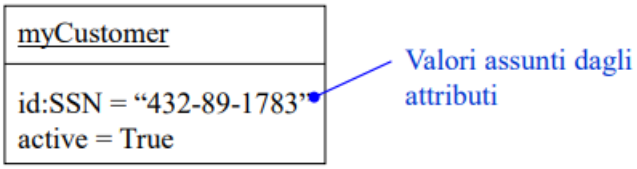
- solo il nome (OID) dell'istanza
- nome dell'istanza e della astrazione (la classe) a cui appartiene
- solo il nome dell'astrazione, qualora non si conosca a priori il nome dell'istanza
- un'istanza orfana, se non si conosce a priori la sua astrazione

3.4.2 Stato

Lo **stato** di un oggetto può essere rappresentato in modo astratto:



oppure indicando esplicitamente i valori assunti dagli attributi dell'oggetto:



3.4.3 Attributi

Graficamente gli attributi sono indicati sotto il nome della classe e si può specificare l'insieme dei valori assunti (oggetti di una classe) e una eventuale inizializzazione.

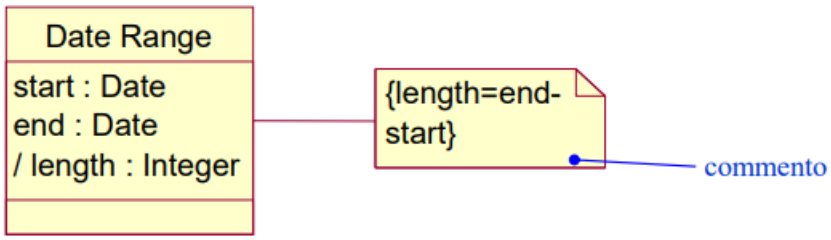
Rettangolo
base: real = 10 altezza: real

Un attributo di classe (detto **statico** in UML) è indicato come sottolineato:

Rettangolo
numRettangoli: integer=0
base: real = 10 altezza:real

L'attributo numRettangoli indica il numero di oggetti della classe Rettangoli che sono stati istanziati. E' un attributo statico (o di classe) in quanto condiviso da tutte le istanze della classe Rettangoli.
Esso è inizializzato a 0 e verosimilmente sarà incrementato dai costruttori della classe Rettangoli, mentre sarà decrementato dai distruttori.

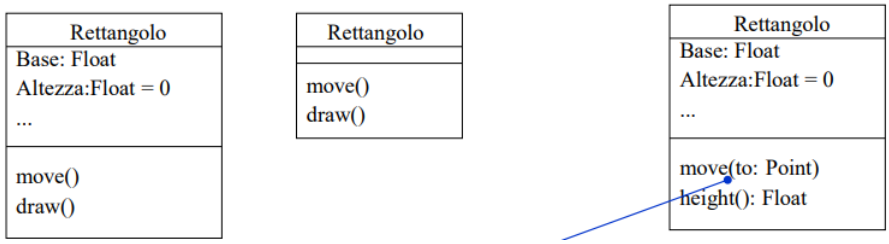
Gli attributi **derivati** sono quelli che possono essere calcolati partendo da attributi. UML prevede una rappresentazione specifica mediante una /.



Il commento sulla destra fa parte della notazione standard UML e in questo caso è utilizzato per indicare come si calcola l'attributo derivato length.
I commenti possono essere aggiunti a qualunque elemento della notazione UML.

3.4.4 Operazioni

In UML le operazioni sono indicate graficamente in una sezione al di sotto degli attributi.



Di una operazione si può specificare la segnatura.

3.4.5 Classe

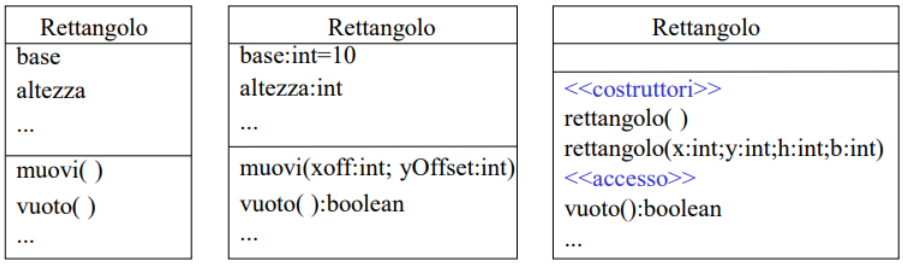
La **classe** può essere rappresentata graficamente mediante un rettangolo:

Ogni classe deve avere un nome che lo contraddistingue dalle altre. Questo può essere semplice o indicare un percorso.

Una classe può essere rappresentata a diversi livelli di dettaglio:

- specificandone solo il nome

- specificandone gli attributi e i metodi



I diversi livelli di dettaglio consentono a chi progetta/modella di attribuire maggiore o minore importanza a determinati fattori a seconda della vista del sistema che si sta considerando.

3.4.6 Stereotipi

Nel precedente esempio si è fatto uso della notazione <<**costruttori**>> e <<**accesso**>>. Questi sono due tipici esempi di **stereotipi**.

Gli stereotipo sono dei tipici *meccanismi di estensibilità* del vocabolario UML e permettono di creare nuovi blocchi per la costruzione dei modelli, derivandoli da blocchi già esistenti ma rendendoli specifici per il particolare dominio.

Nel precedente esempio è stato esteso il blocco *operazioni* in modo da poter distinguere le diverse tipologie di operazioni.

Gli stereotipi sono identificabili perchè racchiusi da una coppia di caporali (*guillemet*, in inglese) << >>.
Sono già previsti diversi stereotipi in UML.

3.4.7 Visibilità

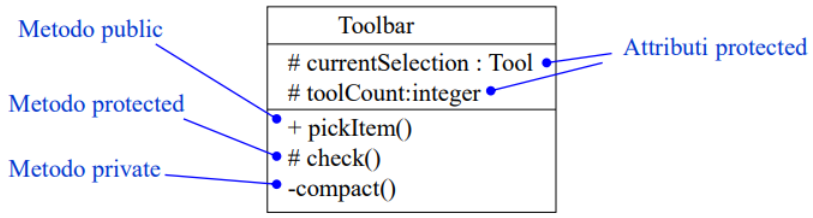
Gli attributi e i metodi di una classe possono avere diversi livelli di **visibilità**. Un elemento (attributo o metodo) ha visibilita:

- **public**: l'elemento può essere visto, utilizzato e/o invocato da altre classi.
- **protected**: l'elemento è visibile all'interno del package e all'esterno solo ai discendenti della classe di appartenenza.
- **private**: l'elemento è accessibile solo dalla classe di appartenenza.
- **package**: l'elemento è visibile all'interno del package che contiene la classe in cui l'elemento è definito.

Ad esempio, un metodo pubblico può essere invocato da qualunque punto del codice (purchè la classe sia *importata* in qualche modo), mentre un metodo privato può essere invocato solo da altri metodi della stessa classe.

UML consente di specificare i livelli di visibilità di attributi e metodi utilizzando la seguente notazione:

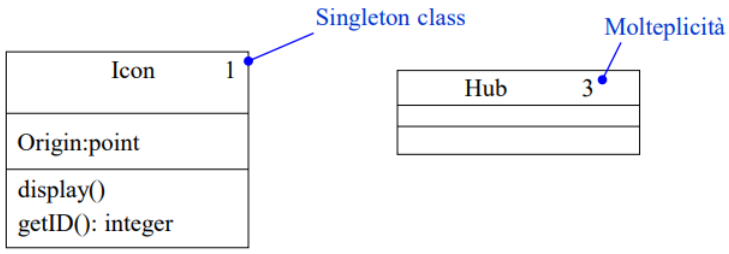
- **public**: elemento preceduto da un +.
- **protected**: elemento preceduto da un #.
- **private**: elemento preceduto da un −.
- **package**: elemento preceduto da un ˜.



3.4.8 Molteplicità di classe

Con il termine **molteplicità di classe** si intende il numero di istanze di una classe che essa può avere. Generalmente non si pone un limite, tuttavia in alcuni casi è necessario indicare che la classe può avere una sola istanza (**classe singoletto**, *singleton class*) o comunque un numero ben definito di istanze.

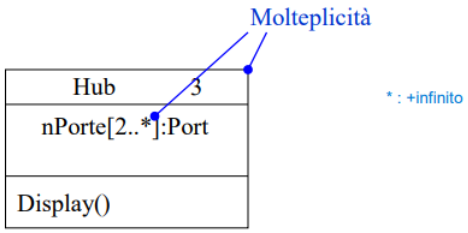
UML impone che tale valore sia indicato in alto a destra nell'icona raprresentante la classe.



Per realizzare una classe singoletto si rendono privati i costruttori della classe e dotando la classe di un attributo statico che è inizializzato all'unico oggetto di quella classe. Nessun metodo della classe singoletto invocherà il costruttore, nè potrà farlo un utente della classe. Inoltre può essere utile disporre di un metodo di classe che restituisce l'unica istanza memorizzata nella stessa classe singoletto.

3.4.9 Molteplicità di attributo

E' possibilità indicare la molteplicità anche per gli attributi, subito dopo il loro nome.



3.4.10 Schema di definizione di un attributo

Si è visto come è possibile specificare in UML le varie caratteristiche degli attributi di una classe. A riepilogo, mostriamo lo schema generale per la definizione di un attributo:

[visibilità] nome [molteplicità] [: tipo] [= valore iniziale] [proprietà]

dove [..] significa opzionale.

Alcuni esempi di definizioni lecite di attributo sono:

- origine	solo il nome
- + origine	visibilità e nome
- origine: point	nome e tipo
- testa: *elemento	nome e tipo complesso
- nome [0..1]: String	nome, molteplicità e tipo
- origine: Point = (0,0)	nome, tipo e valore iniziale
- id: Integer {frozen}	nome, tipo e proprietà

Focalizziamo l'attenzione sull'ultimo degli esempi:

- id: Integer {frozen}	nome, tipo e proprietà

In UML esistono tre **proprietà** che possono essere utilizzate con gli attributi:

- **changeable**: non vi sono restrizioni sulla modificabilità dell'attributo.
- **frozen**: l'attributo non può essere modificato dopo che l'oggetto è stato inizializzato.
- **addOnly**: per gli attributi con molteplicità maggiore di uno, i valori possono essere aggiunti, ma una volta creati, non possono più essere rimossi o modificati.

Nel caso in cui la proprietà non viene specificata si sottintende che assume valore *changeable*.

3.4.11 Schema di definizione di una operazione

Prima di procedere con la definizione dello schema per la definizione di un'operazione, è necessario sottolineare che UML distingue tra operazione e metodo:

- **operazione**: è un servizio che può essere richiesto alla classe
- **metodo**: è un'implementazione concreta del servizio.

Infatti, possono anche esistere diversi metodi per la stessa operazione nei diversi livelli della gerarchia delle classi.

Lo schema generale per la definizione di una operazione è il seguente:

[visibilità] nome [(parametri)] [: valore di ritorno] [proprietà]

Alcuni esempi di definizioni lecite di operazioni sono:

- visualizza	solo il nome
- + visualizza	visibilità e nome
- set(n: nome, s: String)	nome e parametri
- getID(): Integer	nome e tipo del dato restituito
- riparti() {guarded}	nome e proprietà

Ogni parametro riportato nella segnatura prende la forma:

[direzione] nome: tipo [= valore iniziale]

La **direzione** può assumere uno dei seguenti valori:

- **in**: parametro di input, non può essere modificato
- **out**: parametro di output, può essere modificato per comunicare un'informazione al chiamante.
- **inout**: parametro di input che comunque può essere modificato.

UML fornisce diverse proprietà predefinite per le operazioni:

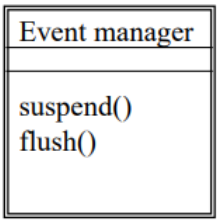
- **isQuery**: l'esecuzione dell'operazione lascia lo stato del sistema immutato, un'operazione con tale proprietà è quindi priva di *side-effect*.
- **leaf**: l'operazione non può essere più specializzata (overriding) nelle sottoclassi (vedi *final* in Java).
- **sequential**: i chiamanti (callers) di questo oggetto devono coordinarsi affinchè solo uno alla volta richieda il servizio. Nel caso di sovrapposizione la semantica e l'integrità dell'oggetto non sono garantite.
- **guarded**: simile al caso precedente, in ogni istante un solo chiamante può usufruire del servizio, tuttavia, in questo caso, la sequenzialità del servizio è gestita dalla classe proprietaria del servizio stesso (vedi *synchronized* in Java).
- **concurrent**: la semantica e l'integrità dell'oggetto sono garantite anche in caso di chiamate multiple.

3.4.12 Classi attive

Le ultime tre proprietà sono rilevanti solo in classi **attive**, cioè in classi i cui **oggetti** sono **attivi**.

Un oggetto è attivo se esso ha un thread e può far partire un thread concorrente.

Una classe attiva è simile ad una classe con l'eccezione che le sue istanze rappresentano elementi il cui comportamento è concorrente con gli altri.
Essa è mostrata con bordi raddoppiati:



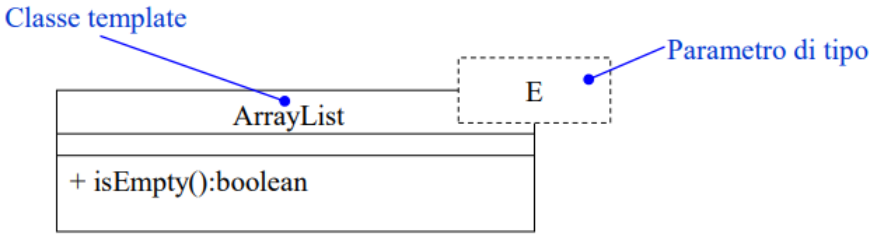
3.4.13 Classi template

Una **classe template** definisce una famiglia di classi parametrizzate (con parametro di tipo). Non è possibile usare direttamente una classe template. E' necessario prima specificare il tipo (operazione di istanziazione).
In Java, una classe template corrsponde a una classe generica.

Esempio:

public class ArrayList<E>

Notazione UML:



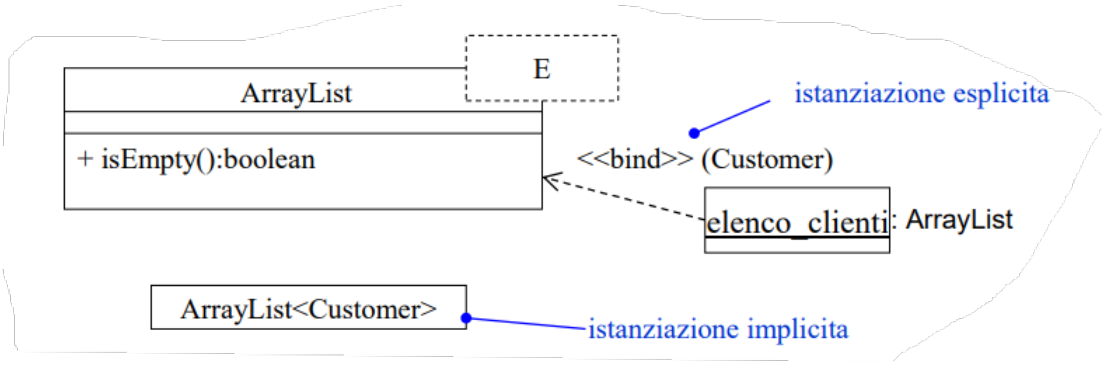
Si potrebbe voler istanziare questo template per creare una lista di clienti.

ArrayList<Customer> elenco_clienti;

L'istanziazione di una classe template può essere effettuata in due modi:

1. Implicitamente, dichiarando una classe il cui nome esplicita i parametri.

2. Esplicitamente, mediante una dipendenza stereotipata <<bind>>.

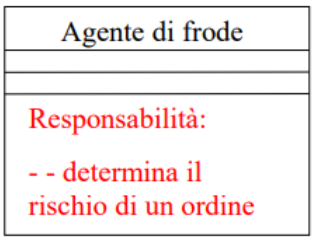


3.4.14 Individuazione delle responsabilità

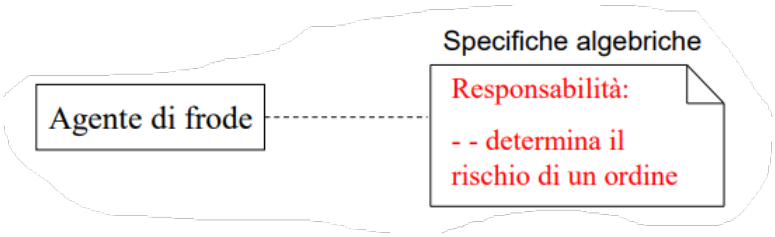
In una buona modellazione Object-Oriented di un sistema sofware è necessario stabilire le responsabilità da attribuire a ciascuna classe individuata.

UML consente di modellare le responsabilità in due modi:

- specificandole all'interno della classe:



- utilizzando le note:

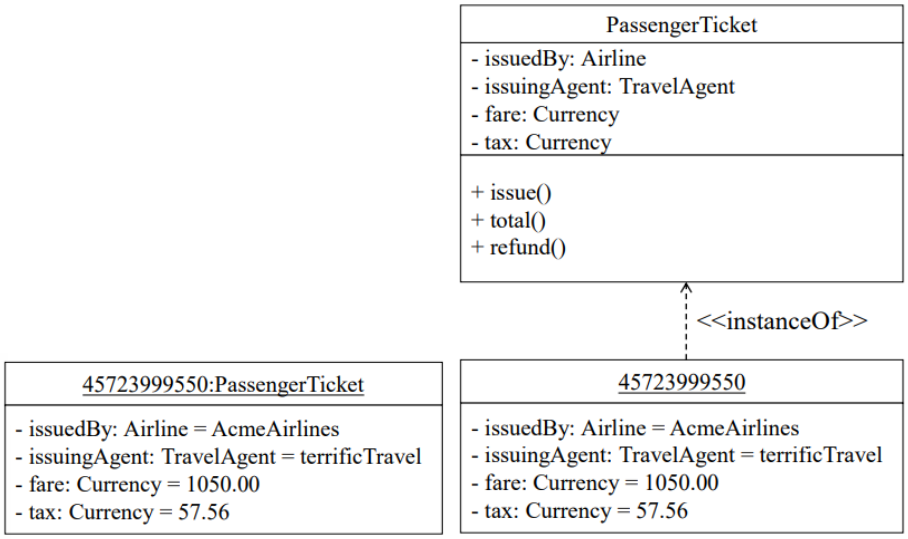


3.4.15 Relazione Instance-of

Fra un oggetto e una classe sussiste una relazione **instance of** che specifica che un oggetto è una istanza di una classe.

In UML questa relazione è resa con lo stereotipo <<instance of>>.

Le seguenti notazioni grafiche sono semanticamente equivalenti (ma cambia il livello di dettaglio):



(a) (b)

3.5 Ereditarietà

Nella progettazione e programmazione OO una relazione fondamentale, e caratteristica fondamentale del paradigma object-oriented, è quella esistente fra le classi: la **relazione di ereditarietà** (inheritance).

Una classe è considerata come un *repertorio di conoscenze* a partire dal quale è possibile definire altre *classi più specifiche*, che completano le conoscenze della loro classe madre.

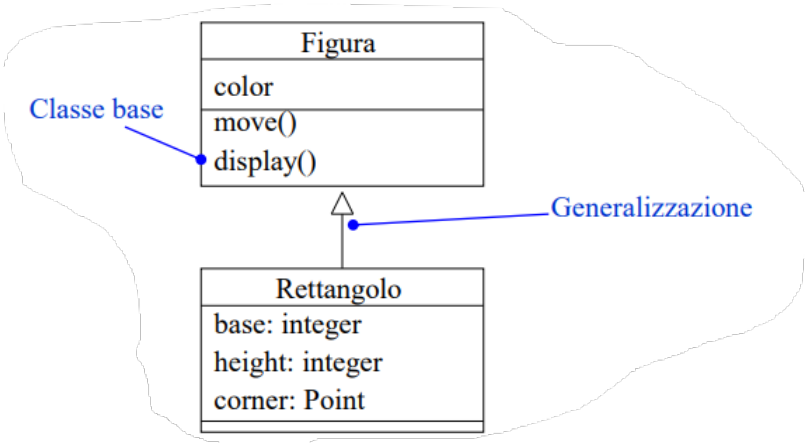
Una **sottoclasse**, è dunque una *specializzazione* della descizione di una classe, detta la sua **superclasse**, della quale essa mutua (parte di) gli attributi e i metodi. Concettualmente è come se le informazioni della superclasse fossero ricopiate nella sottoclasse. A livello fisico, comunque, le duplicazioni sono evitate.

Ci sono diverse forme di ereditarietà.

3.5.1 Ereditarietà per estensione (extension inheritance)

La forma più comune di ereditarietà è quella per estensione. In questo caso, una sottoclasse estende la superclasse aggiungendo nuovi attributi e/o nuovi metodi non presenti nella superclasse e non applicabili a istanze della superclasse.

La visibilità (pubblica, protetta, privata, package) degli attributi e delle operazioni ereditate dalla superclasse non è modificata.



La superclasse non conosce le classi figlie. Le classi figlie conoscono la superclasse.

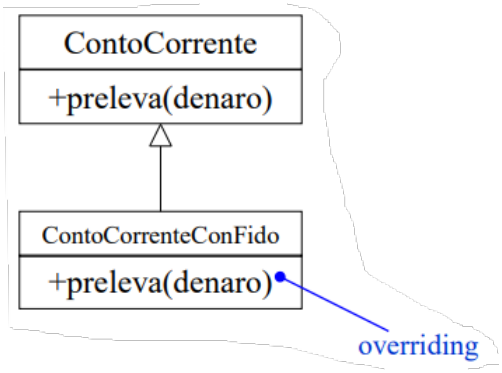
3.5.2 Ereditarietà per variazione funzionale (functional variation inheritance)

In questo caso, si ridefiniscono alcune caratteristiche (metodi) della superclasse quando quelle ereditate si rivelano inadeguate per l'insieme di oggetti descritti dalla sottoclasse.

La ridefinizione (**override**) del metodo ereditato riguarda solo l'implementazione e non la segnatura (nome, parametri formali e parametri di ritorno).

Ogni richiesta di esecuzione del metodo ridefinito da parte di un oggetto della sottoclasse farà riferimento alla nuova implementazione fornita nella sottoclasse.

Esempio Nella classe *ContoCorrente* il metodo *preleva* controlla che il conto non vada in rosso. Il metodo *preleva* della classe *ContoCorrenteConFido* ridefinisce il metodo della superclasse per controllare che il prelievo non vada oltre il fido concesso.
Se cambiano le regole del contesto occorre modificare il metodo preleva di *ContoCorrente*, ma la modifica non viene riportata automaticamente anche all classe *ContoCorrenteConFido*.



La ridefinizione non è incrementale, quindi i cambiamenti nel metodo originale devono essere riportati anche nei metodi ridefiniti. Purtroppo non c'è alcuna garanzia che questo accada e si possono introdurre degli errori.

Anche nell'ereditarietà per variazione funzionale, la visibilità (pubblica, protetta, privata, package) degli attributi e delle operazioni ereditate dalla superclasse non è modificata.
Osserviamo inoltre che la variazione funzionale attiene solo le operazioni di accesso e trasformazioni di una classe e non i costruttori degli oggetti.

3.5.3 Principio di sostituibilità

Data una dichiarazione di una variabile o di un parametro il cui tipo è dichiarato come X, una qualunque istanza di una classe che è discendente di X può essere usato come valore effettivo senza violare la semantica della dichiarazione e il suo uso.
In altri termini, l'istanza di una sottoclasse può essere usata al posto di un'istanza della superclasse senza che il comportamento del programma cambi.

Il principio di sostituibilità è alla base del polimorfismo di inclusione, nella programmazione orientata a oggetti. Questo principio è stato introdotto dalla Prof.ssa Barbara Liskov del MIT nel 1987.

Di conseguenza, una sottoclasse non può rimuovere o rinunciare a proprietà/metodi della superclasse. Altrimenti una istanza della sottoclasse non sarà sostituibile in una situazione in cui si dichiara l'uso di istane della superclasse.

In effetti, preservando la visibilità degli attributi e dei metodi ereditati, così come accade nelle precedenti forme di ereditarietà viste, si garantisce che gli oggetti della sottoclasse offrano quanto meno gli stessi servizi degli oggetti della superclasse (anche se i servizi potranno essere implementati diversamente).

Pertanto il principio di sostituibilità (o polimorfismo di inclusione) è compatibile con l'ereditarietà per estensione e variazione funzionale.

3.5.4 Ereditarietà e relazione is_a

Nell'ereditarietà per estensione e per variazione funzionale, la relazione di ereditarietà fra classi corrisponde a una relazione di **generalizzazione** (o **is_a**). Ciò perchè ogni istanza di una classe derivata da una classe base va considerata come (è anche) una istanza della classe base. Il simbolo ↑ usato in UML denota un generalizzazione.

Esempio Le istanze di Rettangolo cono anche istanze di Figura.

3.5.5 Ereditarietà di implementazione (implementation inheritance)

In questo caso, una sottoclasse utilizza il codice della superclasse (definizioni di attributi e metodi) per implementare l'astrazione associata.

Esempio Una classe *Vettore* mette a disposizione una rappresentazione di un vettore, un costruttore, un metodo di accesso (ad un elemento sulla base dell'indice) e uno di trasformazione (cambiamento di stato di un elemento di cui si conosce la posizione).

Una classe *Pila* può essere definita come sottoclasse di *Vettore*, in modo da potersi basare sulla rappresentazione del vettore per poter rappresentare una pila. Si dice che la realizzazione del dato astratto *Pila* è delegata a quella del vettore. Se si cambia la rappresentazione del vettore si cambia anche quella della pila.

Grazie all'ereditarietà per implementazione, la realizzazione degli operatori *push, pop, top* previsti per una pila può basarsi sugli operatori messi a disposizione dalla classe *Vettore*.
Si noti bene che la *Pila* ha un diverso insieme di operatori rispetto a quelli del *Vettore*. L'ereditarietà riguarda solo la realizzazione del vettore e l'implementazione dei metodi, non l'interfaccia della classe (ciò che è visibile agli utilizzatori di *Vettore*).

Pertanto l'ereditarietà di implementazione comporta la modifica alla visibilità delle caratteristiche ereditate.

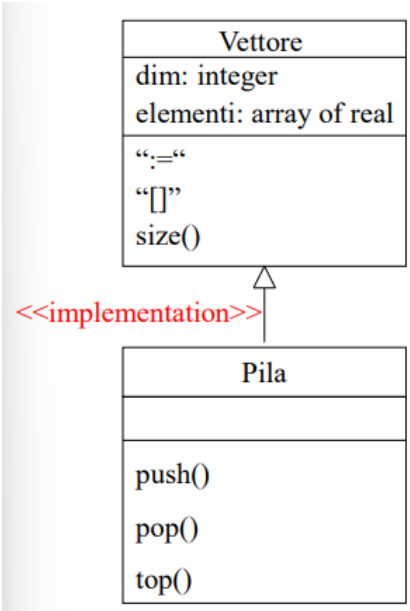
L'ereditarietà a base privata supportata dal C++ e da Ada-95 permette di rendere privati gli attributi e gli operatori pubblici ereditati.
Visibilità con ereditarietà a base pubblica:

Base\Derivata	Visibile ai clienti	Visibile alla sottoclasse
Public Spec.	✓	✓
Private Spec.	✗	✗

Visibilità con ereditarietà a base privata:

Base\Derivata	Visibile ai clienti	Visibile alla sottoclasse
Public Spec.	✗	✓
Private Spec.	✗	✗

In UML l'ereditarietà di implementazione è indicata utilizzando lo stesso simbolo della generalizzazione, ma specificando a fianco lo stereotipo <<implementation>>.



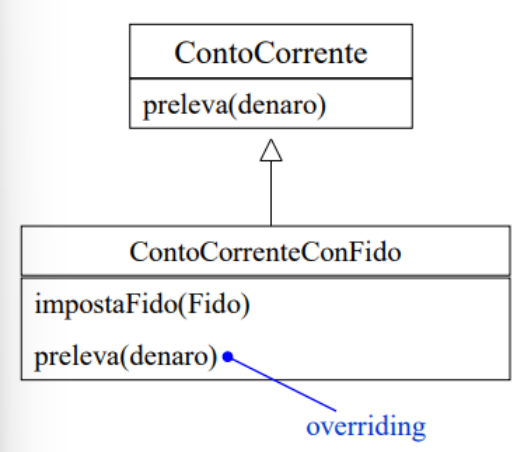
L'ereditarietà di implementazione non è compatibile con il principio di sostituibilità. Pertanto se la classe Y eredita solo l'implementazione della classe X, non si può riutilizzare su istanze di Y tutto il codice in cui si dichiarano e utilizzano dati di classe X, in quanto non vale il polimorfismo di inclusione.

L'ereditarietà di implementazione permette un riuso **parziale** del codice.

3.5.6 Combinazione di ereditarietà

Nella progettazione di una classe si possono combinare diverse forme di ereditarietà.

Esempio La classe *ContoCorrenteConFido* implementa l'ereditarietà per estensione, aggiungendo il metodo *impostaFido* e per variazione funzionale ridefinendo il metodo *preleva*.

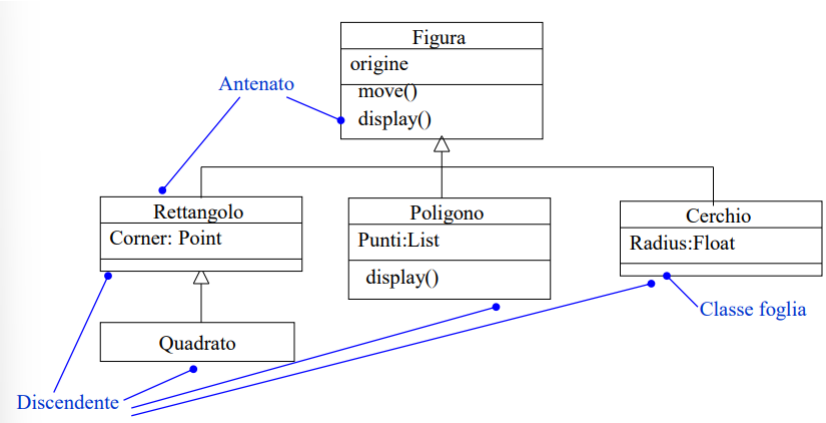


3.5.7 Proprietà della relazione di ereditarietà

La rappresentazione della relazione di generalizzazione fra un insieme di classi definisce un **grado di ereditarietà** che è un **grafo orientato aciclico**. La relazione di generalizzazione è transitiva e antisimmetrica:

- la transitività comporta che le caratteristiche delle classi superiori sono ereditate dalle classi inferiori.
- l'antisimmetria definisce una direzione di attraversamento del grafo di ereditarietà che porta dalla sottoclasse alla superclasse.

Partendo da una classe C e seguendo la direzione che porta al genitore, si trovano tutte le classi **antenate**, che per estensioni sono chiamate **superclassi** di C. Seguendo la direzione opposta di trovano tutte le classi **discendenti** di C.



Ogni classe C ha il proprio **grafo di ereditarietà G(C)** che è una restrizione del grafo di ereditarietà completo delle sole superclassi della classe in esame.

3.5.8 Ereditarietà singola

In questo caso specifico, il grafo è in realtà un albero: ogni classe ha una sola superclasse diretta. Si dice che l'ereditarietà è **semplice** o **singola**.

Il grafo di ereditarietà G(C) di una classe C è una catena di antenati. Gli elementi della catena sono ordinati secondo una relazione d'ordine totale. Poichè un metodo può essere ridefinito in più classi si pone il seguente

Problema Sia dato un metodo m, eventualmente ereditato, della classe C_1. Da quale classe $C_2 in G(C_1)$ si eredita m?

Soluzione

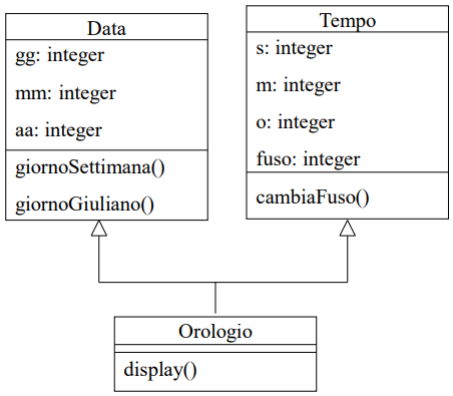
1. Passo: si determina catena di antenati di C_1
2. Passo: si ricerca la prima occorrenza della (ri-)definizione di m a partire dall'estremità C_1 della catena.

3.5.9 Ereditarietà multipla

Una classe può avere più superclassi. In questo caso si parla di ereditarietà **multipla**.

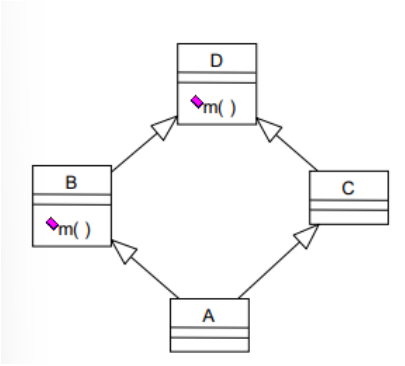
Esempio : La classe *Orologio* necessita dei servizi messi a disposizione sia da *Data* che da *Tempo*. In più ne implementa degli altri.

Il grafo di ereditarietà non è più un albero. Data una classe C, G(C) non è più una catena ma un grafo aciclico orientato. L'ordine fra le classi in G(C) è parziale.



Anche nell'ereditarietà multipla un metodo può essere ridefinito in diverse classi e si pone il seguente

Problema : Sia dato un metodo m, eventualmente ereditato, della classe C_1. Da quale classe $C_2 in G(C_1)$ si eredita m?



Se C_1=A c'è un conflitto fra le diverse definizioni di m in B e D, entrambe ereditabili. Il conflitto può essere risolto in questo caso. Si considera una qualunque *linearizzazione* del grafo G(C_1). i possono avere due casi:

- $A \to B \to C \to D$
- $A \to C \to B \to D$

In entrambi i casi, B precede D, il che significa che il metodo m è ereditato da B. Il metodo m di B è quello più specifico e maschera quello omonimo di D.
Le due linearizzazioni del grafo $G(C_1)$, precedentemente specificate, non aiutano a scegliere.

Esistono dei criteri euristici per gestire queste situazioni conflittuali:

1. **molteplicità dell'ereditarietà**: nel definire che A deriva dalle due superclassi occorrerà elencarle in un qualche ordine:

   $$A \to B, C \text{ piuttosto che } A \to C, B$$

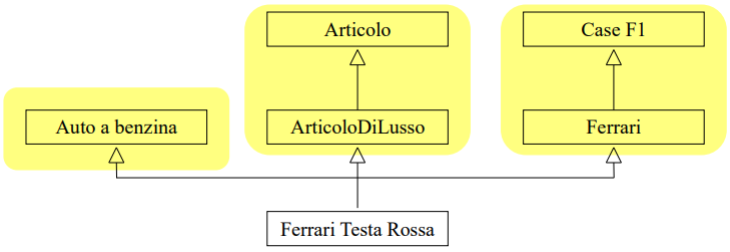
   L'ordine delle classi è utilizzato per preferire una delle due linearizzazioni. Tuttavia questo principio può essere in contraddizione con quello che indica di preferire l'ereditarietà da classi più specifiche. Esempio:

   $$A \to B, C \text{ e } C \to B, D$$

   Se m è definito in B e in C si dovrebbe preferire C in quanto C precede sempre B nelle linearizzazioni:

   $$A \to C \to B \to D$$

2. **modularità**: si può scomporre un grafo di ereditarietà in moduli che corrispondono ai diversi punti di vista sull'oggetto.
   Per esempio, i diversi moduli corrispondono ai diversi punti di vista (tipo alimentazione, tipo articolo, casa costruttrice).



Nelle linearizzazioni non ci sono relazioni di ereditarietà tra classi appartenenti a moduli diversi. Anche in queto caso potrebbero esserci contraddizioni da risolvere specificando l'ordinamento dei moduli.

Quste contraddizioni mettono in evidenza le difficoltà che si incontrano a dare una semantica a un metodo quando si permette l'ereditarietà multipla.
I diversi principi per gestire le situazioni conflittuali non sono universalmente accettati. La risoluzione dei conflitti, in realtà, non può essere efficace se non prende in considerazione le conoscenze specifiche legate all'applicazione.
Per questo molti sconsigliano l'ereditarietà multipla, in qunato i benefici sono pochi mentre i problemi legati ad una semantica ben definita sono molti.

3.5.10 Visibilità protetta

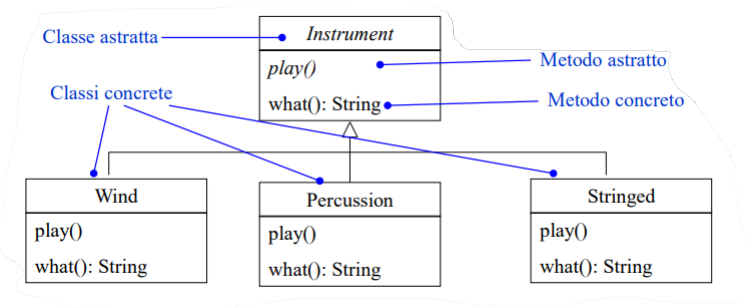
La relazione di ereditarietà introduce un ulteriore livello di visibilità, la **visibilità protetta (protected)**. Un attributo o un metodo con visibilità protetta è visibile solamente all'interno della classe stessa, delle classi del package e delle classi discendenti (anche in altri package).

3.6 Classi astratte

Se in programmazione object-oriented non può esistere un oggetto senza che sia stata creata la classe di appartenenza (come invece accade in ADA), è invece possibile che esistano classi per le quali non è possibile generare delle istanze (**classi astratte**).
Una classe astratta può essere una classe non completamente specificata. In particolare, non è definito il metodo corrispondente ad una operazione (i metodo è astratto).
Una classe astratta è una classe che non può essere istanziata perchè il comportamento dei suoi oggetti non sarebbe completamente definito (ha senso come superclasse di una classe concreta istanziabile).
La realizzazione di ereditarietà riguarda anche le classi astratte.

3.6.1 Notazione

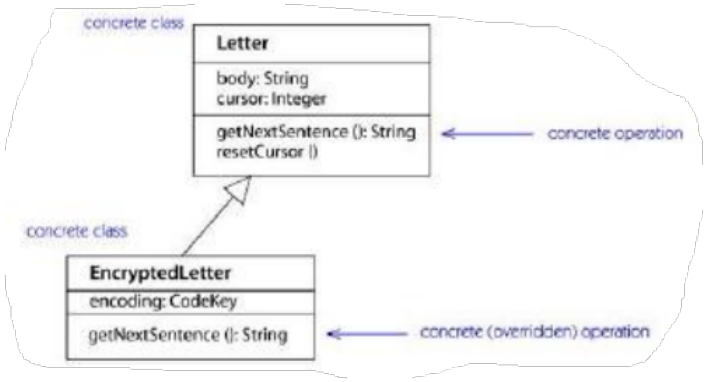
Una classe o un metodo astratto è indicato con il nome in italico. Nel metodo astratto viene eridata solo la segnatura e non ha nè semantica nè implementazione. Per esempio:



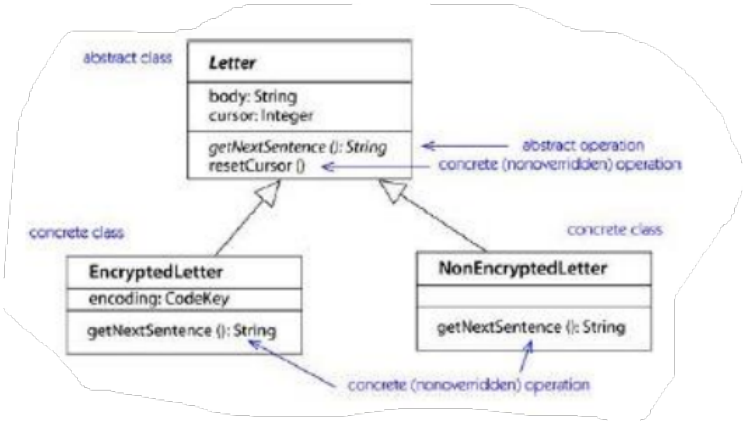
La classe strumenti musicali è astratta perchè non possiamo definire il metodo *play*. Esso è imolementato nelle sottoclassi degli strumenti a fiato, a percussione e nei cordati. Il metodo what è invece concreto (è implementato) e restituisce il nome della classe. Esso è ridefinito in ogni classe.

Le classi astratte sono strumenti per fattorizzare proprietà comuni tra classi simili e poterle organizzare in una gerarchia di ereditarietà. Non potremo mai creare oggetti a partire da una classe astratta, ma possiamo servircene per dare una radice comune a un insieme di classi che condividono le stesse proprietà e poter quindi sfruttare il polimorfismo di inclusione e il binding dinamico (di cui si parlerà in seguito).
Le classi astratte fungono da **serbatoi di ereditarietà**.

Esempio:



La classe Letter ha un metodo che restituisce la frase successiva da leggere e l'operazione resetCursor che riporta all'inizio del testo. La sottoclasse EncriptedLetter rappresenta una lettera crittografata. Il metodo getNextSentence è stato sovrascritto perché il testo dev'essere decrittato prima di essere restituito. L'implementazione dell'operazione getNextSentence è completamente diversa da quella ereditata. PROBLEMA: Il progettista di EncriptedLetter non ha alcuna informazione per comprendere quale metodo di Letter dev'essere sovrascritto.

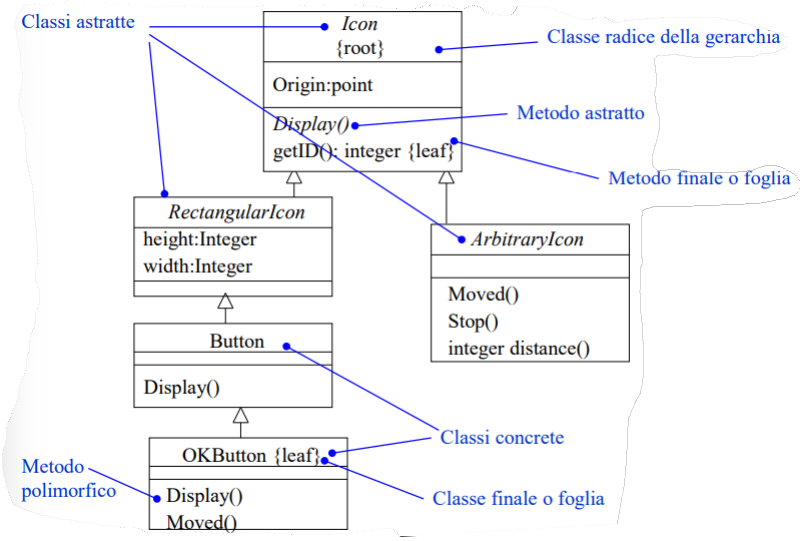


Se il metodo *getNextSentence* di Letter fosse astratto (e dunque fosse astratta anche la classe *Letter*) il progettista verrebbe informato della necessità di ridefinire il metodo. La classe astratta *Letter* fattorizza i metodi concreti comuni alle sue sottoclassi, come resetCursor.

3.7 Classi Finali

Una classe è detta **finale (final)** o foglia (leaf) quando non può essere ulteriormente specializzata, e quindi non può essere modificata.
Si definisce una classe foglia quando il comportamento della classe dev'essere ben stabilito per ragioni di **affidabilità**.
La dichiarazione di una classe foglia permette anche la generazione di codice ottimizzato in quanto facilita l'espansione in linea del codice (impossibile nel caso di metodi sovrascrivibili nelle sottoclassi).
Non ha senso stabilire una classe astratta foglia.

3.8 Class Diagram in dettaglio



Un metodo concreto leaf non può avere override, mentre un metodo astratto leaf da errore.
Mettere leaf i metodi di una classe leaf è inutile ma non sbagliato.
La classe astratta *ArbitraryIcon* non ha senso poichè non ha discendenti, ma non è sbagliata.

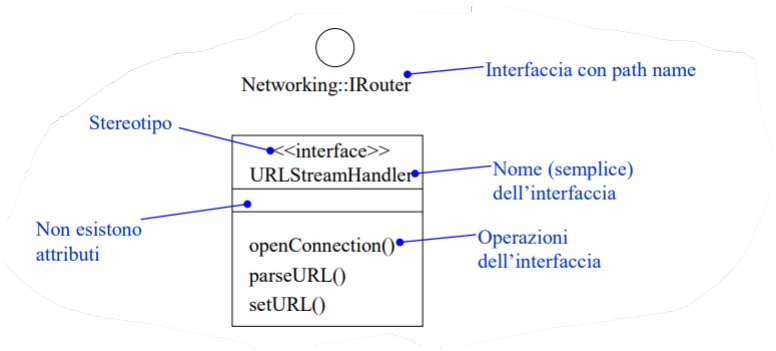
3.9 Interfacce

Una interfaccia è la descrizione del comportamento degli oggetti senza specificarne una implementazione. Essa è una collezione di operazioni, cioè di servizi che possono essere richiesti, priva di informazioni sulle implementazioni dei servizi (i metodi).

Similmente ad una classe, un'interfaccia può avere un qualsiasi numero di operazioni. Diversamente invece, un'interfaccia non specifica la struttura (non sono inclusi attributi, se non statici) e non fornisce un'implementazione (non sono specificati i metodi che implementano le operazioni). Di conseguenza le interfacce non possono essere istanziate.

Una interfaccia è simile ad una classe astratta i cui metodi son tutti astratti e non dispone di attributi.

Nella sua rappresentazione più generale, un'interfaccia è rappresentata mediante l'utilizzo di un cerchio, sopprimendo la visualizzazione delle operazioni. Tuttavia, se lo si ritiene importante per la comprensione del modello, è possibile modellare un'interfaccia come una classe stereotipata



Una o più classi possono realizzare/implementare le operazioni indicate in una interfaccia. La relazione che si stabilisce fra una interfaccia e una classe che la implementa è detta relazione di realizzazione. In UML è indicata con una freccia tratteggiata.

La relazione di realizzazione si presenta come un valido strumento per scindere la specifica di un "contratto" (cosa una classe deve implementare) e la sua implementazione (come si rendono i servizi del contratto).

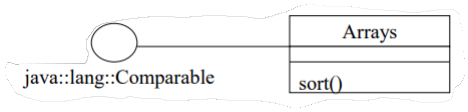
Esempio l'interfaccia *java.lang.Comparable* specifica una sola operazione.

public int compareTo(Object o)

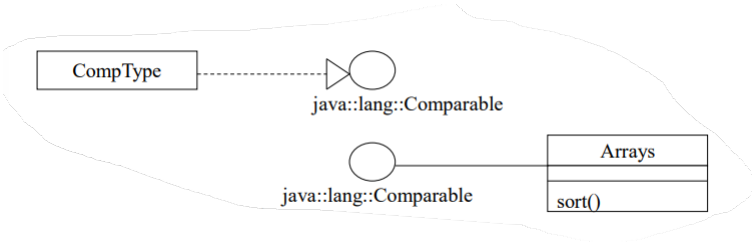
che prende un oggetto generico *Object* come argomento e restituisce un valore negativo se l'argomento è più piccolo dell'oggetto corrente, zero se è uguale e un valore positivo se è maggiore.
Una qualunque classe che implementa questa interfaccia deve fornire una implementazione (un metodo) per *compareTo*.

Le interfacce servono a disaccoppiare la definizione delle operazioni dalla loro implementazione. Per poter usare un certo oggetto è sufficiente conoscere la sua interfaccia: non serve conoscere l'implementazione.

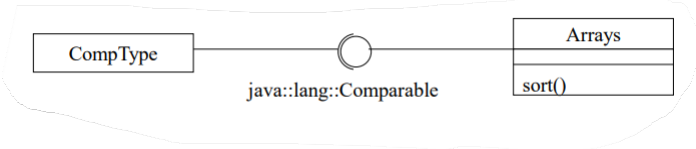
Esempio Il metodo sort della classe Arrays, che ordina un array di oggetti utilizzando l'algoritmo di ordinamento per fusioni successive (mergesort), necessita solo di sapere che gli elementi dell'array offrono il servizio compareTo, in modo da poterli confrontare. Non serve sapere "come" si realizza il servizio, cioè come sono effettivamente confrontati gli oggetti. Si crea così una dipendenza di implementazione fra la classe Arrays e l'interfaccia Comparable.



UML permette di rappresentare in modo compatto le seguenti relazioni:

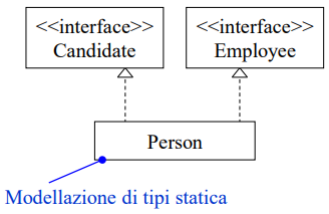


attraverso la notazione:



Anche le interfacce possono ereditare da altre interfacce. Poichè non ci sono implementazioni, la realizzazione di ereditarietà è naturalmente una relazione di generalizzazione "is_a".

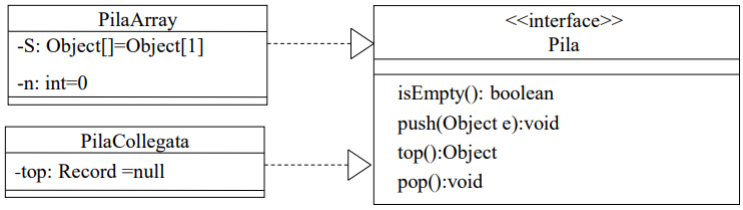
Le interfacce sono realizzabili in java e non presentano il problema dell'ereditarietà multipla con le operazioni comuni nelle superclassi. E' valida solo l'ereditarietà per estensione.
Poichè non si considerano le implementazioni delle operazioni, l'ereditarieta' multipla su interfacce non pone problemi di conflitto di realizzazione.
Per questo alcuni linguaggi di programmazione, come Java, permettono l'ereditarietà singola sulle classi (in quanto specificano anche le implementazioni) mentre permettono l'ereditarietà multipla sulle interfacce.
Questo distingue le classi astratte (per le quali l'ereditarietà multipla può porre dei problemi riguardo ai metodi implementati) dalle interfacce. Ad esempio:



Poiché non possono sorgere problemi di conflitto di realizzazione, è permesso a una classe di realizzare più interfacce, per di più non correlate da una relazione di generalizzazione. In questo modo tutto il codice scritto per Candidate e Employee può essere utilizzato su oggetti di classe Person.

Infine, più classi possono implementare la stessa interfaccia. E' questo il caso di

una interfaccia definita per dati astratti molto utilizzati nello sviluppo del software, come pile, code, liste, alberi e grafi, per le quali sono possibili molteplici realizzazioni che variano per l'efficienza degli operatori. Ad esempio:



Il codice applicativo che intende fare uso di una pila dichiarerà delle variabili di tipo Pila, svincolandosi dalla specifica realizzazione. Questa sarà specificata solo al momento della inizializzazione della variabile. Si garantisce così una forte invarianza ai cambiamenti delle realizzazioni di una pila.

3.10 Aggregazione di oggetti

L'ereditarietà offre molti vantaggi, ma non tutti gli oggetti si ottengono bene **derivandoli** da altri oggetti. Spesso un oggetto è ottenuto **aggregando** altri oggetti.

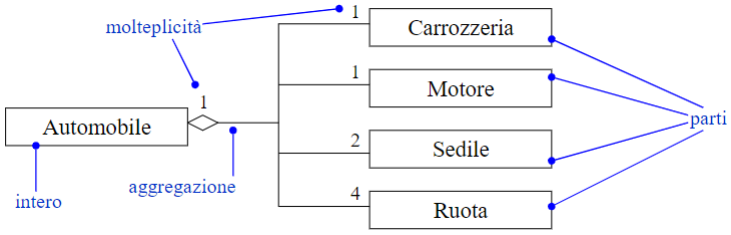
Esempio Ricorrendo all'ereditarietà multipla è allettante definire una classe Automobile partendo dalle classi Carrozzeria, Sedile, Ruota e Motore. Si tratta di un errore concettuale poiché l'ereditarietà multipla permette di definire due oggetti per fusione e non per composizione. Un'automobile è composta da una carrozzeria, da un motore, da sedili e da ruote; ma il suo comportamento non è in alcun caso l'unione dei comportamenti di queste differenti parti.

Nell'esempio specifico, sembrerebbe che il ricorso all'ereditarietà di implementazione sia la soluzione. Essa permette il riuso della sola implementazione (attributi e metodi) ma non dei comportamenti.
Ma neanche questa forma di ereditarietà è adatta allo scopo. Infatti, ereditando dalla classe Ruota si ha la possibilità di rappresentare e manipolare una sola ruota, mentre un'automobile ha quattro ruote.

Una **composizione di oggetti** può essere rappresentata permettendo alle variabili di istanza di una classe di puntare a oggetti di altre classi. Si possono stabilire legami con più istanze di una classe che descrive un componente. La relazione che si stabilisce in questo modo fra le classi è detta di **aggregazione** o **composizione** (o relazione "has_a"). Essa è quindi un'altra possibile relazione fra le classi, diversa dall'ereditarietà e dalla generalizzazione (o "is_a"). Una classe A si dice in relazione di aggregazione con una classe B quando alcune istanze di B contribuiscono a formare una parte delle isstanze di A. L'aggregazione è, come l'ereditarietà, una relazione asimmetrica.

In UML, l'aggregazione è resa mediante una linea piena e un piccolo rombo dalla parte del contenitore. L'aggregazione vale solo tra classi, non tra interfacce. La molteplicità consente di indicare quanti oggetti possono essere aggregati.

Esempio



L'suo dell'aggregazione è consigliato nelle seguenti stituazioni:

- Contenimento fisico: la pagina di un libro
- Appartenenza: il giocatore di una squadra di calcio
- Composizione funzionale: le ruote di una automobile

Si deve osservare che **l'aggregazione non implica una dipendenza esistenziale**: un'automobile può essere distrutta ma alcune sue parti possono essere riutilizzate. Una squadra di calcio può fallire, ma i suoi giocatori non vengono "soppressi".

3.11 Composizione di oggetti

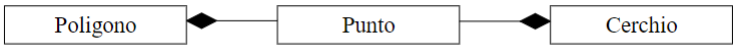
Le aggregazioni sono associazioni **deboli** fra parti e intero. Questo significa che le parti possono esistere senza l'intero.

Un'associazione **forte** fra parti e intero è detta **composizione** e si rappresenta in UML mediante un rombo pieno. La composizione comporta una **dipendenza esistenziale**, in quanto le parti non esistono senza il contenitore.
Ciò presuppone che la crezione e la distruzione delle parti avvengano nel contenitore e che i componenti non siano parti di altri oggetti.

Esempi In questi esempi, la vita delle parti che compongono l'intero dipende dalla vita dell'intero stesso:

1. i dipartimenti non esisterebbero se non esistesse la società;
2. il tempo di vita dei diversi oggetti grafici di interazione dipende da quello della finestra che li contiene.

Esempio



Un'istanza di Punto può essere parte di un poligono oppure il centro di un cerchio, ma non entrambe le cose.

Regola di non condivisione Benchè una classe possa essere componente di molte altre classi, **ogni sua istanza può essere componente di un solo oggetto**. Un diagramma delle classi può mostrare più classi di potenziali possessori di oggetti componenti, ma ogni istanza di componente deve appartenere ad un solo oggetto possessore. Questa regola è caratterizzante della composizione.

La composizione fra classi stabilita in fase di progettazione offre delle informazioni importanti al programmatore che andrà a implementare le classi.
Ad esempio, se il programmatore utilizzerà il C++ farà si che gli oggetti componenti finiscano nello stack e non nello heap (non userà puntatori/new).
Se utilizza il Java, che pone tutto nello heap, si preoccuperà di allocare i componenti nella classe contenitore e di non passare mai all'esterno della classe il riferimento ai componenti, in modo da impedirne la condivisione con oggetti di altre classi (il componente sarà privato e non ci sarà alcun metodo che restituisce il suo riferimento).

3.12 Ereditarietà vs aggregazione

In molti casi è possibile associare due classi mediante ereditarietà o aggregazione/-composizione e la scelta non è immediata come negli esempi visti precedentemente.
Per esempio si supponga di disporre di una classe Data e di una classe Tempo, che rappresentano e manipolano, rispettivamente, date e istanti di tempo. Volendo definire una nuova classe Orologio per rappresentare sia le date che gli istanti di tempo, si può:

1. derivare Orologio da Data e da Tempo, se l'ereditarietà multipla è permessa;

2. comporre Orologio con le due classi.

Sono anche possibili soluzioni ibride, come derivare Orologio da Data e comporlo con Tempo. Le differenze tra la scelta 1) e la 2) sono:

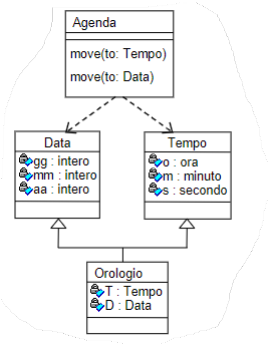
- Nell'ereditarietà, ogni oggetto *o* della classe *Orologio* contiene tutti i campi definiti nelle classi *Data* e *Tempo* e può accedervi direttamente se non sono privati.

- Nell'aggregazione/composizione, ogni oggetto *o* della classe *Orologio* contiene due campi di tipo *Data* e *Tempo* rispettivamente. Per accedere all'informazione sul giorno occorrerà invocare un metodo (per esempio *day()*) sull'oggetto *D*.
- Se l'**ereditarietà** corrisponde a una relazione di generalizzazione "is_a" **vale il polimorfismo di inclusione**, cioè ogni oggetto di *Orologio* è utilizzabile come istanza di *Data* e di *Tempo*, **mentre questo non è vero nel caso della composizione**.

Le scelte di progetto sono totalmente diverse.

- L'ereditarietà permette di poter "dimenticare" il fatto che *gg* è definito nella classe *Data* e di utilizzarlo direttamente come campo di *Orologio* (sempre che sia definito come *protected* e non come *private*). Questa caratteristica della derivazione è particolarmente utile se si vogliono utilizzare più derivazioni successive e creare vere e proprie gerarchie di classi. In questo caso non è necessario conoscere a quale livello della gerarchia è definito un dato campo, ma è possibile utilizzarlo direttamente come se fosse un elemento della classe stessa.
- L'ereditarietà permette di riutilizzare tutti i metodi delle varie classi che operano su oggetti delle classi *Data* e *Tempo*.

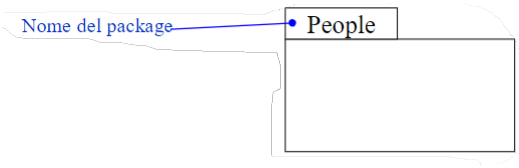
Esempio *Agenda* ha due metodi che possono operare anche su istanze di *Orologio*. Il codice dei metodi polimorfi *move* è riutilizzabile per istanze di *Orologio*.



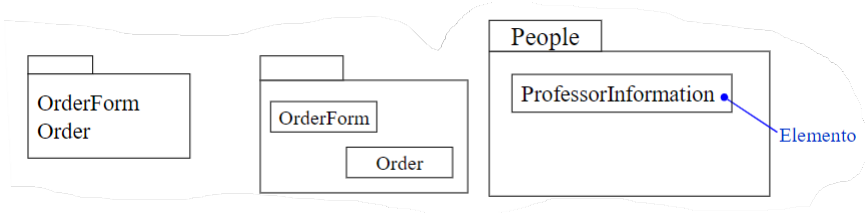
Il meccanismo di aggregazione/composizione è generalmente usato quando si vogliono utilizzare i servizi di una classe predefinita ma **non la sua interfaccia**. L'ereditarietà di implementazione, qualora non dovesse essere permessa da un linguaggio di programmazione, potrebbe essere resa da una relazione di aggregazione/composizione.

3.13 Raggruppare classi

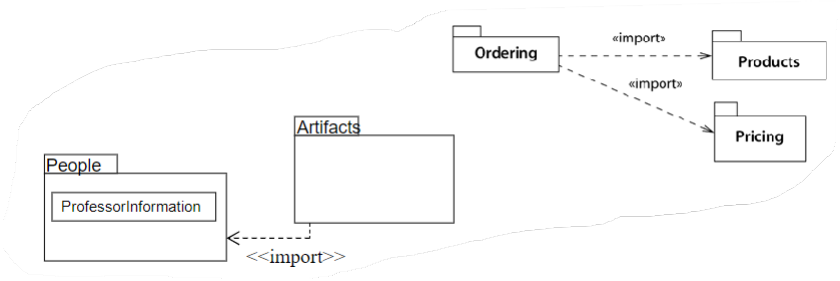
La mole di classi aumenta in modo considerevole all'aumentare della complessità del sistema da modellare. E' importante organizzare tali classi in **gruppi** separati al fine di rendere più facilmente individuabili e accessivili le singole parti che compongono il nostro sistema.
I **package** sono un meccanismo generale per organizzare le classi in gruppi.
In UML un package è rappresentato come segue:



In genere i package si usano per riunire classi, ma nella notazione UML essi possono includere qualsiasi costrutto UML e possono essere persino eterogenei. E' consenrtito mostrare il contenuto di un package sia testualmente sia graficamente:



Un package definisce un **namespace** (spazio degli identificatori) per i suoi elementi. Questo significa che ogni classe di un package di classi deve avere un nome distinto all'interno del package che la racchiude. Il nome completo (o **qualificato**) della classe sarà ottenuto indicato tramite la notazione nomepackage::nomeclasse.
In generale é possibile avere nomi uguali per elementi di tipo differente. Ad esempio è possibile avere una classe o interfaccia Cliente e una package Cliente, ma non due classi Cliente, due interfacce Cliente o una classe Cliente e una Interfaccia Cliente. É possibile avere nomi uguali per elementi della stessa specie se sono inseriti in package differenti.
Per evitare la necessità di utilizzare nomi qualificati, un package può **importare** gli elementi o il contenuto di un altro package nel proprio namespace. Un elemento nel package che importa può quindi riferirsi a un elemento importato come se esso fosse definito direttaente nel package (localmente). L'import di un package viene ottenuto stereotipando la relazione di indipendenza:



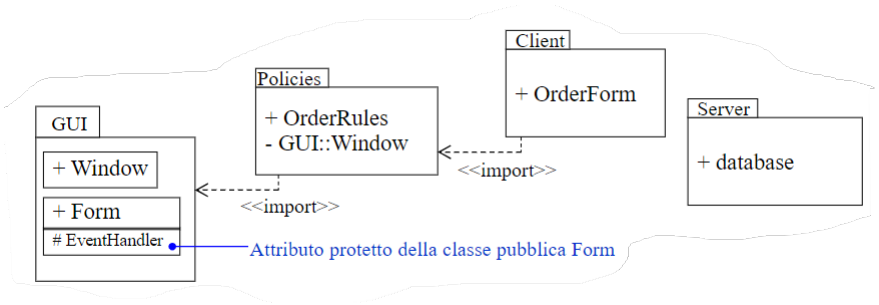
Nota bene:

- La relazione di importazione non è transitiva. Se A importa B e B importa C, allora A può usare (senza qualificarli) gli elementi di B e B può usare (senza qualificarli) gli pubblici di C, ma non è detto che A possa usare (senza qualificarli) gli elementi di C (a meno che A non importi anche C).
- Se c'è conflitto di nomi in due elementi importati, nessuno dei due elementi è aggiunto al namespace.
- Se il nome di un elemento importato è in conflitto con il nome di un elemento definito localmente (internamente) a un package, il nome dell'elemento interno ha precedenza sul nome importato che non viene aggiunto al namespace (sarà sempre raggiungibile mediante qualificazione).

I package possono essere innestati senza alcun limite di profondità . Un package innestato ha accesso a tutti gli elementi contenuti direttamente nei package esterni (a qualunque livello di annidamento) senza necessità di importazione. Si può anche specificare la visibilità degli elementi di un package:

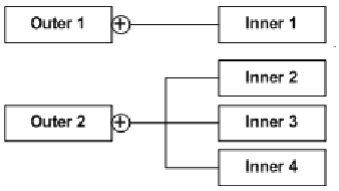
- **public**: visibili ad altri elementi nel package stesso, a uno dei package innestati o a un package che li importano.
- **private**: visibili solo all'interno del package stesso.

La relazione di importazione può specificare la visibilità di ciò che è importato. Se è pubblica, l'elemento importato è visibilie a qualunque altro elemento che può vedere il package che importa. Se è privata, l'elemento importato non è visibile al di fuori del package che importa.
Per esempio:



3.14 Classi interne

Una classe interna (**inner class**) è una classe la quale dichiarazione si trova all'interno di un'altra classe ospite (**top level class**). Le classi interne sono identificate da un nome al pari delle classi top level. **Una classe interna può essere privata**. In tal caso non è visibile all'esterno della classe ospite. (In java, una classe top level non può mai essere privata). La inner class può **accedere a tutti i metodi e i campi della classe ospitante**, mentre la classe ospitante può vedere solo la parte pubblica della inner class. **Un oggetto di classe inner non può esistere se non esiste un oggetto della classe ospitante**. Una classe interna non può avere campi statici.
UML non ha una notazione standard per le classi interne. Si suggerisce la seguente notazione:



3.15 Polimorfismo

Il polimorfismo è una caratteristica chiave della programmazione orientata agli oggetti (OOP) che permette di trattare oggetti di classi diverse attraverso un'interfaccia comune. Di seguito approfondiamo i due tipi principali di polimorfismo:

- **Polimorfismo parametrico**: Questo tipo di polimorfismo si verifica quando una funzione o un metodo può operare con qualsiasi tipo di dato, assumendo che questi tipi condividano una struttura comune. Un esempio classico è una funzione che può operare sia su una lista di interi che su una lista di stringhe. In linguaggi di programmazione come Haskell o Java, questo è ottenuto attraverso l'uso di tipi generici. I tipi generici permettono di scrivere una funzione, una classe o un'interfaccia una sola volta e di utilizzarla con diversi tipi di dati, mantenendo la sicurezza di tipo durante la compilazione.

- **Polimorfismo ad hoc**: Questo tipo di polimorfismo permette di definire funzioni che sembrano lavorare su tipi differenti ma che in realtà sono implementazioni separate per ciascun tipo. Si manifesta principalmente in due forme:

  – *Sovraccarico di funzioni*: dove più funzioni con lo stesso nome esistono con parametri differenti. Ad esempio, una funzione `print()` potrebbe essere definita per stampare sia interi sia stringhe, con un'implementazione specifica per ciascun tipo.

  - *Specializzazione di funzione*: dove una funzione per un tipo di dati generico viene adattata per gestire tipi di dati più specifici. Questo può comportare un'implementazione totalmente differente che è ottimizzata per un particolare tipo di dati.

Entrambe le forme di polimorfismo migliorano la flessibilità e la riusabilità del codice. Nel polimorfismo parametrico, la flessibilità è ottenuta tramite l'uso di tipi di dati astratti che possono essere sostituiti con tipi concreti al momento dell'uso. Nel polimorfismo ad hoc, la riusabilità è migliorata permettendo la stessa interfaccia di essere utilizzata per diversi tipi concreti, ciascuno con una propria implementazione specifica. Se dovessimo raffinare la classificazione dei diversi meccanismi di polimorfismo adotteremmo la seguente struttura:

- Polimorfismo Universale
  - Parametrico
  - Inclusione
- Polimorfismo ad hoc
  - Overloading
  - Coercizione

Questa classificazione introduce la forma di **polimorfismo per inclusione** al fine di modellare concetti di sottotipo e di ereditarietà. Il polimorfismo per inclusione e parametrico sono classificati come due sottocategorie del **polimorfismo Universale** che è posto in contrasto al polimorfismo ad hoc. L'idea del polimorfismo universale è qulla di poter operare su un numero infinito di tipi, a patto che essi rispettino alcuni vincoli. Nel dettaglio, elenchiamo le caratteristiche dei due tipi di polimorfismo:

- Polimorfismo Universale
  - è su un numero potenzialmente illimitato di tipi
  - i diversi morfismi sono generati automaticamente
  - c'è una base unificante, comune a tutti i diversi morfismi che può assumere l'entità polimorfa
- Polimorfismo ad hoc
  - è su un numero finito di tipi, spesso pochissimi
  - i diversi morfismi sono generati in modo manuale o semimanuale
  - non c'è una base comune a tutti i morfismi, al di la delle intenzioni del progettista (l'uniformità è un caso, non la regola)

3.15.1 Coercizione

E' il meccanismo di conversione implicita operata da un compilatore per applicare un operatore definito per oggetti di tipo T1 a oggetti di tipo T2. Per esempio in $3.14 + 5$ l'operatore $+$ è definito per valori reali, ma lo si può usare su un insieme di tipi più grande di quello per il quale è stato definito. Senza coercizione avremmo errore di tipo.
Altri esempi di coercizione sono, in Java per esempio, l'**autoboxing** di un *int* in un *Integer* e l'**unboxing** di un *Integer* in un *int*.
Le coercizioni possono essere stabilite staticamente, inserendole automaticamente fra gli argomenti e le funzioni al momento della compilazione, oppure potrebbero essere determinate da test al run-time sugli argomenti. La coercizione è la forma di polimorfismo più semplice. Essa opera a livello semantico cambiando la rappresentazione del dato.

3.15.2 Overloading

Si ha polimorfismo per **overloading** quando si usa lo stesso identificatore per metodi differenti e si ricorre a informazioni di contesto per decidere quale metodo è denotato da una particolare occorrenza dell'identificatore. La disambiguaizone necessaria per una corretta compilazione si basa sul tipo degli argomenti del metodo o sulla classe dell'oggetto a cui si richiede il servizio. Possiamo immaginare che una pre-compilazione del programma potrebbe disambiguare ed eliminare l'overloading dando nomi differenti a metodi differenti. L'overloading è giusto una conveniente abbreviazione sintattica. E' presente nella maggior parte dei linguaggi. Per esempio:

```
// C++'s user-defined overloading of the + operator:
class Rational {
public: Rational(double);
const Rational& operator + (const Rational& other);
...
};
```

Overloading di operatori

```
// C++'s user-defined overloading of the function name max:
double max(double d1, double d2);
char max(char c1, char c2);
char *max(char *s1, char *s2);
const char *max(const char *s1, const char *s2);
```

Overloading di funzioni

L'overloading può efficacemente integrare con la coercizione.
Nel paradigma a oggetti si ha overloading anche nel caso di funzioni con medesimo nome ma definite in classi non correlate gerarchicamente.

3.15.3 Polimorfismo parametrico

Nel poliformismo parametrico, una funzione polimorfa ha un parametro di tipo esplicito o implicito, che determina il tipo dell'argomento per ciascuna applicazione della funzione. Le funzioni che esibiscono il polimorfismo parametrico sono dette anche **funzioni generiche**. Una funzione generica può lavorare su argomenti di molti tipi, generalmente esibendo lo stesso comportamento indipendentemente dal tipo di argomento. Per esempio:

$$\text{length(x)} = \text{if (x=nil) then 0 else (1+length(tail(x)))}$$

La funzione **length** ha lo stesso comportamento, indipendentemente che si tratti di liste di interi, di reali, di liste ecc.

3.15.4 Polimorfismo per inclusione

Nasce da una relazione di inclusione fra insiemi di valori. E' tipico dei sottotipi, tuttavia non è necesssariamente legato ai sottotioi. Si ha polimorfismo per inclusione, nella progrmammazione ad oggetti, se un oggetto appartiene a una classe e a tutte le sue superclassi. Esso si manifesta in almeno due modi:

- si può assegnare un oggetto di una qualsiasi sottoclasse di una classe C a una variabile definita di classe C
- Una funzione che opera su un oggetto di classe C può essere applicata anche a oggetti di classe C', sottoclasse di C.

Questi due utilizzi del polimorfismo di inclusione non sono molto distanti dall'utilizzo che permette il concetto di sottotipo anche nella programmazione imperativa. L'utilizzo più interessante si ha quando le invocazioni dei metodi su oggetti di classi diverse, ma gerarchicamente correlate, produce un comportamento differente, anche se la definizione della funzione è unica. Ciò dipende dal **tipo di legame statico/dinamico** fra identificatore di funzione e relativa realizzazione.

Nella maggior parte dei linguaggi di programmazione la visibilità degli identificatori e dei legami (binding) dei nomi alle dichiarazioni è determinata a compile-time. Si parla di **ambito d'azione statico** (static scope). Nell'ambito d'azione **dinamico** (dynamic scope), il legame fra l'uso di un identificatore e la sua dichiarazione dipende dall'ordine di esecuzione, e così è differito a run-time.

Esempio In Pascal:

```
program dynamic(input,output);
var x: integer;
        procedure A;
        begin
                ...;
                write(x);
```

```
                ...;
        end; {A}
        procedure B;
        var x: real;
        begin
                ...;
                A;
                ...;
        end; {B}

begin
        ...;
        B;
        ...;
        A;
        ...;
end. {dynamic}
```

Poiché il Pascal adotta la regola dell'ambito statico, l'uso della variabile x in A è legato alla variabile intera x nel programma principale. Questo permette di tradurre la write(X) semplicemente in una chiamata a una funzione di libreria di I/O per la scrittura degli interi.
Tuttavia se il legame nome-dichiarazione fosse dinamico, l'uso di x in A sarebbe vincolato alla dichiarazione di x più recente. Pertanto, quando la procedura A è chiamata dalla procedura B, l'uso di x in A viene vincolato alla dichiarazione della variabile reale x nella procedura B, mentre quando A è chiamata nel programma principale, l'uso di x sarebbe vincolato alla dichiarazione della variabile intera x nel programma principale.
Con un dynamic-binding, la traduzione della chiamata della write(x) può essere determinata solo al run-time.
Questo non vuol dire che il controllo di tipo non possa essere effettuato, ma solo che viene ritardato al momento dell'esecuzione, quando è noto il tipo al quale x è vincolato.
Si osservi che con lo static-binding il legame dei nomi ai tipi (name-type binding) è anch'esso fissato al momento della compilazione.

3.15.5 Legame dinamico e polimorfismo di inclusione

Nella progrmammazione oo il leame fra nome di funzione e sua realizzazione può essere determinato a run-time. Si consideri il seguente esempio in C++:

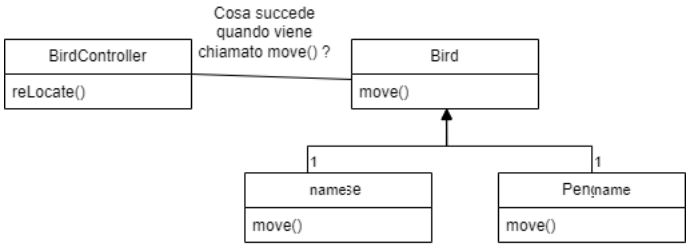
```
class A{
        public:
                virtual void f() {cout << "A";}
                ...
};
class B: public A{
```

```
        public:
                void f() {cout << "B";} //override
                ...
};
class Tester{
        public:
                void test(A a) {a.f()...}
};

...
obj1 = new B();
obj2 = new Tester();
obj2.test(obj1);
...
```

Il comportamento della funzione obj2.test(obj1) dipende da quando si effettua il legame fra l'identificatore di funzione f in test e la sua implementazione. In c++ è possibile definire la funzione f come **virtuale**, cioè il legame dev essere **dinamico**, di conseguenza verrà stampata una B. Diversamente, il legame sarebbe **statico** e verrebbe stampata una A. La selezione del metodo, nel caso di funzioni virtuali, avviene a run-time.

Esempio



Il legame dinamico ci permette di far evolvere il programma aggiungendo nuove sottoclassi di Bird e specificando il codice di move() senza che si modifichi il codice di reLocate() per operare su nuove istanze di nuove sottoclassi di Bird.

Combinando polimorfismo di inclusione con legame dinamico si hanno vantaggi di **estensibilità** di codice.

C++ è un esempio di linguaggio dove la tipizzazione è statica ma il legame può essere dinamico (**statically-typed dynamic binding**). Lo stesso dicasi per Java ma in questo linguaggio **il legame dinamico è la regola, non l'eccezione**. Solo nei metodi final o static il legame nomedifunzione - realizzazione è statico.