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?***

Software Ingegnerizzato … ?

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

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**

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

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?
- **...**

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**

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?

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***

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**

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 *scor-relati*: nulla indica che le une agiscano sugli altri**
- **devono essere sviluppati da gruppi, ma**
  - **la decomposizione funzionale e il disaccoppia-mento dati/funzioni non agevolano la decompo-sizione del lavoro**

(segue)

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 modo 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***

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**

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***

LA “CRISI GESTIONALE”

Programmi di medie dimensioni

- **individu**
- **l’impatt** **sa del non-ac** **fun-zioni o** **cata.**

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***

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

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*)**

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)"

PRODUZIONE DEL SOFTWARE: COSA OCCORRE?

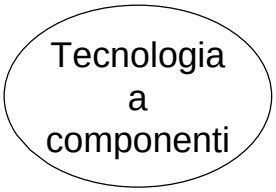
- **Concetti e Metodologie**
  per pensare

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

PRINCIPI STRUTTURALI

- **Information hiding, incapsulamento**
- **Località**
- **Parametricità**

- ***Astrazione* vs. *rappresentazione***
- **"*Cosa fa*" vs. "*come lo fa*"**
  - a livello di "dati"
  - a livello di "elaborazione"



INTERFACCIA e IMPLEMENTAZIONE

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

  di un ente computazionale

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*

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

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

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**

RAPPRESENTAZIONE & ASTRAZIONE

Due grandi approcci:

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

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**

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?**

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**

MODULI

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

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**

ESEMPIO: IL CONTATORE

Scopo: def[illegible]e

Specifica:

- un comp[illegible]i istante da[illegible]
- 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*

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"

ESEMPIO: IL CONTATORE

Due possibilità:

- **definire in un *modulo* la *singola astrazione di dato "contatore"* e poi usarlo "così com'è"**
- **def… "c… e p… di…**

- **Pro: possibile garantire vero incapsulamento**
- **Contro : si può definire <u>un solo</u> oggetto "contatore" (l'oggetto *coincide col modulo* che lo realizza)**

ESEMPIO: IL CONTATORE

D…

- **…**

- **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"**

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);
```


IL CONTATORE COME MODULO

- **D**
- **con operazioni che ag**
  ***implicitamente* su ess**

Le operazioni *non hanno alcun parametro* perché ora agiscono tutte, implicitamente, sull'unico contatore esistente: quello rappresentato dalla variabile statica `cont`

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

CONTATORE COME MODULO: USO

- **Si importano *solo* le dichiarazioni delle funzioni (`mcounter.h`)**
- **Si <u>usa semplicemente</u> il contatore "racchiuso" nel modulo *senza dover definire alcunché***

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


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***

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) "contator**

```
type
```

- **...con le sue operazioni:**

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

`reset` e `inc` devono *cambiare lo stato* del contatore → necessario passare *un puntatore*

`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?**

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);
}
```

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;
```


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;
}
```

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.**
- ***per il cliente non cambia interfaccia***

ok **solo** per numeri fino a 20

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);
}
```

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*

L'APPROCCIO DEGLI ADT

- **Consente di *separare interfaccia e***

Rispettare il *protocollo di accesso* stabilito dell'interfaccia è, per il cliente, *un atto volontario*

dalla
(servitore)

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*

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

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?

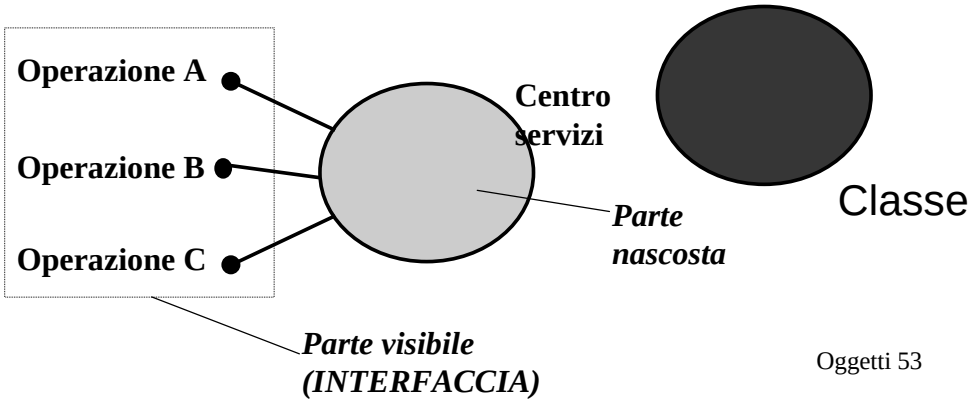
IL CONTATORE: USO ESTERNO

- 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);
}
```

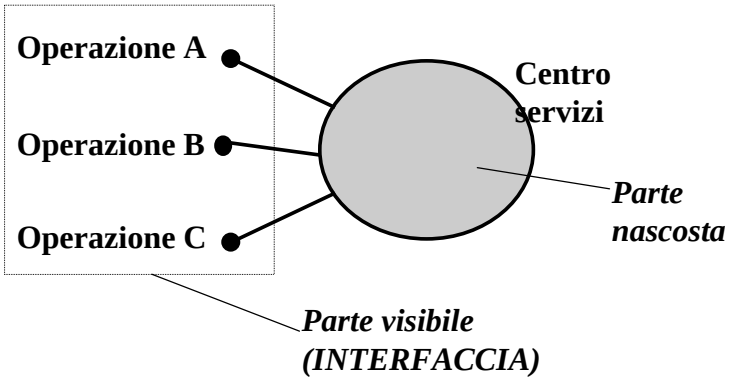
OGGETTI come ADT

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



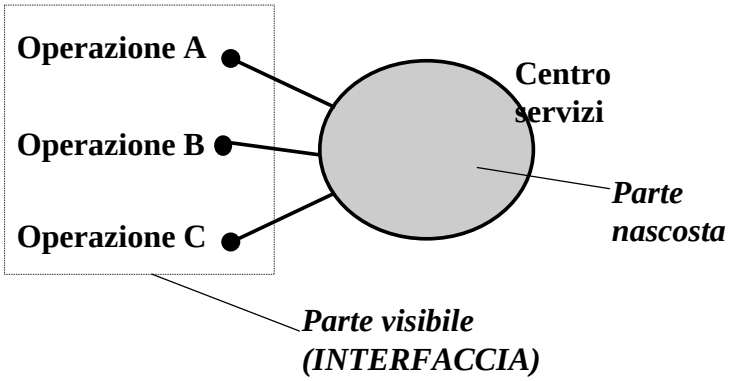
IL CONCETTO DI OGGETTO

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



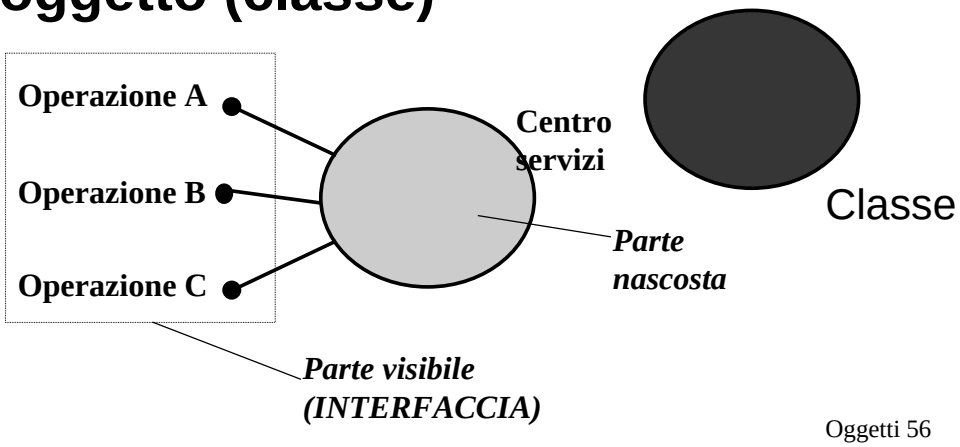
IL CONCETTO DI OGGETTO

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



IL CONCETTO DI OGGETTO

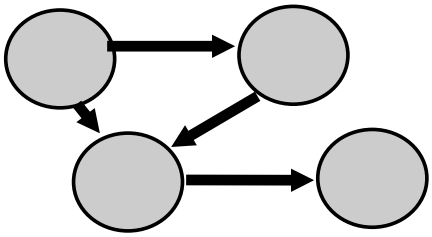
Ogni cliente può *creare* (*istanziare*) *tanti oggetti quanti gliene occorrono* a partire da una sorta di "modello" dell'oggetto (classe)



SISTEMI A OGGETTI

L'architettura di un sistema a oggetti:

- **un insieme di oggetti che *interagi-scono gli uni con gli altri***
- **senza conoscere *nulla* delle rispet-tive rappresentazioni concrete**
- **Modello di intera-zione a *"scambio di messaggi"***



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**

LE PROPRIETÀ DI UN OGGETTO

- **Un oggetto possiede *stato, funziona-mento* 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**

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***

RELAZIONI TRA OGGETTI

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

EREDITARIETÀ

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

EREDITARIETÀ

- **Una *relazi***
- **Ereditariet**
  condivide
  mento defi
- **Ereditarietà**
  condivide la struttura e/o il fu
  mento definito *in varie alt*

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)**

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.**

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**

QUALI OPERAZIONI ?

Classificazione delle operazioni	*Categorie*
dal punto di vista di **chi le usa**	• costruttori/distruttori • 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 UTENTE

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

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.

IL PUNTO DI VISTA DELL'IMPLEMENTATORE

Configurazione: un'operazione che inizializza un oggetto

- **L'implementat**
  ***classificazioni* in relazione all'a**
  intende mettere in luce
  - **operazioni di costruzione / configurazione**
  - **operazioni primitive / non primitive**
  - **operazioni private / pubbliche**
- C*operano su*
  *d*ll'altro
  ivate o pubbliche,
  ubblici, etc.

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

IL PUNTO DI VISTA DELL'

- **L'implementa**
  classificazio
  intende mette
  - **operazioni di <u>costruzione</u> / <u>confi</u>zione**
  - **operazioni <u>primitive</u> / <u>non primitive</u>**
  - **oper / <u>private / pubbliche</u>**
- *operano su*
  o all'altro
  e private o pubbliche,
  pubblici, etc.

Non primitive:
- operazioni *<u>indipendenti</u> dalla rappresentazione*
- <u>non accedono</u> 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*

IL PUNTO DI VISTA DELL'IMPLEMENTATORE

- **L'implementatore, in**
  ***classificazioni* in rela**
  intende mettere in lu
  - **operazioni di <u>costruzione</u> / <u></u>gurazione**
  - **operazioni <u>primitive</u> / <u>non p</u>mitive**
  - **operazioni <u>private</u> / <u>pubbliche</u>**
- cazione *operano su*
  rispetto all'altro
  primitive private o pubbliche,
  privati e pubblici, etc.

Pubbliche:
tutti possono invocarle per operare sull'oggetto

Private:
possono essere invocate *solo da altre operazioni* del medesimo oggetto