

# Costruzione di modelli con OCL

Lezione 23 di Ingegneria del software

Docente: Giuseppe Scollo

Università di Catania, sede di Comiso (RG)

Facoltà di Scienze Matematiche, Fisiche e Naturali  
Corso di Studi in Informatica applicata, AA 2007-8

## Indice

1. Costruzione di modelli con OCL
2. aggiungere informazione con OCL
3. aggiungere invarianti
4. operare su collezioni di oggetti
5. Set, Bag, OrderedSet, Sequence
6. aggiungere pre- e post-condizioni
7. ereditarietà, commenti, variabili locali
8. estensioni di diagrammi di classi
9. pre- e post-condizioni su operazioni
10. altre estensioni dei diagrammi di classi
11. estensioni di diagrammi di componenti
12. estensioni di diagrammi di interazioni
13. OCL su diagrammi di macchine a stati
14. OCL su diagrammi di attività e di casi d'uso
15. alcune indicazioni di stile di modellazione

## aggiungere informazione con OCL

con riferimento al modello di rete di servizi presentato nella lezione 17:  
**inizializzazione di attributi (o estremi di associazioni)**

---

```
context SchedaIscrizione::valida
init: true
context CorsoDiStudio::iscritti
init: Set{}
```

---

specifica di **derivazione** del valore di elementi del modello

---

```
context SchedaIscrizione::nomeTitolare
derive: titolare.titoloStudio.concat(' ').concat(titolare.nome)
```

---

specifica di **operazioni query**

---

```
context CorsoDiStudio::getServiziF():Set(ServizioFormativo)
body: fornitori.serviziResi->asSet()
```

---

specifica di **nuovi attributi e operazioni** (v. il diagramma precedente per la classe **ModuloIns**)

---

```
context ServizioFormativo
def: getIns():Set(Insegnamento) = getM().insegnamenti->asSet()
```

---

## aggiungere invarianti

è buona regola di stile dare un **nome** a ciascun invariante  
con riferimento al modello di rete di servizi presentato nella lezione 17:  
**invarianti su attributi**

---

```
context Studente
inv lunghezzaCF: codiceFiscale.length = 16
```

---

quando il **tipo** di un attributo è una **classe**

---

```
context Transazione
inv servizioCdS: corso.getServiziF()->includes(ServizioFormativo)
```

---

invarianti su **oggetti associati**

---

```
context SchedaIscrizione
inv lunghezzaCFtitolare: titolare.codiceFiscale.length = 16
```

---

invarianti e **classi di associazione**

---

```
context Iscrizione
inv schedaIschr: iscritti.schedeServizi->includes(self.scheda)
```

---

## operare su collezioni di oggetti

con riferimento al modello di rete di servizi presentato nella lezione 17:

alcune **operazioni** sulle collezioni: **size**, **select**

---

```
context CorsoDiStudio
inv minServizi: fornitori.serviziResi->size() >= 1
```

---

```
context Studente
inv schedeCdS: corsi->size() = schedeServizi->select(valida)->size()
```

---

altre operazioni sulle collezioni: **reject**, **forAll**, **isEmpty**

---

```
context ModoServizio
inv lsIscrizioni: registrazioni->forAll(scheda.livello = ls)
```

---

```
context ModoServizio
inv lsIscrizioni: registrazioni->reject(scheda.livello = ls)->isEmpty()
```

---

l'operazione **collect**, di frequente uso **implicito** nella "dot notation" su collezioni:

```
context CorsoDiStudio
fornitori.nrStudenti abbrevia fornitori->collect(nrStudenti)
```

## Set, Bag, OrderedSet, Sequence

altre operazioni sulle collezioni:

```
notEmpty, includes(object), union(collection), intersection(collection)
```

**Bag** = **Set** + molteplicità

**OrderedSet** = **Set** + ordine lineare

**Sequence** = **Bag** + ordine lineare

collect su **Set** → **Bag**

collect su **OrderedSet** → **Sequence**

**N.B.** collect implicita nella **navigazione**

con riferimento al modello di rete di servizi presentato nella lezione 17:

---

```
context PartnerFormazione
inv: nrStudenti <= offertaFormativa.iscritti->asSet()->size()
```

---

tenendo conto dell'ordinamento dei sottotipi di **LivelloServizio** (v. appresso per **oclIsTypeOf**):

---

```
context CorsoDiStudio
inv: livelli.first().oclIsTypeOf(Auto)
```

---

## aggiungere pre- e post-condizioni

pre- e post-condizioni: utili per specificare il **Design by Contract** (Meyer)

contesto: un'operazione

suffisso @pre :

ammesso solo nelle **post-condizioni**, fa riferimento al valore di un attributo o operazione query nello **stato precedente l'applicazione** dell'operazione

ad es., con riferimento al modello di rete di servizi presentato nella lezione 17:

---

```
context ContoIscrizione::acquisisciCF(u:Integer)
pre: dfu >= u
post: (dfu = dfu@pre - u) and (cfu = cfu@pre + u)
```

---

assenza di pre- o post-condizione: meglio se esplicita con un **commento**, sintassi: -- commento

---

```
context ContoIscrizione::acquisisciDF(u:Integer)
pre: -- nulla
post: dfu = dfu@pre + u
```

---

specifica di invio di **messaggi** nelle postcondizioni: operatore ^ (leggi: "hasSent")

---

```
context ContoIscrizione::acquisisciCF(u:Integer)
post: self^consumoDF(u)
```

---

## ereditarietà, commenti, variabili locali

la meta-operazione `oclIsTypeOf` :

permette di **discriminare fra sottotipi** in una gerarchia di ereditarietà

ad es., con riferimento al modello di rete di servizi presentato nella lezione 17:

---

```
context ContoIscrizione
inv: dfu = transazioni->select(oclIsTypeOf(AcquisizioneDFU)).uf->sum()
- transazioni->select(oclIsTypeOf(ConsumoDFU)).uf->sum()
```

---

per i commenti su più righe, sintassi:

```
/* commento
   su più righe */
```

**variabili locali**, sintassi: `let <var>[:<type>] = <expr> in <expr>`

possono migliorare la comprensibilità di espressioni complesse, ad es:

---

```
context ContoIscrizione
inv: let dfuAcquisiti = transazioni->select(oclIsTypeOf(AcquisizioneDFU)).uf->sum()
     in let dfuConsumati = transazioni->select(oclIsTypeOf(ConsumoDFU)).uf->sum()
        in dfu = dfuAcquisiti - dfuConsumati
```

---

vediamo ora, in modo più sistematico, esempi di **estensioni OCL** a vari tipi di diagrammi UML

...

## estensioni di diagrammi di classi

con riferimento al modello di rete di servizi presentato nella lezione 17:  
specifica di regole di derivazione, anche per nuovi attributi o operazioni

---

```
context ContoIscrizione::serviziUsati:Set(ServizioFormativo)
derive: transazioni.ServizioFormativo->asSet()
```

---

inizializzazione di attributi (o estremi di associazioni)

---

```
context ContoIscrizione::dfu:Integer
init: 0
context ContoIscrizione::transazioni:Set(Transazione)
init: Set{}
```

---

specifica di operazioni query

---

```
context ContoIscrizione::contoVuoto():Boolean
body: dfu = 0
```

---

specifica di invarianti, meglio se con nome

---

```
context ContoIscrizione
inv unicoLivelloServizio: transazioni.ServizioFormativo.ModoServizio.ls->asSet()->size() = 1
```

---

## pre- e post-condizioni su operazioni

principio del **Design by Contract** (Meyer):

l'“implicazione” **pre**  $\Rightarrow$  **post** specifica **diritti e doveri** di fornitore e utenti di un'operazione:

**utente:** deve soddisfare **pre**, nel qual caso ha **diritto** alla validità di **post**

**fornitore:** ha **diritto** alla validità di **pre**, nel qual caso **deve** soddisfare **post**

contesto: un'operazione

suffisso @pre : ammesso solo nelle **post-condizioni**, v. sopra

ad es., possiamo estendere il modello di rete di servizi presentato nella lezione 17 con le operazioni:

---

```
context CorsoDiStudio::iscrivi(s:Studente)
pre: s.nome <> ''
post: iscritti = iscritti@pre->including(s)
```

---

specifica di invio di **messaggi** nelle postcondizioni: operatore ^ (leggi: “hasSent” )

---

```
context ServizioFormativo::registra(i:Iscrizione)
pre: let c = i.CorsoDiStudio in let s = i.Studente in
  (fornitore.offertaFormativa->includes(c)) and (c.iscritti->includes(s))
  and (i.conto.dfu >= dfuRichiesti) and (ModoServizio.ls = i.scheda.livello)
post: (ModoServizio.registrazioni = ModoServizio.registrazioni@pre->including(i))
  and i.conto^contoVuoto()
```

---

## altre estensioni dei diagrammi di classi

risoluzione di **ambiguità** causate da cicli  $\Leftarrow$  **invariante** su una classe nel ciclo  
ad es., con riferimento al modello di rete di servizi presentato nella lezione 17:

---

```
context Transazione
inv contoTrans: ServizioFormativo.ModoServizio.registrazioni.conto->includes(conto)
```

---

definizione di **classi derivate**:

concetto analogo a quello di **vista** nelle basi di dati relazionali

si specifica una classe derivata mediante **derive** di ciascuna delle sue caratteristiche, ed eventualmente aggiungendo invarianti

**esercizio:** con riferimento al modello di rete di servizi presentato nella lezione 17, definire una classe derivata **VerbaleEsame** con classe associata **AcquisizioneCFU**

precisazione di **molteplicità dinamica** o **opzionale**  $\Leftarrow$  **invarianti** su classi associate

---

```
context Studente
inv nrIscrizioniValide: corsi->size() = schedeServizi->select(valida)->size()
```

---

risoluzione di **ambiguità** del **vincolo or** su una coppia di associazioni bidirezionali opzionali su una stessa coppia di classi  $\Leftarrow$  **invariante** su **una** delle due classi

## estensioni di diagrammi di componenti

espressioni **OCL** utili quando il diagramma di componenti contiene **interfacce** o **classi**

in tal caso, si applica quanto già visto:

**interfacce:** specifica delle operazioni mediante **pre-** e **post-condizioni**

**classi:** derivazione di caratteristiche e/o specifica di **invarianti**

## estensioni di diagrammi di interazioni

usi principali di OCL per diagrammi di interazioni:

espressione di **condizioni** su interazioni, o su frame condizionali (alt, opt)

espressione di **valori di parametri o di ritorno in messaggi**

**attenzione** : elementi dei diagrammi di interazioni sono **istanze** (non classi), ciò implica:

determinazione (ovvia?) dell'**istanza contestuale** e del **tipo contestuale** di un'espressione OCL per un diagramma di interazioni

un po' meno ovviamente... tenere in conto che:

per l'espressione del **valore di un parametro** di un messaggio (invocazione di operazione), il suo tipo è specificato nella classe del **destinatario** del messaggio

per l'espressione di un **valore di ritorno**, il suo tipo è specificato nella classe del **mittente** ...

... perché ?

uso di **nomi di istanze** in espressioni OCL

## OCL in diagrammi di macchine a stati

usi principali di OCL per diagrammi di macchine a stati:

**condizioni [guardia]** su transizioni

**condizioni in eventi when (condizione)**

**valori di parametri in azioni** (invocazione di operazione, invio di segnale)

**riferimenti a oggetti** nelle azioni suddette

**vincoli dipendenti dallo stato** dell'istanza contestuale

quando il diagramma rappresenta la dinamica di un oggetto, questo è l'**istanza contestuale** delle espressioni OCL e il suo tipo ne è il **tipo contestuale**

nel caso di espressioni OCL di **valori di parametri in azioni**, il **contesto** dell'espressione è lo **stato** o la **transizione** in cui l'azione viene eseguita

l'operazione booleana `oclInState(stato)` può applicarsi all'istanza contestuale `self` per specificare vincoli dipendenti dal suo stato, ad esempio:

---

```
context SchedaIscrizione
inv: self.oclInState(attiva) implies valida
inv: self.oclInState(scaduta) implies not(valida)
```

---

## OCL in diagrammi di attività e casi d'uso

usi principali di OCL per diagrammi di attività:

analoghi a quelli dei diagrammi di interazioni, espressione di **condizioni, valori di parametri, riferimenti a oggetti**

i **riferimenti** alle istanze che eseguono le azioni dell'attività, di solito assenti nel diagramma, possono essere **aggiunti** nella sua estensione con OCL

ciò però può risultare difficile quando il diagramma rappresenta il workflow di un **intero sistema (??)**, perché questo non è di solito visto come un unico oggetto, ragion per cui il riferimento self all'istanza contestuale non è disponibile, dunque manca il **tipo contestuale**

quest'ultimo problema si presenta sistematicamente nel (molto discutibile) tentativo di estendere i **diagrammi di casi d'uso** con espressioni OCL

(opinione personale: meglio non provarci ;)

OCL è un linguaggio di espressioni per **oggetti**

la specifica dei requisiti mediante casi d'uso **non** è orientata agli oggetti

sembra più proficuo formalizzarla mediante **altri tipi di diagrammi**: di classi, e di interazioni o di attività o di macchine a stati

ed estendere questi con OCL

## alcune indicazioni di stile di modellazione

**definizioni** di attributi o operazioni query

espressione di **vincoli di inclusione** fra associazioni

**invarianti** in luogo di ereditarietà

adoperare **navigazioni brevi**

fare attenzione alla **scelta del contesto**

evitare allInstances quando è possibile

**separare i vincoli** in congiunzioni di vincoli

usare l'abbreviazione "." per collect

dare nome agli **estremi di associazioni**