

PROGETTO DI SISTEMI

- Progettare un “sistema software” è cosa ben diversa dal progettare un algoritmo
CRISI DEL SOFTWARE: i costi di gestione diventano preponderanti su quelli di produzione
- Cosa occorre per ottenere un sistema che non solo funzioni, ma che sia anche “ben fatto” o ingegnerizzato?

Oggetti 1

Software Ingegnerizzato ... ?

- *Ben organizzato*
- *Modulare*
- *Protetto*
- *Riusabile*
- *Riconfigurabile*
- *Flessibile*
- *Documentato*
- *Incrementalmente estendibile*
- *A componenti*

Oggetti 2

COSTRUZIONE DEL SOFTWARE

- **Ingredienti computazionali**
 - le mosse di un linguaggio
- **Requisiti**
 - *non basta un sistema che “funzioni”*
- **Principi**
 - regole per una buona organizzazione
- **Modelli, Concetti, Paradigmi**
 - fonti di ispirazione

Oggetti 3

LO SVILUPPO STORICO

- 1950-1970:
 - cosa significa computing?
 - un linguaggio di programmazione, quali azioni primitive deve fornire?
- 1970-1980:
 - principi di organizzazione del software?
 - programmazione **strutturata** per piccoli componenti isolati

Oggetti 4

LO SVILUPPO STORICO

- 1980-1990:
 - importanza del modello a oggetti **OO**
 - diverse alternative alla classificazione
- 1990-2000:
 - la piattaforma di calcolo diventa *un sistema globale interconnesso* (Internet e Web)
 - come usare e mantenere vecchie applicazioni sulle nuove piattaforme?
- ...

Oggetti 5

PROGETTO & LINGUAGGI

I linguaggi di programmazione devono fornire

- **non solo un modo per esprimere**
computazioni
- **ma anche un modo per** *dare struttura alla descrizione*
- **e un supporto per organizzare il processo produttivo del software**

Oggetti 6

PROGETTO & STRUMENTI

PROBLEMA:

- le **strutture dati** e le **strutture di controllo** (programmazione strutturata)
- le **funzioni** e le **procedure**
- **file** e **moduli**

non bastano per ottenere software
***modulare* e sviluppabile in modo**
dinamico e incrementale
Perché?

Oggetti 7

LA “CRISI DIMENSIONALE”

Cambia la dimensione del problema

- **non solo le “dimensioni fisiche”**
- **ma anche le astrazioni, i modelli, gli strumenti** più opportuni per progettare

L'attenzione si sposta
dal **singolo algoritmo** (e da una o più funzioni)
al **coordinamento di componenti**
e agli **strumenti** che consentono il
lavoro di gruppo, **cooperativo** e con **riuso**

Oggetti 8

LA “CRISI del SOFTWARE”

Programmi di piccole dimensioni (in the small)

- enfasi sull'*algoritmo*
- programmazione strutturata

Programmi di medie dimensioni

- funzioni e procedure come astrazioni di espressioni/istruzioni complesse
- decomposizione degli algoritmi in blocchi funzionali

Oggetti 9

LA “CRISI del SOFTWARE”

Programmi di grandi dimensioni (in the large)

- trattano grandi quantità di informazioni
 - la decomposizione funzionale è inadeguata
 - dati e funzioni che elaborano i dati sono *scorrelati*: nulla indica che le une agiscano sugli altri
- devono essere sviluppati da gruppi, ma
 - la decomposizione funzionale e il disaccoppiamento dati/funzioni non agevolano la decomposizione del lavoro

(segue)

Oggetti 10

LA “CRISI del SOFTWARE”

Obiettivo dei programmi e componenti

- trattano dati relativi a *entità del mondo reale* (persone, oggetti, grafici, documenti)
- *interagiscono* con entità del mondo reale
- Tuttavia:
 - le entità del mondo reale non sono "dati" su cui operano delle funzioni
 - sono entità che devono essere trattate in modo *coerente alla loro essenza*

Oggetti 11

LA “CRISI GESTIONALE”

Il costo maggiore nel processo di produzione del software è dovuto alla manutenzione

- correttiva (per eliminare errori)
- adattativa (per rispondere a nuove esigenze)

Programmi di piccole dimensioni

- trovare gli errori non è difficilissimo
- l'impatto delle modifiche è intrinsecamente limitato dalle piccole dimensioni del programma

Oggetti 12

LA “CRISI GESTIONALE”

Programmi di medie dimensioni

- individuare gli errori è già più complesso
- l'impatto delle modifiche si propaga, a causa del non-accoppiamento dati/funzioni, anche a funzioni o procedure *diverse* da quella modificata

Programmi di grandi dimensioni

- trovare gli errori può essere *estremamente difficile e oneroso*
- data la propagazione delle modifiche, ogni cambiamento coinvolge *tutto il team di sviluppo*

Oggetti 13

LA “CRISI GESTIONALE”

Programmi di medie dimensioni

- individuare
- l'impatto
- non-accoppiamento
- funzioni o

***È necessario cambiare
RADICALMENTE il modo
di concepire, progettare e
programmare il software***

sa del
fun-
cata.

Programmi di grandi dimensioni

- trovare gli errori può essere *estremamente difficile e oneroso*
- data la propagazione delle modifiche, ogni cambiamento coinvolge *tutto il team di sviluppo*

Oggetti 14

OBIETTIVO

Costruzione di software

*ben organizzato, modulare, protetto,
riusabile, riconfigurabile (dinamicamente?),
flessibile, documentato, incrementalmente
estendibile, ...*

*L'enfasi non è più tutta / solo / prioritariamente
su efficienza e su ottimizzazione*

Oggetti 15

QUALE PROGETTO?

Spesso

- si studia un linguaggio
- si progetta in termini del linguaggio, usando
 - i costrutti del linguaggio
 - metodologie legate al linguaggio (*idiomi, framework*)
 - modi standard (*schemi indotti dal linguaggio*)

Oggetti 16

COSA SI DOVREBBE FARE?

1. Definire i **requisiti**
2. Fare l'**analisi** e il **progetto** adottando un **modello adeguato**
3. **Implementare**
(anche in un linguaggio di più basso livello rispetto al modello adottato)
4. Effettuare il **testing**

Processo di sviluppo "iterativo (a spirale)"

Oggetti 17

PRODUZIONE DEL SOFTWARE: COSA OCCORRE?

- **Concetti e Metodologie**
per pensare
- **Metodologie, Processi e Strumenti**
per produrre

Oggetti 18

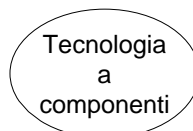
PRINCIPI STRUTTURALI

- Information hiding, incapsulamento
- Località
- Parametricità

- **Astrazione vs. rappresentazione**

- **"Cosa fa" vs. "come lo fa"**

- a livello di "dati"
- a livello di "elaborazione"



Oggetti 19

INTERFACCIA e IMPLEMENTAZIONE

- **L'interfaccia** esprime una *vista astratta* di un ente computazionale, nascondendo
 - organizzazione interna
 - dettagli di funzionamento
- **L'implementazione** esprime
 - la *rappresentazione dello stato interno*
 - il codice**di un ente computazionale**

Oggetti 20

ASTRAZIONE

- Si focalizza sul *funzionamento osservabile dall'esterno* di un componente
 - la *struttura interna* di un servitore è inessenziale agli occhi del cliente
 - basta assicurare il *rispetto del contratto* stabilito dall'*interfaccia*

Oggetti 21

INCAPSULAMENTO

- Si focalizza sull'*implementazione* di un componente
- Incapsulamento significa che la *rappresentazione concreta* di un dato può essere modificata
- *senza che vi siano ripercussioni* sul resto del programma

Astrazione e incapsulamento

Oggetti 22

DAL DIRE AL FARE ...

- In assenza di precisi supporti linguistici, in fase di codifica si può però *compromettere* il livello di astrazione
e con esso la modularità e la riusabilità della soluzione

Oggetti 23

RAPPRESENTAZIONE & ASTRAZIONE

- Attraverso i *costruttori di tipo* (array, struct, enum, etc.) il progettista può definire strutture dati che siano la *rappresentazione concreta* delle astrazioni che ha in mente
- Occorre catturare la *semantica* delle astrazioni di dato, cercando di *impedire l'accesso diretto* alla rappresentazione concreta del dato

Oggetti 24

RAPPRESENTAZIONE & ASTRAZIONE

Due grandi approcci:

- definire *singole Astrazioni di Dato*
- definire Tipi di Dati Astratti (ADT)

Oggetti 25

ASTRAZIONE di DATO

- Astrazione per focalizzare le proprietà di una entità
- Nascondere la rappresentazione concreta (incapsulamento)
- Visibilità solo di alcune operazioni sulla unica entità protetta

Oggetti 26

TIPI di DATO ASTRATTO

- Gli ADT focalizzano l'attenzione sulle proprietà logico-concettuali di un tipo
- *Lasciano sullo sfondo la rappresentazione concreta*
 - Quali proprietà rilevanti?
 - Quali operazioni visibili ai clienti (interfaccia)?
 - Quali criteri per definire l'interfaccia?
 - Quali metodologie per una soluzione flessibile?

Oggetti 27

MODULI come COSTRUTTI

- Il modulo permette di raggruppare *dati, funzioni e procedure* in una singola unità sintattica, ottenendone l'incapsulamento
- L'uso di un modulo rende possibile costruire una *barriera di astrazione* intorno alla rappresentazione concreta di una struttura dati

Oggetti 28

MODULI

- Se ben definito, un modulo permette di:
 - garantire l'uso consistente e corretto di una nuova astrazione di dato
 - ottenere *indipendenza dalla rappresentazione concreta*
- è facile rappresentare *una singola risorsa* (astrazione di dato):
non è infatti possibile includere lo stesso modulo più volte in un'applicazione

Oggetti 29

ESEMPIO: IL CONTATORE

Scopo: definire il concetto di *contatore*

Specifica:

- un componente caratterizzato in ogni istante da un valore (intero)
- a cui si accede tramite le operazioni di:
 - reset: azzera il valore del contatore
 - inc: incrementa di uno il valore
 - getValue: restituisce il valore attuale

Oggetti 30

ESEMPIO: IL CONTATORE

Scopo: definire

Specifica:

- un componente caratterizzato in ogni istante da un valore (intero)
- a cui si accede tramite le operazioni di:
 - reset: azzera il valore del contatore
 - inc: incrementa di uno il valore
 - getValue: restituisce il valore attuale

Non importa come è realizzato
dentro: importa il suo
comportamento osservabile

Oggetti 31

ESEMPIO: IL CONTATORE

Due possibilità:

- definire in un *modulo* la *singola astrazione di dato* “contatore” e poi usarlo “così com'è”
- definire il tipo di dato astratto (ADT) “contatore” e poi sfruttarlo per *creare* “oggetti” di tipo “contatore”

Oggetti 32

ESEMPIO: IL CONTATORE

Due possibilità:

- definire in un *modulo* la *singola astrazione di dato* “contatore” e poi usarlo “così com’è”

- definire il tipo di dato astratto (ADT) “contatore” e poi sfruttarlo per *creare* “oggetti” di tipo “contatore”
 - Pro: possibile garantire vero incapsulamento
 - Contro : si può definire un solo oggetto “contatore” (l’oggetto *coincide col modulo* che lo realizza)

Oggetti 33

ESEMPIO: IL CONTATORE

- Pro: possibile definire tanti oggetti “contatore” quanti ne servono
- Contro: mancanza di vero incapsulamento (chiunque può in realtà accedere allo stato interno)
- definire il tipo di dato astratto (ADT) “contatore” e poi sfruttarlo per *creare* “oggetti” di tipo “contatore”

Oggetti 34

IL CONTATORE COME MODULO

- Definire il contatore come *singola risorsa protetta* dentro a un modulo

```
static int cont;
```

- con operazioni che agiscono *implicitamente* su essa

```
void reset(void);
```

```
void inc(void);
```

```
int getValue(void);
```

Oggetti 35

IL CONTATORE COME MODULO

- Le operazioni *non hanno alcun parametro* perché ora agiscono tutte, implicitamente, sull’unico contatore esistente: quello rappresentato dalla variabile statica `cont`
- con operazioni che agiscono *implicitamente* su essa

```
void reset(void);
```

```
void inc(void);
```

```
int getValue(void);
```

Oggetti 36

CONTATORE COME MODULO: USO

- Si importano *solo* le dichiarazioni delle funzioni (`mcounter.h`)
- Si usa semplicemente il contatore “racchiuso” nel modulo *senza dover definire alcunché*

```
#include "mcounter.h"
main() {
    int v;
    reset(); inc(); inc();
    v=getValue();
}
```

Oggetti 37

L'APPROCCIO A MODULO

- Separa *interfaccia e implementazione*
- Rende il cliente *indipendente dalla struttura interna dell'AD* (servitore)
- Garantisce l'incapsulamento
 - i clienti vedono *solo le dichiarazioni* delle operazioni: *non conoscono la struttura interna* della risorsa (privata) del modulo
- Offre al cliente una **singola risorsa** (da usare senza doverla definire): *non è adatto se servono più risorse*

Oggetti 38

IL CONTATORE COME ADT (?)

- Definire il tipo di dato astratto (ADT) “contatore”...

```
typedef ..... contatore;
```

- ...con le sue operazioni:

```
void reset(contatore*);
void inc(contatore*);
int getValue(contatore
```

tutti i clienti vedono la typedef
→ tutti sanno in realtà *come è fatto*

IL CONTATORE COME ADT

- Definire il tipo di dato astratto (ADT) “contatore”
 - reset e inc devono *cambiare lo stato* del contatore → necessario passare *un puntatore*
- ...con le sue operazioni:

```
void reset(contatore*);
void inc(contatore*);
int getValue(contatore);
```

getValue invece deve *solo accedere allo stato* del contatore, ma senza cambiarlo
→ è più opportuno il passaggio *per valore*

IL CONTATORE COME ADT

- **NOTA:** la specifica del tipo di dato astratto “contatore” *non ha richiesto per ora alcuna scelta* circa la sua organizzazione interna
- La struttura interna del contatore è irrilevante per i clienti che devono usarlo!
- Ma è astrazione e incapsulamento?

Oggetti 41

IL CONTATORE COME ADT: USO

- Si importano la definizione dell’ADT e le dichiarazioni delle funzioni (`counter.h`)
- Si definiscono tante variabili contatore quante ne occorrono, e si usano

```
#include "counter.h"
main() {
    int v1, v2;
    contatore c1, c2;
    reset(&c1); reset(&c2);
    inc(&c1); inc(&c1); inc(&c2);
    v1=getValue(c1); v2=getValue(c2);
}
```

Oggetti 42

IL CONTATORE COME ADT

Implementazione

- La struttura interna del contatore diventa rilevante quando giunge il momento di *realizzarlo*
- Se ora scegliamo di rappresentare lo stato con un intero, avremo:
`typedef int contatore;`

Oggetti 43

IL CONTATORE COME ADT

È ora possibile *definire (e non solo dichiarare) le operazioni:*

```
void reset(contatore* c){
    *c = 0;
}
void inc(contatore* c){
    (*c)++;
}
int getValue(contatore c){
    return c;
}
```

Oggetti 44

IL CONTATORE COME ADT

Una implementazione alternativa

- Se invece scegliessimo di rappresentare lo stato con una stringa di "I" (notazione a numeri romani):

typedef char contatore[21];

- Qui
 - la stringa vuota indica lo zero
 - "I" indica 1, "II" indica 2, etc.

ok solo per numeri fino a 20

- *per il cliente non cambia interfaccia*

Oggetti 45

IL CONTATORE COME ADT

Le operazioni diventerebbero:

```
void reset(contatore* c){
    (*c)[0] = '\0';
}

void inc(contatore* c){
    int x=strlen(*c);
    (*c)[x]='I'; (*c)[x+1]='\0';
}

int getValue(contatore c){
    return strlen(c);
}
```

Oggetti 46

L'APPROCCIO DEGLI ADT

- Consente di *separare interfaccia e implementazione*
- Rende il cliente *indipendente dalla struttura interna dell'ADT* (servitore)
- Permette al cliente di definire *tanti oggetti quanti gliene occorrono*
- **Ma non garantisce incapsulamento**
 - tutti i clienti vedono la `typedef`,
 - *conoscono la struttura interna dell'ADT*
 - *e possono violare il protocollo di accesso*

Oggetti 47

L'APPROCCIO DEGLI ADT

- Consente di *separare interfaccia e implementazione*
 - Rispettare il *protocollo di accesso* stabilito dall'interfaccia è, per il cliente, un atto volontario (servitore)
- **Non garantisce incapsulamento**
 - Se un cliente malevolo decide di *aggirare le operazioni previste* e di accedere direttamente allo stato interno degli oggetti, nessuno può impedirlo
 - *conoscono la struttura interna dell'ADT*
 - *e possono violare il protocollo di accesso*

Oggetti 48

L'OBIETTIVO

L'ideale sarebbe:

- garantire l'incapsulamento come è in grado di fare l'approccio "a modulo"
- ma nel contempo consentire al cliente di creare *tanti oggetti quanti gliene occorrono* come nell'approccio "ad ADT"
- garantendo inoltre
 - separazione interfaccia / implementazione
 - indipendenza dalla rappresentazione

Oggetti 49

IL CONTATORE COME ADT (?)

- Definire il tipo di dato astratto "contatore" all'interno del modulo ADT, all'esterno una chiave (ad es. intero)

```
typedef int contkey;
```

- e le sue operazioni esterne

```
contkey create(void); destroy(contkey);  
  
void reset(contkey); void inc(contkey);  
int getValue(contkey);
```

i clienti vedono una chiave e non il dato → e non sanno *come è fatto*

IL CONTATORE ADT RAPPRESENTAZIONE INTERNA

- La specifica esterna del tipo di dato astratto "contkey" *non specifica* nulla della organizzazione interna
- Sono state inserite due operazioni di creazione e di distruzione

```
contkey create(void); destroy(contkey);
```

- Astrazione e Incapsulamento?

Oggetti 51

IL CONTATORE: USO ESTERNO

- Si importano la definizione dell'ADT e le dichiarazioni delle funzioni (counter.h)
- Si definiscono tante variabili contatore quante ne occorrono, e si usano

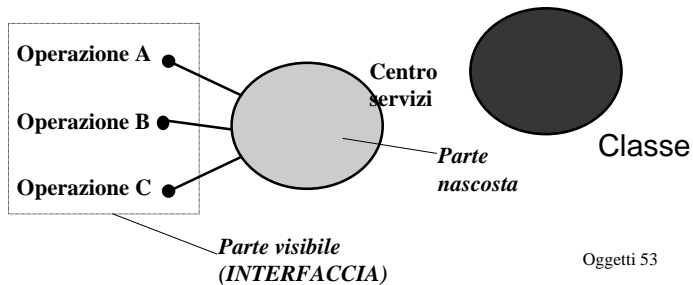
```
#include "counter.h"  
main() {  
    int v1, v2; contkey c1;  
    c1 = create();      reset(c1);  
    inc(c1); inc(c1);  
    v1=getValue(c1); v2=getValue(c1);  
    destroy (c1);  
}
```

Oggetti 52

OGGETTI come ADT

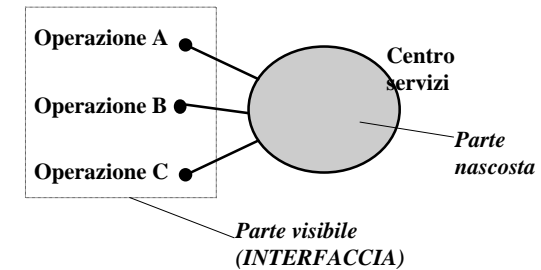
Un oggetto incarna la idea di ADT in senso completo

Viene creato da un tipo, detto classe, e è capace di esportare operazioni e incapsulare lo stato interno



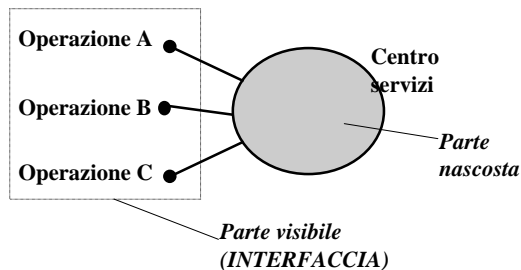
IL CONCETTO DI OGGETTO

Un oggetto viene inteso come un *centro di servizi* dotato di una *parte visibile (interfaccia)* e di una *parte nascosta*



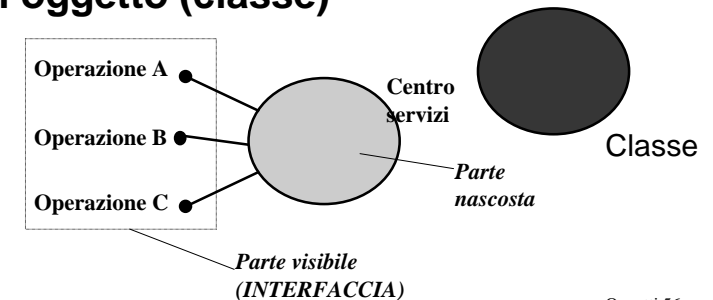
IL CONCETTO DI OGGETTO

Offre agli altri oggetti (clienti) un insieme di attività (operazioni) *senza che sia nota/accessibile la sua organizzazione interna*



IL CONCETTO DI OGGETTO

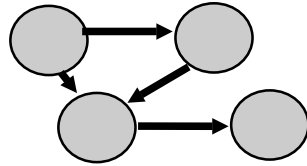
Ogni cliente può *creare (istanziare)* *tanti oggetti quanti gliene occorrono* a partire da una sorta di “modello” dell’oggetto (classe)



SISTEMI A OGGETTI

L'architettura di un sistema a oggetti:

- un insieme di oggetti che *interagiscono gli uni con gli altri*
- senza conoscere *nulla* delle rispettive rappresentazioni concrete
- Modello di interazione a “*scambio di messaggi*”



Oggetti 57

L'IDEA DI OGGETTO

- integra *dati e elaborazione (comportamento)*
- promuove approcci di progettazione e sviluppo sia top-down sia bottom-up
- cattura i principi fondamentali di una corretta strutturazione del software
- introduce una interazione molto ricca e orientata a gestire la complessità

Oggetti 58

LE PROPRIETÀ DI UN OGGETTO

- Un oggetto possiede *stato, funzionamento e identità*
- Struttura e funzionamento di *oggetti simili* sono definiti nella loro classe comune, di cui essi sono *istanze*
- I termini *istanza* e *oggetto* sono intercambiabili

Oggetti 59

IL CONCETTO DI CLASSE

- La *classe* descrive la *struttura interna* e il *funzionamento* di un oggetto
- Oggetti appartenenti a una stessa classe hanno:
 - la stessa *rappresentazione interna*
 - le stesse *operazioni*
 - lo stesso *funzionamento*

Oggetti 60

RELAZIONI TRA OGGETTI

- Le relazioni tra oggetti sono *il punto-chiave della progettazione “in-the-large”*
- Sono oggi codificate in “relazioni”
 - incapsulamento, possesso
 - ereditarietà

Oggetti 61

EREDITARIETÀ

- Una *relazione tra classi*
- Ereditarietà singola: una classe condivide la struttura e/o il funzionamento definito *in un'altra classe*
- Ereditarietà multipla: una classe condivide la struttura e/o il funzionamento definito *in varie altre classi*

Oggetti 62

EREDITARIETÀ

- Una *relazione*
- Ereditarietà singola: una classe condivide la struttura e/o il funzionamento definito *in un'altra classe*
- Ereditarietà multipla: una classe condivide la struttura e/o il funzionamento definito *in varie altre classi*

La nozione di ereditarietà scaturisce dall'esigenza di poter *condividere (parti di) una descrizione*, cioè di *riusare concetti già esistenti e codice già scritto e provato*.



LINGUAGGI AD OGGETTI

- Linguaggi object-oriented
 - supportano sia incapsulamento, sia ereditarietà
- Linguaggi object-based
 - supportano primariamente il concetto di modulo (come Ada e Modula2)

Oggetti 64

SVILUPPO “OBJECT-ORIENTED”

- Si può sviluppare software in modo “object-oriented” anche *senza disporre a livello implementativo di un linguaggio a oggetti?*
- Si... ma
 - è responsabilità del progettista *autolimitarsi* e strutturare il software in modo coerente
 - il linguaggio non vincola *obbligatoriamente* al rispetto di incapsulamento e altri principi
 - il linguaggio non supporta direttamente incapsulamento, ereditarietà, etc.

Oggetti 65

CLASSI, MODULI, ADT, TIPI

- Il concetto di *classe* non coincide con il concetto di tipo
 - La classe può essere intesa come la *specifica implementazione di un tipo*
 - incluse le modalità di *costruzione* degli oggetti
- Il costrutto `class` integra
 - aspetti tipici dei costrutti per esprimere modularità
 - aspetti tipici dei costrutti per definire ADT

Oggetti 66

QUALI OPERAZIONI ?

<i>Classificazione delle operazioni</i>	<i>Categorie</i>
dal punto di vista di chi le usa	<ul style="list-style-type: none">• costruttori/distruttori• selettori• trasformatori• predicati•
dal punto di vista di chi le realizza	<ul style="list-style-type: none">• primitive / non-primitive• operazioni di costruzione / operazioni di configurazione• operazioni private / pubbliche

Oggetti 67

IL PUNTO DI VISTA UTENTE

- Dal punto di vista dell'utente, vi sono:
 - operazioni di costruzione (*costruttori/distruttori*)
 - operazioni di selezione di componenti (*selettori*)
 - operazioni di verifica di proprietà (*predicati*)
 - operazioni di trasformazione (*trasformatori*)
- La presenza o assenza di certe categorie di operazioni implica diversi *tipi di oggetti*:
 - oggetti *atomici* vs. oggetti *composti*
gli oggetti atomici *non hanno selettori*
 - oggetti *con o senza stato*
gli oggetti senza stato *non hanno trasformatori*

Oggetti 68

IL PUNTO DI VISTA DELL'IMPLEMENTATORE

- L'implementatore, invece, opera *diverse classificazioni* in relazione all'aspetto che intende mettere in luce
 - operazioni di costruzione / configurazione
 - operazioni primitive / non primitive
 - operazioni private / pubbliche
- Questi criteri di classificazione *operano su dimensioni diverse* uno rispetto all'altro
 - possono quindi esistere primitive private o pubbliche, operatori di costruzione privati e pubblici, etc.

Oggetti 69

IL PUNTO DI VISTA DELL'IMPLEMENTATORE

- L'implementatore, invece, opera *diverse classificazioni* in relazione all'aspetto che intende mettere in luce
 - operazioni di costruzione / configurazione
 - operazioni primitive / non primitive
 - operazioni private / pubbliche
- Questi criteri di classificazione *operano su dimensioni diverse* uno rispetto all'altro
 - possono quindi esistere primitive private o pubbliche, operatori di costruzione privati e pubblici, etc.

Configurazione: un'operazione che inizializza un oggetto

Costruzione: un'operazione di *definizione*, che alloca memoria e eventualmente inizializza l'oggetto. Anche **Distruzione** esplicita o implicita.

Oggetti 70

IL PUNTO DI VISTA DELL'IMPLEMENTATORE

- L'implementatore, invece, opera *diverse classificazioni* in relazione all'aspetto che intende mettere in luce
 - operazioni di costruzione / configurazione
 - operazioni primitive / non primitive
 - operazioni private / pubbliche
- Questi criteri di classificazione *operano su dimensioni diverse* uno rispetto all'altro
 - possono quindi esistere primitive private o pubbliche, operatori di costruzione privati e pubblici, etc.

Non primitive:

- operazioni *indipendenti* dalla rappresentazione
- non accedono alla rappresentazione interna → non cambiano se essa viene modificata

Primitive:

- operazioni *dipendenti* dalla rappresentazione
- le uniche a poter accedere alla rappresentazione interna
- un insieme *funzionalmente completo*

Oggetti 71

IL PUNTO DI VISTA DELL'IMPLEMENTATORE

- L'implementatore, invece, opera *diverse classificazioni* in relazione all'aspetto che intende mettere in luce
 - operazioni di costruzione / configurazione
 - operazioni primitive / non primitive
 - operazioni private / pubbliche
- Questi criteri di classificazione *operano su dimensioni diverse* uno rispetto all'altro
 - possono quindi esistere primitive private o pubbliche, operatori di costruzione privati e pubblici, etc.

Pubbliche:

tutti possono invocarle per operare sull'oggetto

Private:

possono essere invocate solo da altre operazioni del medesimo oggetto

Oggetti 72