Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria – Reggio Emilia

CORSO DI

Reti di Calcolatori
Programmazione ad Oggetti

Ing. Franco Zambonelli

Lucidi realizzati in collaborazione con
Ing. Enrico Denti - DEIS, Univ. Bologna

Anno Accademico 2001-2001

INTRODUZIONE ALLA TECNOLOGIA OBJECT-ORIENTED

MOTIVAZIONI

- progettare sistemi software di grandi dimensioni richiede *adeguati supporti*
- "crisi del software": i costi di gestione diventano *preponderanti* su quelli di produzione
- il software dovrebbe essere **protetto, riusabile, documentato, modulare, incrementalmente estendibile**

IL PUNTO CHIAVE

- 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 bene il processo produttivo del software**

L'OBIETTIVO

- costruzione ***modulare* e *incrementale*** del software
- ottenuta per **estensione** / **specializzazione** di ***componenti***
  ➔ tecnologia ***component-based***
- Le strutture dati e le strutture di controllo (programmazione strutturata) *non bastano*
- Le funzioni e le procedure *non bastano*
- File e moduli come "contenitori di descrizioni" *non bastano*

LE NECESSITA'

- servono strumenti per la costruzione ***modulare* e *incrementale*** del software

CRISI DIMENSIONALE

Programmi di piccole dimensioni

- accento sull'algoritmo
- diagrammi di flusso
- programmazione strutturata

Non appena il programma cresce in dimensioni, non si riesce più a gestirlo: bisogna diminuire la complessità!

Programmi di Medie Dimensioni

- funzioni e procedure come astrazioni di istruzioni complesse
- decomposizione degli algoritmi in blocchi funzionali (anche su file diversi, per permettere lo sviluppo di equipe)

Alla base rimane il concetto di "algoritmo": tutto il programma è un grosso algoritmo spezzato su più funzioni/file

Programmi di Grandi Dimensioni

- tipicamente trattano grandi moli di dati MA
  - la decomposizione funzionale non è adeguata
  - non c'è accoppiamento dati-funzioni che li elaborano: dati elaborati globalmente da tutto il programma
- tipicamente devono essere sviluppati da team MA
  - la decomposizione funzionale e il disaccoppiamento dati-funzioni non permette la decomposizione del lavoro
- tipicamente trattano ed elaborano dati relativi ad entità del mondo reale (persone, oggetti, grafici, documenti), e interagiscono con entità del modo reale MA

- le entità del mondo reale non sono "dati" su cui operano delle funzioni, ma sono entità che devono essere trattate in modo coerente alla loro essenza

CRISI GESTIONALE

Il costo maggiore nel processo di produzione del software è dovuto al suo mantenimento:

- correttivo
- adattativo

Programmi di piccole dimensioni

- non difficilissimo trovare gli errori
- propagazione degli effetti delle modifiche limitate intrinsecamente dalle dimensioni del programma

Programmi di Medie Dimensioni

- gestione basata sulle procedure per l'individuazione degli errori
- gli effetti delle modifiche non sono limitati alle procedure in cui tale modifiche sono fatte, ma si propagano, a causa del non-accoppiamento dati funzioni

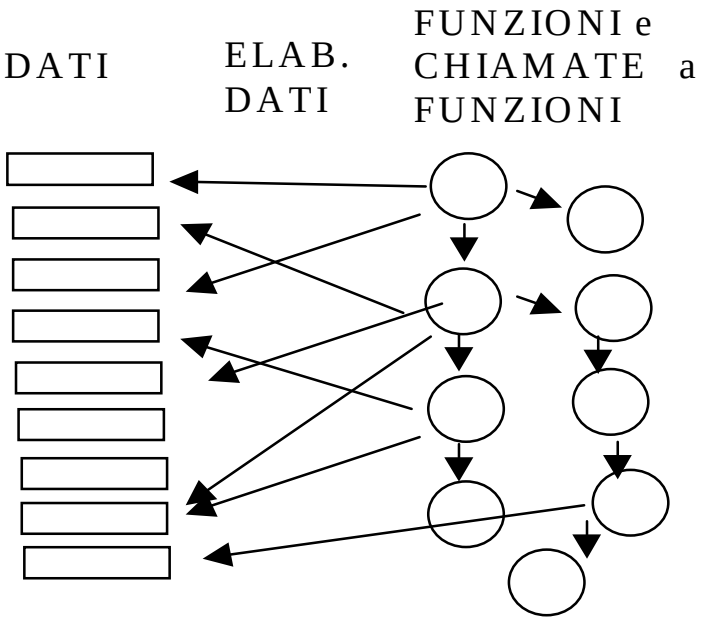
Programmi di Grandi Dimensioni

- quasi impossibile trovare gli errori
- se le modifiche si possono propagare non è possibile fare delle modifiche senza coinvolgere tutto il team di sviluppo

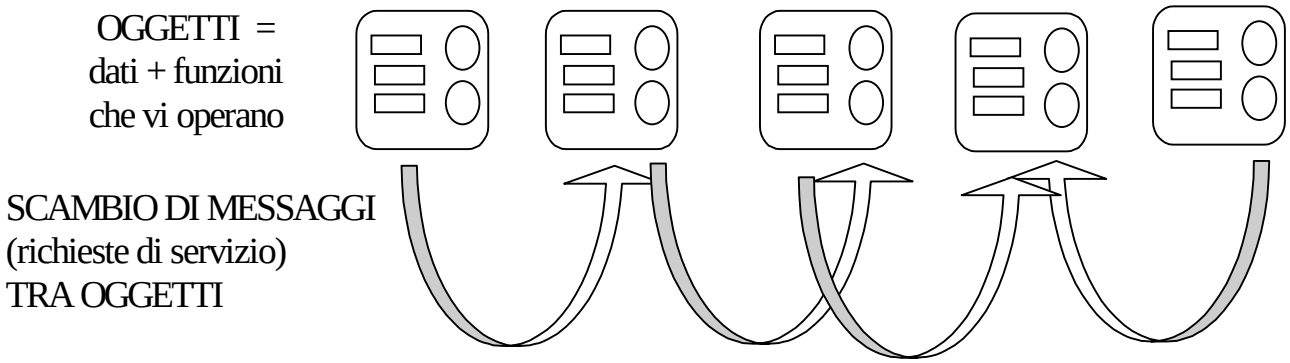
è necessario cambiare RADICALMENTE il modo di concepire, progettare e programmare il software!

IL CAMBIO DI PARADIGMA

Tradizionale:



Ad Oggetti:



CONCETTI CHIAVE

- Astrazione di Dato
- Tipi di dato astratto (ADT, *Abstract Data Type*)
- Modello Cliente / Servitore (*Client* / *Server*)
- Oggetti e Classi
- Moduli

ASTRAZIONE DI DATO

- Focalizzare l'attenzione sulle *categorie concettuali* dei dati del dominio applicativo
- costruire il programma in termini di "entità", rappresentate da una struttura dati interna NASCOSTA e da operazioni che possono AGIRE SUI DATI, che interagiscono tra loro

VANTAGGI

- agevola la modularità accoppiando dati e funzioni (che è giustificata sia per la diminuzione di complessità sia per il fatto che certi dati possono essere trattati solo tramite certe funzioni sia perché certe funzioni possono trattare solo certi dati);
- favorisce il controllo sull'integrità dei dati (solo certe funzioni possono agire sui dati e non funzioni arbitrarie che potrebbero fare dei danni o cose errate!).Se un dato risulta errato, si localizza facilmente l'errore perché solo certe operazioni hanno potuto agire su questi dati
- favorisce la creazione di componenti autonomi e "validati"
- minimizza la distanza concettuale tra problema e sua risoluzione in termini software

TIPO DI DATO ASTRATTO

Entità simili possono essere considerate "dello stesso tipo" (o della stessa classe), anche se con attributi specifici diversi.

ESEMPIO: il contatore

Approccio classico:

```
main()
{
…
int cont;

cont++;
cont--;

if (cont == MAX) {….}

cont = cont*cont;
/* puó avere senso per un intero ma non ha alcun senso
per una entità contatore */
}
```

PROBLEMI

- dov'è l'entità contatore?
- Dove sono le operazioni ammesse sugli oggetti di tipo contatore?
- possiamo fare cose NON SENSATE sul contatore
- stiamo usando le mosse elementari della macchina C (integer e operatori di incremento), non stiamo usando delle categorie concettuali di tipo contatore! non possiamo ri-usare il concetto di contatore!

ESEMPIO: il contatore

TIPO DI DATO ASTRATTO "Contatore"

DATI (Attributi che caratterizzano un contatore):

```
valore_cont;
```

OPERAZIONI (quelle permesse su un tipo di dato contatore")

```
    void inc();
    void dec();
    int getValue();
```

Vorremmo Poter Fare:

```
main()
{Contatore cont; /* definisco un TDA Contatore*/
/* NEL LINGUAGGIO A OGGETTI SI DICE "ISTANZIARE" */

Contatore cont2; /* ne istanzio un'altro */

cont.inc(); /* invoco delle operazioni sul contatore */
cont.dec();

cont2.dec() /* le stesse operazioni le posso invocare
chiedendole all'altro contatore: sono dello stesso tipo,
ma ovviamente avranno attributi specifici diversi!*/

if(cont.getValue() > MAX) {….}

cont++; /* ERRORE: NON E' UN INTERO */

cont.valore_cont=cont*345;
cont.valore_cont++   /*   ERRATE   ENTRAMBE:   il   dato
valore_cont è accessibile solo tramite le operazioni di
cont */
}
```

ESEMPIO: studenti

Approccio Classico:

```
typedef struct Studente {
        char nome[20];
        char cognome[20];
        int matricola;
        int num_esami_dati;
        struct esami[29] {char nome[20]; int voto;}
                    }

void nuovo_esame_dato(Studente s, char *nome_esame, int
voto_preso)
{
   strcpy(s.esami[s.num_esami_dati].nome, nome_esame);
   strcpy(s.esami[s.num_esami_dati++].voto, voto_preso);
}
/* e altre funzioni varie…..*/

main()
{
Studenti s1, s2;


s1.nome = "pippo"; s2.cognome = "rossi";
/* NON BELLO: COSI' ACCEDO AI DATI INTERNI DI STUDENTE*/

/* POI POSSO FARE */
new_esame(s1, "Sistemi Operativi", 18);

/* MA ANCHE */
s1.esami[4].voto++; /* NON DOVREI! NON HA SENSO! Non e'
una operazione ammissibile sul tipo di dato*/

                              }
```

ESEMPIO: studenti

DATI:

```
char nome[20];
char cognome[20];
int matricola;
Cont num_esami_dati;
struct esami sostenuti {char * nome; int voto};
```

OPERAZIONI (alcune pensabili tra quelle ammissibili)

```
void nuovo_esame_dato(char *nome_esame, int voto_preso);
Int calcola_media(void);
void iscrivi_anno_successivo();
```

Vorremmo Poter Fare:

```
main()
{
Studenti s1("Franco Zambonelli"), s2("Donald", "Duck");


s1.nuovo_esame(s1, "Sistemi Operativi", 28);

}
```

GROSSO CAMBIAMENTO DI PARADIGMA:

NON si richiede a funzioni di operare su dati

```
nuovo_esame(s1, "Sistemi Operativi", 28);
```

MA si richiede alle entità di eseguire delle funzioni

```
s1.nuovo_esame(s1, "Sistemi Operativi", 28);
```

MODELLO CLIENTE/SERVITORE

- Dato un problema, basarsi su un insieme di *centri di servizio*, ognuno capace di eseguire un ben preciso *insieme di attività*

Spostamento del fuoco
- NON chiamare una funzione che elabori dei dati MA
- chiedere a una entità software di svolgere un servizio

Esempio:
dato un vettore, NON invocare una procedura di ordinamento, MA chiedere a una entitá ordinatrice di farci il servzio di ordinarci il vettore

Esempio:
dato un file, non invocare una funzione che scriva dati sul file, ma chiedere al file stesso di svolgere il servizio relativo alla scrittura di file su di esso

- I clienti *non conoscono* l'organizzazione interna dei centri di servizio, e *non possono* accedere *direttamente* a essa
  - non interessa l'algoritmo (come sono fatte le cose) ma interessa che le cose vengano fatte
  - i clienti sono indipendenti da come sono fatti i servitori: facilita la manutenzione!

Nell'esempio precedente:

- il main chiede servizi allo Studente s1, il quale chiede servizi a un Contatore cont;

COSA VORREMO?

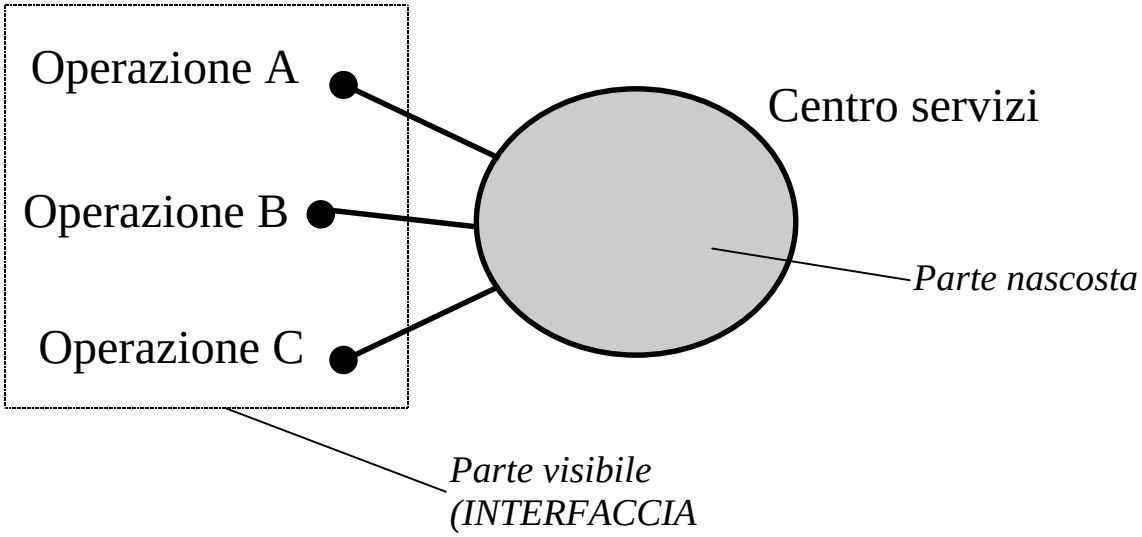
UN LINGUAGGIO PER DEFINIRE E ISTANZIARE ADT, CHE POSSONO INTERAGIRE TRA LORO SOLO TRAMTE RICHIESTE DI SERVIZIO

```
ADTdef Contatore {
    int valore_cont;  /* struttura dati interna */
    /* operazioni ammesse */
    void inc() {valore_cont++;}
    void dec() {valore_cont--;}
    int getValue() {return valore_cont;} }
ADTdef Studente {
    /* struttura dati interna */
    char nome[20];
    char cognome[20];
    int matricola;
    Contatore num_esami_dati;
    struct esami[29] {char nome[20]; int voto;}
    /* operazioni ammesse */
    void nuovo_esame(char *nome_esame, int voto_preso)
           { int cont = num_esame_dati.getValue();
             esami[cont].nome = nome; …}
    …   }

main()
{Studente s1;

s1.new_esame("Fondamenti II", 30);
/* corretta */
new_esame(Fondamenti II", 18);
/*   ERRATO:   a   quale   entità   stiamo   richiedendo
l'operazione?*/
s1.matricola++;
/* ERRATO non possiamo agire sulla struttura dati interna
se non tramite le operazioni ammesse */
}
```

MA QUESTO SAREBBE ESATTAMENTE UN LINGUAGGIO DI PROGRAMMAZIONE AD OGGETTI!

IL CONCETTO DI OGGETTO

- Un oggetto come ***astrazione di dato*** o ***centro di servizi***
- con una *parte visibile* → *INTERFACCIA*
- e una *parte nascosta*



LE OPERAZIONI DI INTERFACCIA RAPPRESENTANO:

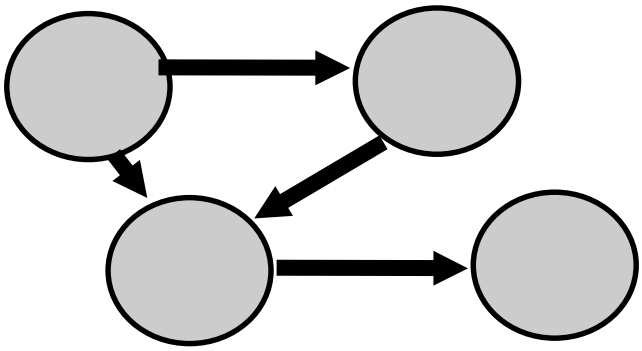
- I servizi che gli altri oggetti possono richiedere all'oggetto
- Modalità di trattamento dei dati dell'oggetto o di accesso ai dati

LA PARTE NASCOSTA CONTIENE

- **DATI**
  - Gli attributi che caratterizzano l'oggetto
  - Lo stato dell'oggetto
  - Sono accessibili solo attraverso richieste di servizio
- **OPERAZIONI PRIVATE**
  - Sfruttate dalle operazioni pubbliche ma non servizi esterni

ARCHITETTURA DI UN SISTEMA A OGGETTI

- Un ***insieme di oggetti***
- che **intergiscono gli uni con gli altri**
- senza conoscere nulla delle rispettive rappresentazioni concrete



DESCRIZIONE DI UN SISTEMA A OGGETTI

- Ogni oggetto appartiene a ("è istanza di") una data ***classe***
- **La classe *racchiude e incapsula la specifica*** di
  - **struttura** dell'oggetto (dati)
  - **comportamento** dell'oggetto (operazioni)
- Le classi ***integrano ed estendono il concetto di* ADT**
- e possono essere correlate tra loro → *tassonomie* di ***ereditarietà*** che permettono il collegamento tra classi simil per raggruppare le proprietà simili e, in ultima analisi, ri-usare facilemente il codice comune alle classi correlate e ri-definire velocemente nuove classi

VANTAGGI DELLA PROGRAMMAZIONE AD OGGETTI

Facilitazione di costruzione *cooperativa* di software:

- diverse persone sviluppano diverse classi
- ogni programmatore può semplicemente verificare il comportamento delle sue classi istanziandone oggetti e verificandone il comportamento in risposta a richieste di servizio
- unico accordo necessario per integrare il tutto in un unico sistema finale: definire le interfacce delle classi

Facilitazione della *gestione e manutenzione*

- Se vi sono errori sui dati in un oggetto, è facile scoprire dove si trova l'errore, perché (siccome i dati non sono visibili all'esterno dell'oggetto) esso non potrà essere che all'interno dell'oggetto che gestisce quei dati
- le modifiche a una classe non rendono necessario modificare il resto del programma (le altre classi) a meno che non venga modificata l'interfaccia

Supporto a progettazione e sviluppo *incrementali*

- si possono definire nuove classi sfruttando il codice di classi già esistenti

Possibilità di *rapida prototipazione*

- non importa che una classe sia completamente definita per poter iniziare a "testare" il funzionamente degli oggetti di quella classe.

REALIZZAZIONE ADT IN LINGUAGGI NON AD OGGETTI

Nei linguaggi “classici”, non a oggetti, è comunque possibile seguire una “filosofia di progettazione del software” a oggetti. Però le limitazioni del linguaggio rendono l'approccio o "sporco" o limitato dal punto di vista espressivo.

In C, due approcci possibili:

- ADT realizzati tramite ***typedef e variabili*** (e/o puntatori)
- Atrazioni di Dato realizzati tramite ***moduli*** (files)

I° Caso: ***tipo di dato astratto*** definito da `typedef`

- **gli oggetti sono variabili** del tipo definito da `typedef`
  → il cliente può *crearne tante istanze* quante desidera
- **le operazioni sono funzioni**
  fra i cui parametri figura **il nome della variabile-oggetto** (istanza) su cui devono agire

Conseguenza:
il cliente deve *trasferire esplicitamente l’oggetto* ai suoi servitori (funzioni che realizzano le operazioni dell’oggetto)

II° Caso: ***astrazione di dato*** realizzato tramite moduli o file

- il C *non fornisce un costrutto “modulo” nel linguaggio*
- si usano i **files** come **contenitori**, sfruttandoli
  - *sia* per separare interfaccia e implementazione (`.h` / `.c`)
  - *sia* come (elementari) *meccanismi di protezione*

Conseguenza:

la *separazione concettuale* interfaccia/implementazione diventa una *separazione fisica*
→ *i clienti includono lo header* per usare l'ADT

ES. 1: UN CONTATORE come ADT

counter.h...

```
typedef Contatore int;

int  getValue(Contatore);
void setValue(Contatore*, int);
void inc(Contatore *c);
void dec(Contatore *c);
```

.... e counter.c

```
#include “counter.h”
int  getValue(Contatore c){return c;}
void setValue(Contatore* c, int v) {*c=v;};
void inc(Contatore *c){ (*c)++;}
void dec(counter *c){ (*c)--;}
```

Uso:

```
#include “counter.h”
void main(){
  Contatore c1, c2;
  setValue(&c1, 10); setValue(&c2, -31);
  printf("Valore di c1 = %d", getValue(c1));
  inc(&c2); dec(&c1);
}
```

PRO:

- Possibilità di definire e usare *più oggetti* di tipo counter

CONTRO:

- Necessità di passare puntatori
- Rischio di usare oggetti *non inizializzati* (se ci si dimentica setValue...)
- Possibilitá di fare operazioni non ammesse sul contatore, poiché c’è libero accesso alla variabile. Es.

  Contatore *=Contatore;

ES. 2: UN CONTATORE come singola astrazione di dato

`count.h`...

```
int  getValue(void);
void setValue(int);
void inc(void);
void dec(void);
```

.... e `count.c`

```
static int Contatore = 0;

int  getValue(void){return Contatore;}
void setValue(int v) {Contatore=v;};
void inc(void){ Contatore++;}
void dec(void){ Contatore--;}
```

Uso:

```
#include "count.h”
void main(){
  setValue(10);
  printf("Valore = %d", getValue());
  inc();
}
```

PRO:

- Non è necessario definire variabili e passare oggetti o puntatori
- Il contatore è protetto, essendo inaccessibile dall’esterno del file `count.c` in modo diretto
- Solo le operazioni dentro definite dentro al file count.c possono accedere a cont, non c’è rischio di fare operazioni errate!

CONTRO:

- E’ impossibile definire e usare *più oggetti* contatori: non c'e' il tipo di dato astratto! E’una *singola astrazione di dato*
- C’è *un solo contatore*, già esistente, e si può usare *solo quello!*

ADT e OPERAZIONI

Operazioni = meccanismo per disaccoppiare i clienti dalla rappresentazione degli oggetti

Classificazione delle operazioni	*Categorie*
dal punto di vista di **chi le usa**	• costruttori • selettori • trasformatori • predicati • ....
dal punto di vista di **chi le realizza**	• primitive / non-primitive • operazioni di costruzione / operazioni di configurazione • operazioni private / pubbliche

Il punto di vista dell'utente

- operazioni di costruzione (*costruttori*)
- operazioni di selezione di sotto-componenti (*selettori*)
- operazioni di verifica di proprietà (*predicati*)
- operazioni di trasformazione

La presenza o meno di certe categorie di operazioni crea differenti ***categorie di oggetti*:**

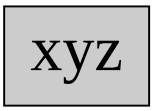
- **oggetti atomici / oggetti composti**
  - gli oggetti atomici *non hanno selettori*
- **oggetti privi di stato / oggetti dotati di stato**
  - gli oggetti privi di stato *non hanno trasformatori*

L’ADT "PAROLA": presenza di trasformatori...?

LA DICOTOMIA VALORI / CONTENITORI

Esempio: “una parola è una sequenza di caratteri (non-separatori)”

una parola



una parola diversa

Problema: si può modificare una parola?

Due alternative:

- concetto di “parola” come *contenitore di caratteri*
  → sono possibili modifiche al contenuto
  (ma i singoli caratteri sono valori...)
- concetto di “parola” come *valore*
  → non sono possibili modifiche al contenuto
  → non ci sono trasformatori (attenzione all’assegnamento!)

Questa tematica si ritrova, *nascosta*, nella più “semplice” (?) delle operazioni: l’assegnamento.

L-VALUE e R-VALUE

```
x = x + 2
```

- il simbolo **x** denota
  - un **valore** (intero) quando compare a destra dell’ = (*r-value*)
  - un **contenitore** quando compare a sinistra dell’ = (*l-value*)
- solo un l-value può comparire a sinistra dell’ =

L’ADT PAROLA: il punto di vista dell’implementatore

- operazioni di costruzione / di configurazione
- operazioni primitive / non-primitive
- operazioni private / pubbliche

Costruzione:

- un’operazione di *definizione*, che alloca memoria e eventualmente inizializza

Primitive:

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

Non primitive:

- rimangono inalterate *anche se cambia la rappresentazione*
- purché le primitive restino formalmente identiche

Protezione

- operazioni che tutti possono eseguire (*pubbliche*)
- operazioni *private*, che solo altre operazioni possono usare
  → *come garantirlo?*

Tipicamente esistono:

- primitive visibili
- non-primitive visibili
- primitive private (ad uso esclusivo del servitore stesso, per realizzare altre operazioni più importanti in modo indipendente dalla rappresentazione)
- non-primitive private