

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 1

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 2

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 3

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 4

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 5

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

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

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 6

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 7

ESEMPIO

- Facciamo un esempio: un contatore
- Vediamo, passo dopo passo, come si possa migliorare l'organizzazione del programma

Oggetti 8

ESEMPIO: VERSIONE 1

```
main.c
#include <stdio.h>
typedef struct {
    int valore; /* valore attuale */
    int step;
} Contatore;

void init(int vcont, int stp);
int getValue();
void incre();

Contatore C;
void main ()
{
    init(0, 2);
    printf("%d\n", getValue());
    incre();
    printf("%d\n", getValue());
}

void init (int vcont, int stp)
{
    C.valore=vcont;
    C.step=stp;
}

int getValue()
{
    return C.valore;
}

void incre ()
{
    C.valore+=C.step;
}
```

Oggetti 9

VERSIONE 1 - CONSIDERAZIONI

- E' tutto nel main
- Non abbiamo alcuna forma di separazione e di protezione

Oggetti 10

ESEMPIO – VERSIONE 2/1

```
contatore.h
typedef struct {
    int valore; /*
    valore attuale */
    int step;
} Contatore;

Contatore C;

void init (int vcont, int stp);
int getValue();
void incre ();
```

```
contatore.c
#include "contatore.h"
void init (int vcont, int stp)
{
    C.valore=vcont;
    C.step=stp;
}

int getValue()
{
    return C.valore;
}

void incre ()
{
    C.valore+=C.step;
}
```

Oggetti 11

ESEMPIO – VERSIONE 2/2

```
main.c
#include <stdio.h>
#include "contatore.h"

void main ()
{
    init(0, 2);
    printf("%d\n", getValue());
    incre();
    printf("%d\n", getValue());
}
```

Oggetti 12

VERSIONE 2 - CONSIDERAZIONI

Abbiamo fatto un primo passo:

- lavoriamo con più files...
- contatore globale definito nel .h
- funzioni separate dal main

Oggetti 13

ESEMPIO – VERSIONE 3/1

```
contatore.h
void init (int vcont, int stp);
int  getValue();
void incre ();
```

```
contatore.c
typedef struct {
    int valore; int step;
} Contatore;
Contatore C;
void init (int vcont, int stp)
{
    C.valore=vcont;
    C.step=stp;
}
int getValue()
{
    return C.valore;
}
void incre ()
{
    C.valore+=C.step;
}
}
```

Oggetti 14

ESEMPIO – VERSIONE 3/2

```
main.c

#include <stdio.h>
#include "contatore.h"

void main ()
{
    init(0, 2);
    printf("%d\n", getValue());
    incre();
    printf("%d\n", getValue());
}
}
```

Oggetti 15

VERSIONE 3 - CONSIDERAZIONI

- E' un tipo di dato astratto
- Abbiamo information hiding
- Ma... possiamo usare solo un contatore

Oggetti 16

ESEMPIO – VERSIONE 4/1

```
contatore.h
typedef struct {
    int valore;
    int step;
} Contatore;

void init (int vcont, int stp,
           Contatore *Cont);
int  getValue(Contatore Cont);
void incre (Contatore *Cont);
```

```
contatore.c
#include "contatore.h"
void init (int vcont, int stp,
           Contatore *Cont)
{
    (*Cont).valore=vcont;
    (*Cont).step=stp;
}
int getValue(Contatore Cont)
{
    return Cont.valore;
}
void incre (Contatore *Cont)
{
    (*Cont).valore+=(*Cont).step;
}
}
```

Oggetti 17

ESEMPIO – VERSIONE 4/2

```
main.c
#include <stdio.h>
#include "contatore.h"
void main ()
{
    Contatore C1, C2;
    init(0, 2, &C1);
    init(0, 1, &C2);
    printf("c1 .... %d\n", getValue(C1));
    incre(&C1);
    printf("c1 .... %d\n", getValue(C1));
    printf("c2 .... %d\n", getValue(C2));
    incre(&C2);
    printf("c2 .... %d\n", getValue(C2));
    incre(&C2);
    printf("c2 .... %d\n", getValue(C2));
    incre(&C2);
    printf("c2 .... %d\n", getValue(C2));
}
}
```

Oggetti 18

VERSIONE 4 - CONSIDERAZIONI

- E' quasi un ADT
- Le funzioni sono separate dal main
- E' possibile definire più contatori
- Ma... manca information hiding

Oggetti 19

ESEMPIO – VERSIONE 5/1

<pre>contatore.h typedef struct Contatore * ContHandler; /* contatore non è definito qui */ void init (int vcont, int stp, ContHandler *Cont); int getValue(ContHandler Cont); void incre (ContHandler Cont);</pre>	<pre>contatore.c #include "contatore.h" #include <malloc.h> #include <stdio.h> typedef struct { int valore; int step; } Contatore; void init (int vcont, int stp, ContHandler *Cont) { (* Cont) = (Contatore *) malloc (sizeof(Contatore)); ((Contatore *) (*Cont))->valore=vcont; ((Contatore *) (*Cont))->step=stp; } int getValue(ContHandler Cont) { return ((Contatore *)Cont)->valore; } void incre (ContHandler Cont) { Contatore app; app = * (Contatore *)Cont; app.valore+=app.step; * (Contatore *)Cont = app; }</pre>
--	--

Oggetti 20

ESEMPIO – VERSIONE 5/2

```
main.c

#include <stdio.h>
#include "contatore.h"

void main ()
{
ContHandler C1, C2;
init(0, 2, &C1);
init(0, 1, &C2);

printf("c1 .... %d\n", getValue(C1));
incre(C1);
printf("c1 .... %d\n", getValue(C1));

printf("c2 .... %d\n", getValue(C2));
incre(C2);
printf("c2 .... %d\n", getValue(C2));
incre(C2);
printf("c2 .... %d\n", getValue(C2));
incre(C2);
printf("c2 .... %d\n", getValue(C2));
}
```

Oggetti 21

VERSIONE 5 - CONSIDERAZIONI

- Abbiamo quello che volevamo:
 - E' un ADT
 - Rispettiamo l'information hiding
 - Separazione fra funzioni e main
- ... ma è complesso da realizzare...
- ... con il trucco usato dobbiamo gestire una forma di garbage collection...
- ... è meglio lavorare con una metodologia che implementa in modo "naturale" questi concetti.

Oggetti 22

PRINCIPI STRUTTURALI

- Information hiding, incapsulamento
 - Località
 - Parametricità
-
- **Astrazione vs. rappresentazione**
 - **"Cosa fa" vs. "come lo fa"**
 - a livello di "dati"
 - a livello di "elaborazione"

Tecnologia
a
componenti

Oggetti 23

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 24

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 25

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 26

RAPPRESENTAZIONE & ASTRAZIONE

Due grandi approcci:

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

Oggetti 27

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 28

TIPI di DATO ASTRATTO

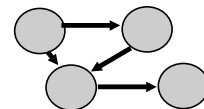
- Gli ADT focalizzano l'attenzione sulle proprietà logico-concettuali di un tipo
- **Lasciano sullo sfondo** la rappresentazione concreta
 - Quali proprietà rilevanti?
 - Quali operazioni visibili ai clienti (interfaccia)?
 - Quali criteri per definire l'interfaccia?
 - Quali metodologie per una soluzione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

Oggetti 29

SISTEMI A OGGETTI

L'architettura di un sistema a oggetti:

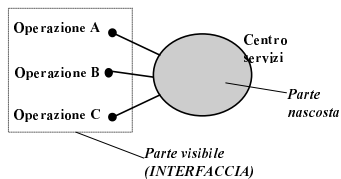
- un insieme di oggetti che **interagiscono gli uni con gli altri**
- senza conoscere **nulla** delle rispettive rappresentazioni concrete
- **Modello di interazione a "scambio di messaggi"**



Oggetti 30

IL CONCETTO DI OGGETTO

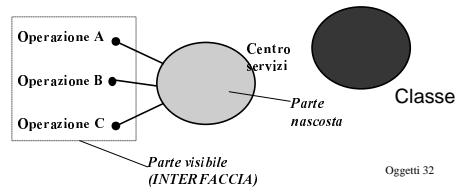
Un oggetto viene inteso come un *centro di servizi* dotato di una *parte visibile (interfaccia)* e di una *parte nascosta*



Oggetti 31

OGGETTI come ADT

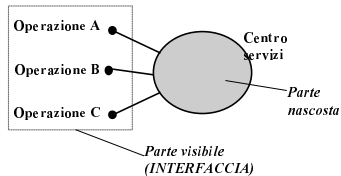
Un oggetto incarna la idea di ADT in senso completo
Viene creato da un tipo, detto classe, e è capace di esportare operazioni e incapsulare lo stato interno



Oggetti 32

IL CONCETTO DI OGGETTO

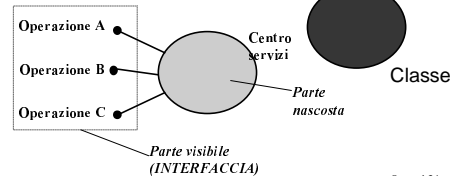
Offre agli altri oggetti (clienti) un insieme di attività (operazioni) *senza che sia nota/accessibile la sua organizzazione interna*



Oggetti 33

IL CONCETTO DI OGGETTO

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



Oggetti 34

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 35

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 36

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 37

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 38

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 39

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*.

