



PROGETTAZIONE SOFTWARE

By Edoardo



Progettazione Software

ATTORI

Committente

Esperi Domini: persona esperta nell'ambito di utilizzo del software da costruire

Analista

Progettista

Programmatori

Utente finale

Mantenitore

Classificazione delle applicazioni

1) Rispetto al flusso di controllo

- Sequentiali | interesse del corso
- Concorrenti (Simultanei)

• Dipendenti dal tempo: applicazioni con vincoli temporali esatti

2) Rispetto agli elementi di interesse primario

- orientate alla realizzazione di funzioni | interesse del corso
- Orientate alla gestione dei dati
- Orientate al controllo

Ciclo di vita del software

tutta della creazione del software e non del suo "utilizzo reale".

Raccolta requisiti (software e hard ware), analisi dei requisiti, progetto e realizzazione

Poi Verifica e Manutenzione: in caso di problemi si torna alla raccolta Requisiti.

Fattori di qualità

ESTERNE:

Correttezza, Affidabilità, Estendibilità, Riciclabilità

INTERNE:

Strutturazione, Modularità, Comprensibilità.

STRUTTURAZIONE E MODULARITÀ

Il software è strutturato in moduli.



- Principio di unitarietà: ogni modulo identifica un'unica entità concettuale e tutti i relativi concetti **1**
- Porte interfacce: deve comunicare solo con i moduli necessari **2**
- Poca comunicazione: deve scambiare solo le informazioni strettamente necessarie
- Comunicazione chiara: l'informazione scambiata deve essere il più astratta possibile **3**
- Occultamento di informazioni inessenziali: le info. inutili deve gestirle internamente **4**

1 Alto coesione

2 Bassa Accoppiamento

3 Interaccoppiamento esplicito

4 Information Hiding

TIPO DATO ASTRATTO

È un insieme di elementi e operazioni che possono muoversi su di esso.

Precondizioni = quando la funzione può essere eseguita.

Postcondizioni = cosa calcola la funzione

I tipi di dato astratto possono identificare un valore o un oggetto (entità): il secondo può "combinare stato".

Per lavorare sulle entità è più adatto usare le classi.

SCHEMI REAZZATTIVI

La struttura del software si differenzia per la presenza o meno di **SIDE EFFECT** e **CONDIVISIONE DI MEMORIA**.

Nel caso in cui le funzioni non applicano side effect agli oggetti sono dette **funzionali** e sono usate prevalentemente per i dati di tipo "valore".

A seconda del compito e dei dati del programma, la struttura del software può condividere tra più oggetti lo stesso struttura dati. Ciò è preferibile nel corso di operazioni di tipo funzionale e "proibito" se in presenza di side effect (**interferenza**).

SHARING: gli oggetti appartenono a se stessi e quindi vengono usati per identificare garantendo la possibilità di modificare senza modificare l'identità dell'oggetto stesso.

- Astrazione di valore semplice = ogg. immutabili con condivisione memoria e funzionale
- Astrazione di valore collezione = ogg. immutabili con condivisione memoria e funzionale
 - = ogg. mutabili senza condivisione memoria e con side effect
- Astrazione di entità = ogg. mutabili senza condivisione memoria e con side effect.

N.B. clone() va ridefinito solo in astrazione di valore collezione senza condivisione di memoria
hashcode()/equals() va ridefinito solo in astrazione valore semplice e collezione

ANALISI

INPUT = requisiti raccolti

OUTPUT = schema concettuale

⇒ costruire un modello dell'applicazione completo e preciso traducibile in un programma

Lo schema è costituito da:

- DIAGRAMMA delle classi e oggetti.
- DIAGRAMMA ATTIVITÀ: cosa ci发生 con i dati?
- DIAGRAMMA STATI E TRANSIZIONI: descrive il ciclo di vita delle istanze
- DIAGRAMMA DI SPECIFICA

• ORIENTATI AGLI OGGETTI

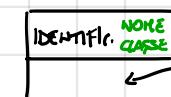
UML

Lingueggi per modellazione concettuale e analisi del programma

Noi lo useremo UML per disegnare il diagramma delle classi.

- OGGETTO UML

Modella un elemento del dominio di analisi.



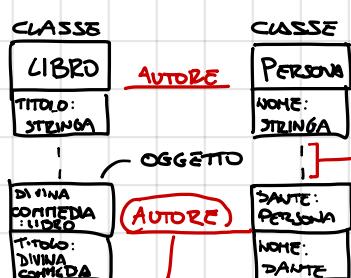
Proprietà (attributi) della classe

Le classi sono livelli INTENSIONALI mentre gli oggetti sono i livelli EXTENSIONALI (\rightarrow NEW)

d'identificatore DEVE essere UNIVOCO e anche se tutti i campi di due oggetti (attributi) sono uguali, l'identificatore sarà differente.

Gli attributi sono funzioni il cui dominio è il tipo della classe mentre il codominio sarà un'estensione di valore (es. stringa, int ecc.).

ASSOCIAZIONE



L'associazione unisce i una proprietà sia dell'oggetto LIBRO che dell'oggetto di PERSONA

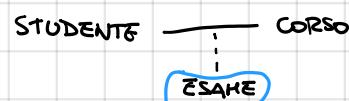
Instance of

A livello estensionale l'associazione diventa link identificato da una coppia.

Tra due classi possono esserci più associazioni. Una relazione non dice il numero di link tra classi.

Le associazioni possono avere degli attributi che non sono di resumo dei termini dell'associazione, ma della coppia che esse formano. Per far ciò bisogna creare la CLASSE ASSOCIAZIONE (permettendo di dare una proprietà ad essa) che è legata alla coppia.

Tale classe è vista come uno funzione dc ha come dominio la coppia e come codominio l'attributo



Ogni associazione può avere un numero minimo e massimo di copie (**MOLTEPLICITÀ**)

ES



Ogni C_2 è legato ad almeno x e massimo y istanze di C_1 mediante associazione A
Ogni C_1 è legato ad almeno w e massimo z istanze di C_2 mediante associazione A

$0..*$ = nessun vincolo (si può omettere)

$0..1$ = massimo 1

$1..1$ = esattamente 1 (obbligatoria)

$1..*$ = minimo 1 (obbligatoria)

Lo moltiplicità può estendersi anche agli attributi.

Un attributo di tipo T della classe C con moltiplicità diversa da 1..1 si dice **MULTIVALORE**

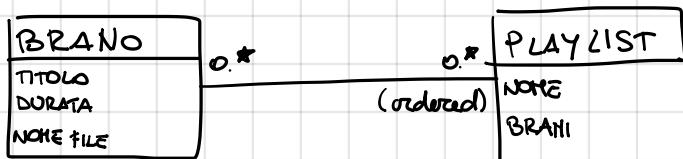
ASSOCIAZIONI N-ARIE: Associazioni tra 3 o più classi

ASSOCIAZIONI ORDINATE

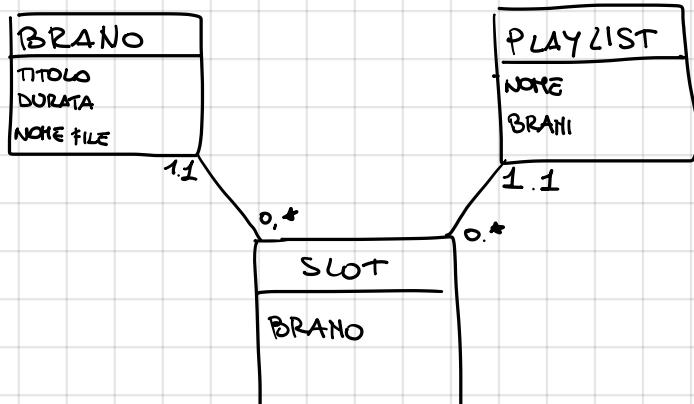


Ordinaz esplicita il fatto che l'associazione persona-gruppo ha "una proprietà" di ordinamento.

ES.



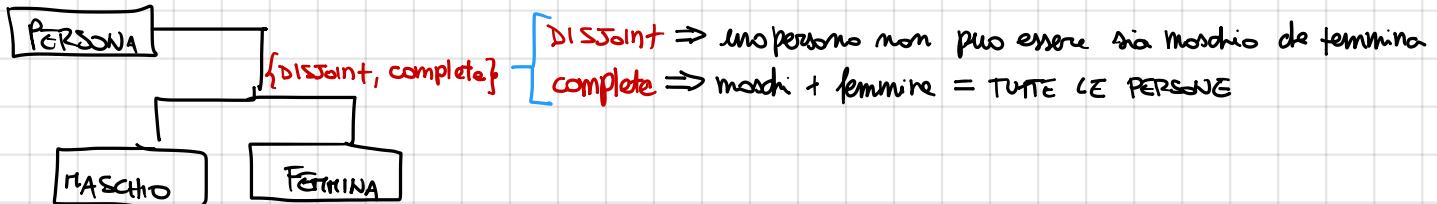
(Cosa senza ripetizioni)



(Cosa con ripetizioni)

Sottoclasse ed Ereditarietà sono simili a Java

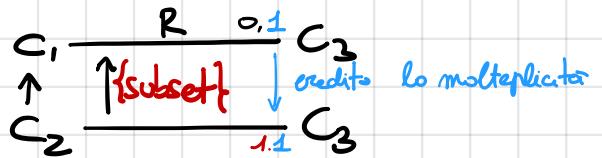
GENERALIZZAZIONE



Se disjoint non è possibile creare una classe che è sottoclasse delle due (Si Se Joint)
Non è possibile creare un'istanza che deriva da due classi senza creare una sottoclasse.

SPECIALIZZAZIONE:

C_2 sottoclasse di C_1 , che è associata a C_3 tramite R
altronde anche C_2 è associato a C_3 tramite R



OPERAZIONI

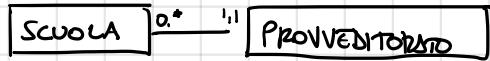
Corrispondono ai metodi di Java



Nella classe si scrive solo la segnatura del metodo.

Dopo lo parte di Analsi, bisogna scegliere lo "tecnologia" da usare per la creazione del programma.
Dopo il "cosa?" (Analsi) il "come?" ovvero lo strumento per la realizzazione: nel nostro caso Java.
L'input dello fase di progetto è l'output della fase di analisi \Rightarrow il grafico delle classi.

Responsabilità sulle associazioni: la classe C ha responsabilità sull'associazione A se per ogni x istanza di C posso eseguire operazioni su tutte le istanze di A a cui x partecipa.



Dato il provveditore vogliamo conoscere le scuole? se no allora PROVVEDITORE non ha responsabilità sull'associazione

\Rightarrow se $!= 0$, è responsabile!

APPARTIENE	SCUOLA	SI
	PROVVEDITORE	NO
		(responsabile dell'altra classe)

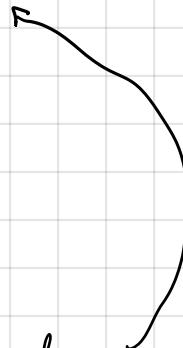
La responsabilità può essere per:

- 1) Moltiplicità
- 2) Operazioni
- 3) Requisiti

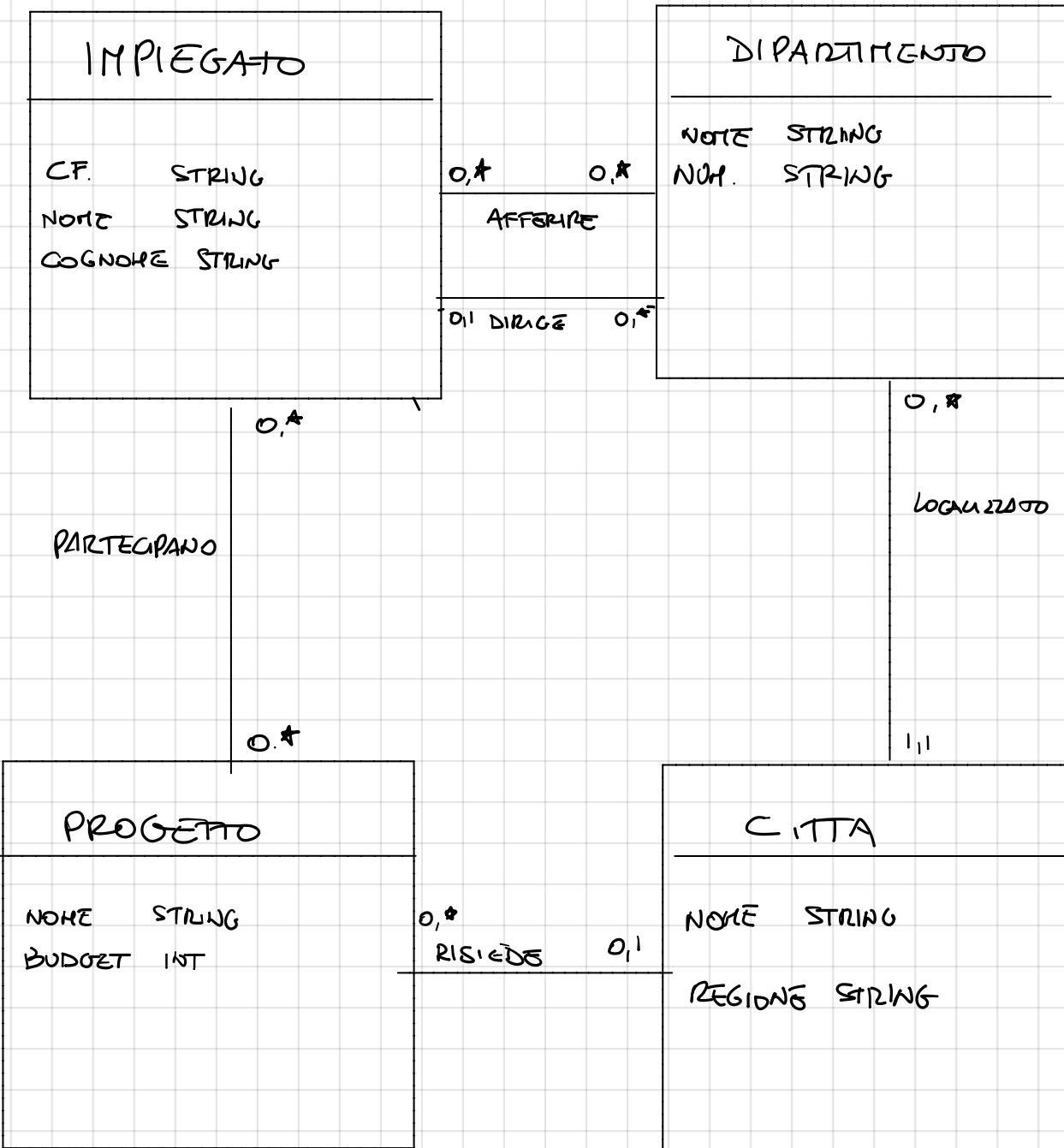
- Moltiplicità se $!= 0, *$

- Operazioni se devo effettuare delle operazioni sulle tuple tra le classi.

- Requisiti se specificato dal testo (o se ho 0..* da entrambe le parti \rightarrow Requisiti "nascosti")

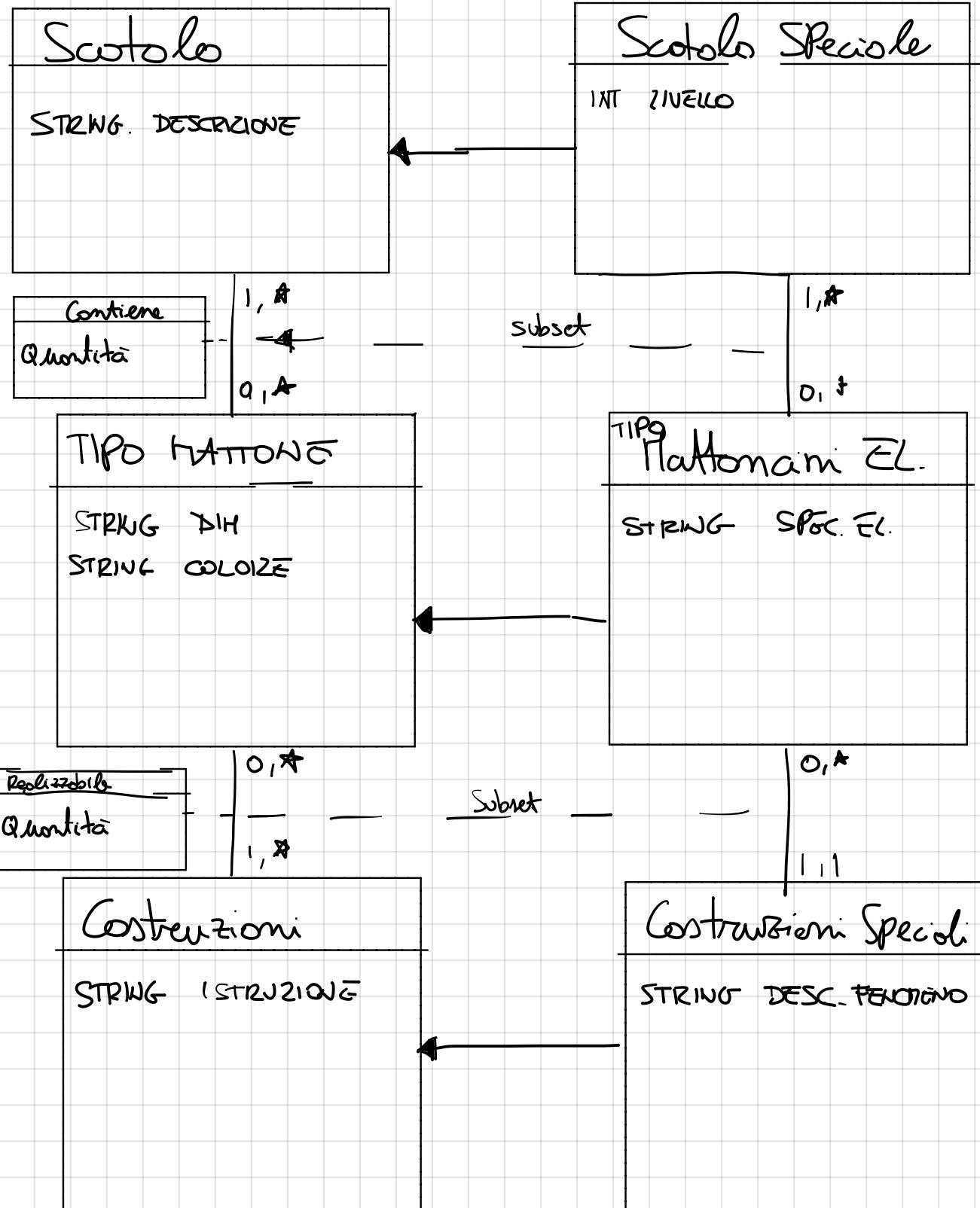


ESERCITAZIONE USO RISTORO



AFFERISCE	<u>IMPiegato</u>	SI'
	<u>DIPARTIMENTO</u>	NO
DIRIGE	<u>IMPiegato</u>	NO
	<u>DIPARTIMENTO</u>	SI'
PARTECIPA	<u>IMPiegato</u>	SI'
	<u>PROGETTO</u>	NO
RISIEDE	<u>PROGETTO</u>	SI'
	<u>CITTÀ</u>	NO
LOCALIZZ.	<u>CITTÀ</u>	NO
	<u>DIPARTIMENTO</u>	SI'

Esecuzione / laboratorio



PROCESSI

Rappresentare come l'applicazione accede, modifica, usa le informazioni.

In UML si usa il **diagramma delle attività**.

Tale diagramma descrive le attività che il sistema deve compiere:

- attività atomiche realizzate nel sistema (es funzione)
- flusso di lavoro (workflow) in cui sono coinvolte (ordine operazioni)

• ATTIVITÀ



Le attività possono essere atomiche o complesse

- Specifiche

Per le attività atomiche siamo interessati all'input e all'output e non al procedimento in mezzo (pre e post condizioni)

Nelle attività complesse invece interessa anche il processo.

- ATOMICHE (TASK)

nome - attività $(x_1:T_1 \dots x_n:T_n) : (y_1:T_1 \dots y_n:T_n)$

- PRECONDIZIONI:

- POSTCONDIZIONI: → EFFETTO ATTIVITÀ

$x_i:T_i$ sono i parametri in ingresso

$y_i:T_i$ sono i risultati (in UML posso avere più di uno)

In JAVA sono metodi statici e quindi non si riferiscono a nessun oggetto

- I/O

nome - I/O () : $(x_1:T_1 \dots x_n:T_n)$

Sono particolari attività atomiche che hanno il compito di prendere dati necessari alle successive attività o restituire i risultati richiesti dal programma.

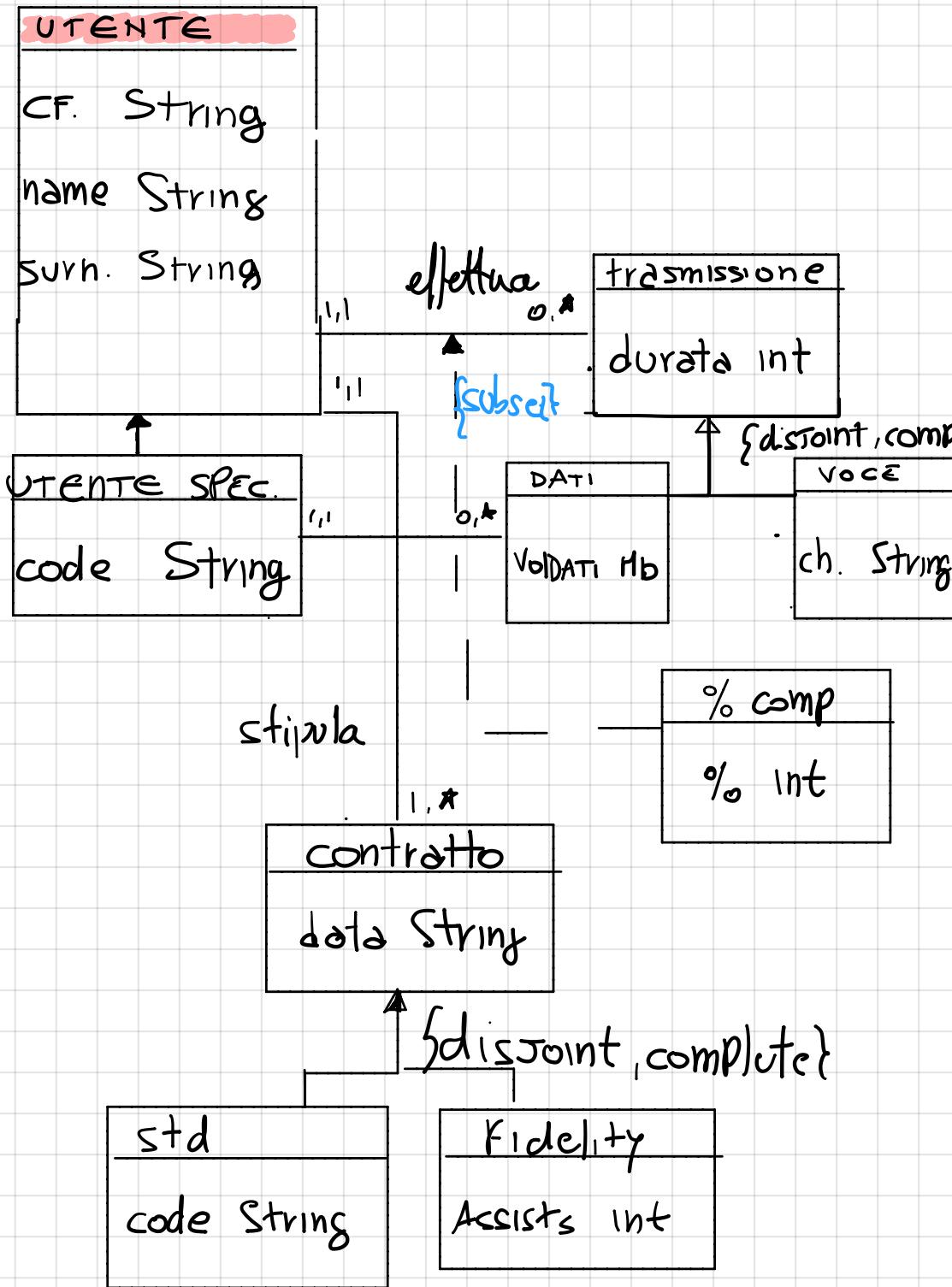
- COMPLESSE

In questo caso è di interesse il **flusso di controllo**.

La segnatura è uguale a quella delle attività atomiche + variabili di processo

Le attività complesse hanno sempre accesso al diagramma delle classi sia in lettura che in scrittura.

Le **variabili di processo** sono delle variabili private che funzionano in memoria locale per passare valori tra sottoattività.



Operazione 1 (contratto: c) (% comp: p)

Precondizione: nessuna

postcondizione: #trasm. = card ($\{(u, t) \mid u \in \text{utente} \wedge t \in \text{contratto}\}$)

$$\#trasm_{doti} = \text{card} (\{u, t \mid u \in \text{utente} \wedge t \in \text{contratto}\})$$

$$p = \frac{\# \text{doti}}{\# \text{trasm.}} \cdot 100$$

Operazione 2 ($I = \text{insieme(utenti)}$) (res: % comp)

precondizione: $\forall u \in I, u \in UT.\text{Spec}$.
postcondizione:

InsiemeComp è l'insieme di tutte le % comp di tutti gli elementi di I

$$res = \text{Inox}(\text{InsiemeComp})$$

Responsabilità:

effettua
utente $SI^{2,2}$
trasm. SI^1

stipula
utente SI^1
contratto $SI^{1,2}$

effettua Spec
utente Spec. $SI^{2,2}$
trasm. dati SI^1

PATTERNS

• MVC Model View Controller

Isolare il gestore dei dati e il "visualizzatore" degli stessi per poter riutilizzare lo stesso sorgente per applicazioni diverse (app, webapp ecc) Lo view è la "VISTA" ovvero la visualizzazione dei dati (model)



Il controller conosce il model, ma non può interagire direttamente con il model, ma attiva delle "attività" che agiscono sul model.

N.B. Come ANDROID!

Lo scopo di questo pattern è di minimizzare le dipendenze: se cambio il model, mi basta cambiare il controller (de serie).

Rendere il codice modulare.

In questo modo è possibile sviluppare view e model in parallelo.

DIAGRAMMA STATI / TRANSIZIONI

Viene definito per UNA CLASSE e descrive l'evoluzione di un oggetto di quella classe.

STATO: situazione in cui un oggetto ha delle proprietà stabili

● iniziale ○ finale

Un oggetto può non avere stato finale

TRANSIZIONE: è il passaggio da uno stato all'altro

A — EVENTO (CONDIZIONE) / AZIONE — B

(La condizione è chiamata guardia (guard).

L'evento è quasi sempre presente (condizione e azione sono optional).

Gli stati possono essere composti (MACRO-STATI) ovvero avere un nome e contenere a loro volta un diagramma. I sottostati ereditano le transizioni in uscita dal macro-stato.

ASSOCIAZIONI RESPONSABILITÀ SINGOLA (es. in A)



(0,1): campo privato di tipo B e relativo set/remove/get in A

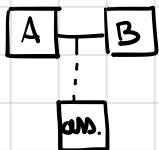
(0,*): campo privato di tipo HashSet e relativo set /remove /get

- lo suo get restituisce una copia (set) nel caso senza condivisione di memoria
- lo suo get restituisce un iterator nel caso con condivisione di memoria

(0,1) con attributi: l'associazione diventa una classe con le due classi che formano l'associazione e gli attributi come campo privato dei rispettivi tipi, i getter, equals (ndo A = B no attributi) Nella classe A verrà creato un campo privato di tipo della classe dell'associazione Il setter in A verifica che il l'attributo associazione è null e che A sia tra gli attributi dell'cls. stessa. Per cambiare associazione in A è necessario eliminare quello precedente.

(0,*) con attributi: In questo caso l'attributo in A sarà una collection (hash) di tipo link e lo remove link verifica che lo A nel link sia effettivamente lo A che invoca la funzione removeLink. Il resto riguarda a sopra.

ASSOCIAZIONI RESPONSABILITÀ DOPPIA



Verrà creata una classe ManagerAssociazione che inserisce ed elimina le associazioni. Ogni suo oggetto ha un riferimento ad un oggetto Java che rappresenta un link di tipo associazione.

(0,1)

Il suo costruttore sarà privato e la classe final. Le due classi invoceranno il metodo setter con argomento il link, il metodo invocerà a sua volta il metodo statico setter del manager che:

- 1) controllo che il link ≠ null

2) i due riferimenti al link nelle domi che ne fanno parte sono null

være poi creato il manager con il costruttore privato e passato come argomento ad un altro setter che prende come argomento il manager: se ! null il riferimento al link in A e B verrà posto uguale a manager.riferimentoLink.

(0,*)

Come sopra con alcune differenze. La classe che partecipa all'associazione con molteplicità (0,*) avrà un campo di tipo set che contiene tutti i link e lo relativo get restituisce una copia del set e avranno le funzioni che prendono come argomento il manager lavoreranno su una collection e non più un campo singolo. (es. link = manager.getLink → insieme.add(manager.getLink))

(x,*)

Le classi con cardinalità minima diversa da zero avranno un'ulteriore funzione che ritornerà il numero di link per verificare se link.size() > x e lo get link lancierà un'eccezione se tale numero sarà < x.

(0,x)

Sono combinatori di sopra, ma con i segni invertiti

ASSOCIAZIONI n-arie

Si assume che la molteplicità è sempre $(0, *)$, ovvero se senza attributi viene creata la classe link, con responsabilità singolo la classe con responsabilità over il set che contiene i link. Con responsabilità multipla si crea la classe manager come visto prima.

ASSOCIAZIONI ORDINATE

Analoghe a quelle NON ordinate con la differenza che si utilizzano list e linked list al posto di Set e HashSet e bisogna quindi stare attenti alle ripetizioni. NB nel caso di responsabilità doppia list e linked list sono usate solo nelle classi in cui si specifica ordered, nelle altre si continuano ad usare set e hashset.

GENERALIZZAZIONI

La sottoclasse estende la classe padre e il suo costruttore richiama quello del padre e setta gli attributi "in più". Nel caso di generalizzazione completa la classe padre sarà abstract.

