

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”?

SOFTWARE “BEN FATTO” ... ?

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

LA 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, Pattern**
 - fonti di ispirazione

LO SVILUPPO STORICO

- 1950-1970:
 - cosa significa computare?
 - quali mosse primitive deve avere un linguaggio di programmazione?
- 1970-1980:
 - quali principi di organizzazione del software?
 - basta la programmazione strutturata?

LO SVILUPPO STORICO

- 1980-1990:
 - perché il modello a oggetti è importante?
 - vi sono alternative alla classificazione?
- 1990-2000:
 - quali ripercussioni se la piattaforma computazionale diventa *una rete*?
 - come recuperare 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 come “contenitori di descrizioni”

**non bastano per ottenere software
modulare e sviluppabile in modo
incrementale.**

Perché?

LA “CRISI DIMENSIONALE”

Il cambio di dimensioni del problema *cambia*

- 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)
alle entità del mondo reale da modellare
e agli strumenti che consentono il lavoro di gruppo

LA “CRISI DIMENSIONALE”

Programmi di piccole dimensioni

- 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 DIMENSIONALE”

Programmi di grandi dimensioni

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

(segue)

LA “CRISI DIMENSIONALE”

Programmi di grandi dimensioni (segue)

- 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*

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’impatto non-acc
- zioni 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.*

L'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.*

EFFICIENZA... MA NON SOLO

- *Premature optimization is the root of all evil*
 - Donald E. Knuth
- *Make it work first, before you make it work fast*
 - Bruce Whiteside
- *Make it fail-safe before you make it faster*
- *Make it clear before you make it faster*
 - Kernighan A. Plaugher

QUALE PROGETTO?

Spesso

- si studia un linguaggio
- si “**pensa**” in termini del linguaggio conosciuto, usando
 - i costrutti del linguaggio
 - l'information space del linguaggio
 - metodologie legate al linguaggio (*idiomi, framework*)
 - schemi standard (*pattern*)

INFORMATION SPACE

- Un insieme di **dimensioni, astrazioni** e **concetti** usati per esprimere (un modello di) un sistema software
- L'information space proposto dalle *macchine di Turing* (o di *Von Neumann*) **non introduce concetti adatti a trattare sistemi complessi.**

COSA SI DOVREBBE FARE?

1. Definire i requisiti
2. Fare l'analisi e il progetto adottando un *information space* e un modello adeguati
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

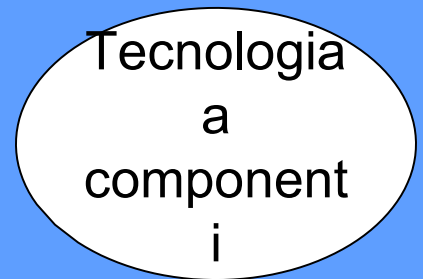
- Information hiding, **incapsulamento**
 - Località
 - Parametricità
-

- *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, nascondendone
 - l'organizzazione interna
 - i dettagli di funzionamento
- *L'implementazione* esprime
 - la *rappresentazione dello stato interno*
 - il codicedi un ente computazionale

ASTRAZIONE

- Si focalizza sul **funzionamento osservabile** di un ente
- *”Abstraction helps people to think about what they are doing”*
 - 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 ente
- “*Encapsulation allows program changes to be reliably made with limited effort*”

Astrazione e incapsulamento sono concetti *complementari*.

INDIPENDENZA DALLA RAPPRESENTAZIONE

- Incapsulamento comporta che la *rappresentazione concreta* di un dato può essere *modificata*
- *senza che vi siano ripercussioni sul resto del programma.*

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 & ATRAZIONE

- 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 **tipi di dati astratti (ADT)**
- sfruttare i ***moduli*** per definire ***singole astrazioni di dato***

Questi approcci hanno *caratteristiche complementari*.

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

- 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 rappresentazione concreta*
- Però, è adatto a 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: definire

Specifica:

- un componente istante da cui si può accedere
- 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 il **tipo di dato astratto (ADT)** “contatore”
e poi sfruttarlo per *creare “oggetti”* di tipo “contatore”
- definire in un **modulo** la **singola astrazione di dato “contatore”**
e poi usarlo “così com’è”.

ESEMPIO: IL CONTATORE

Due possibilità:

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

ESEMPIO: IL CONTATORE

- Pro: possibile garantire **vero incapsulamento**
 - Contro : si può definire **un solo oggetto “contatore”** (l'oggetto coincide col modulo che lo realizza)
- di tipo **contatore**
- definire in un **modulo** la **singola astrazione di dato “contatore”** e poi usarlo “così com'è”.

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”

type

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!**

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.

ok solo per numeri fino a 20

- ***ma per il cliente non cambia niente!***

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 “ill-behaved” 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*

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** Le operazioni *non hanno alcun parametro* perché ora agiscono tutte, implicitamente, sull'unico contatore esistente: quello rappresentato dalla variabile statica `cont`.
- **C** Con operazioni che agiscono *implicitamente* su esso.

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

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

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

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();
}
```

L'APPROCCIO A MODULO

- Separa *interfaccia* e *implementazione*
- Rende il cliente *indipendente dalla struttura interna dell'ADT* (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

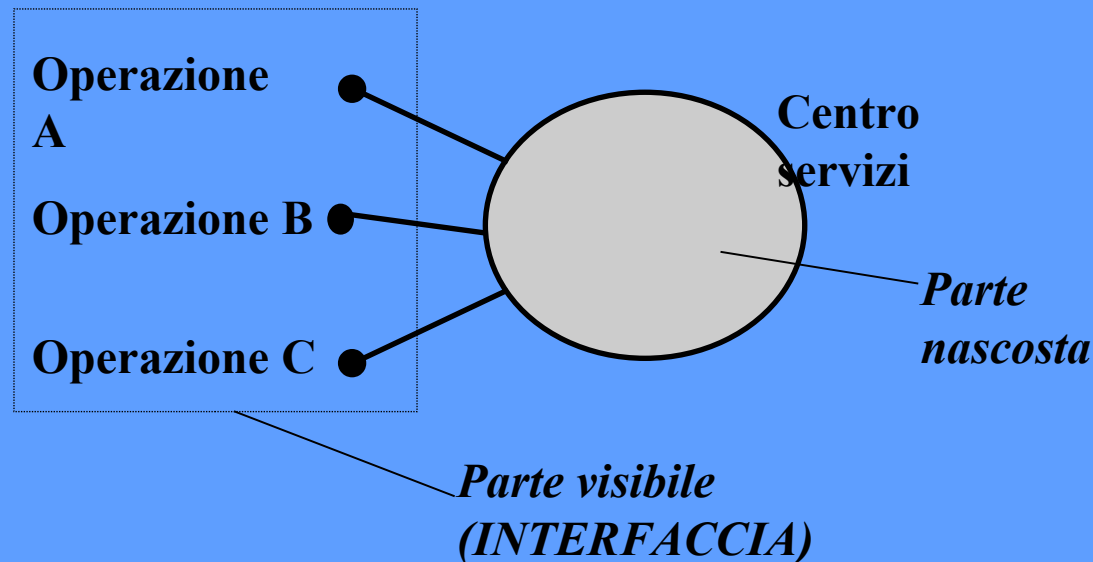
L'OBIETTIVO

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

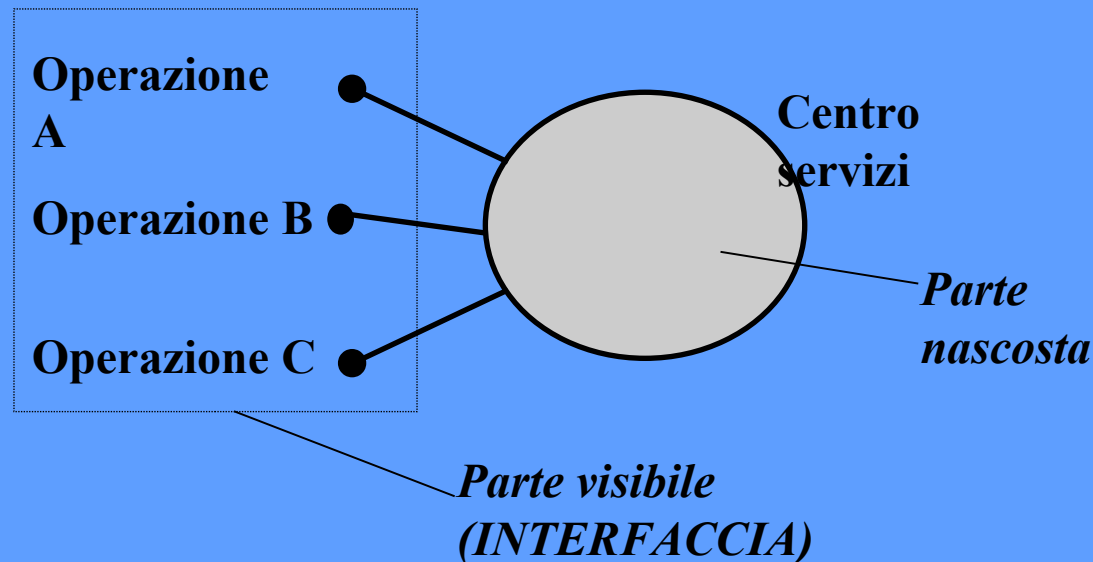
IL CONCETTO DI OGGETTO

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



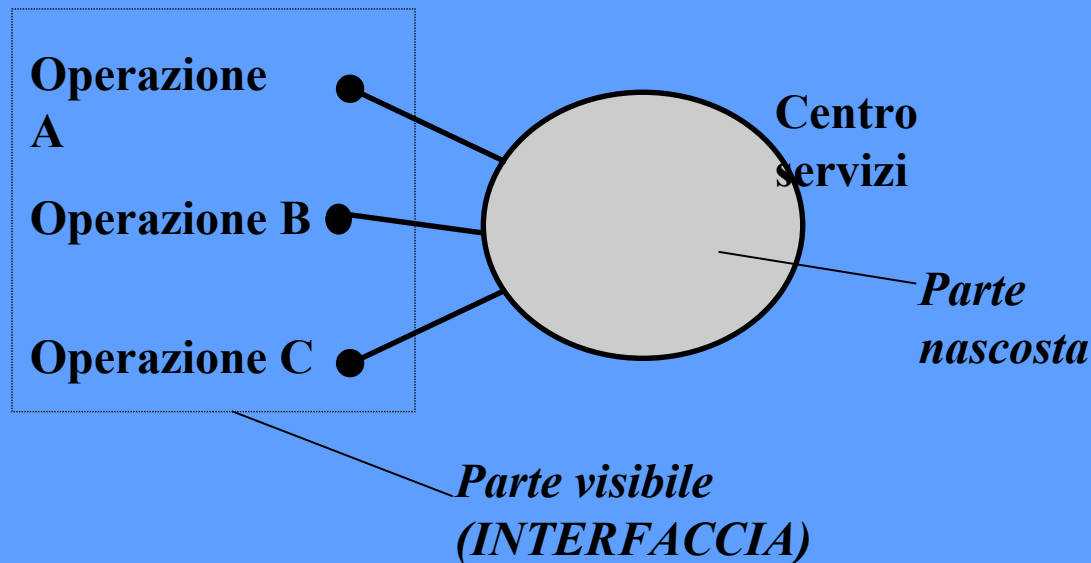
IL CONCETTO DI OGGETTO

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



IL CONCETTO DI OGGETTO

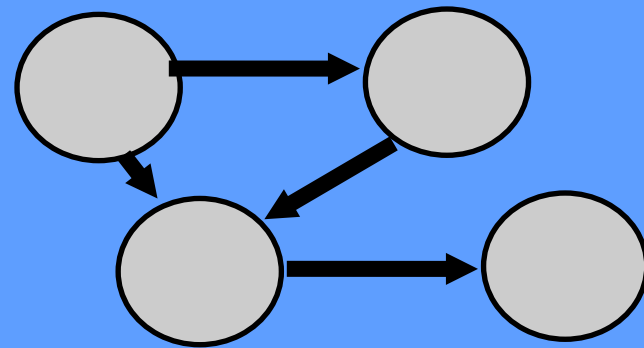
Ogni cliente può *creare* (*istanziare*)
tanti oggetti quanti gliene occorrono
a partire da una sorta di “modello”
dell’oggetto (*classe*)



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*”



ELABORAZIONE E INTERAZIONE

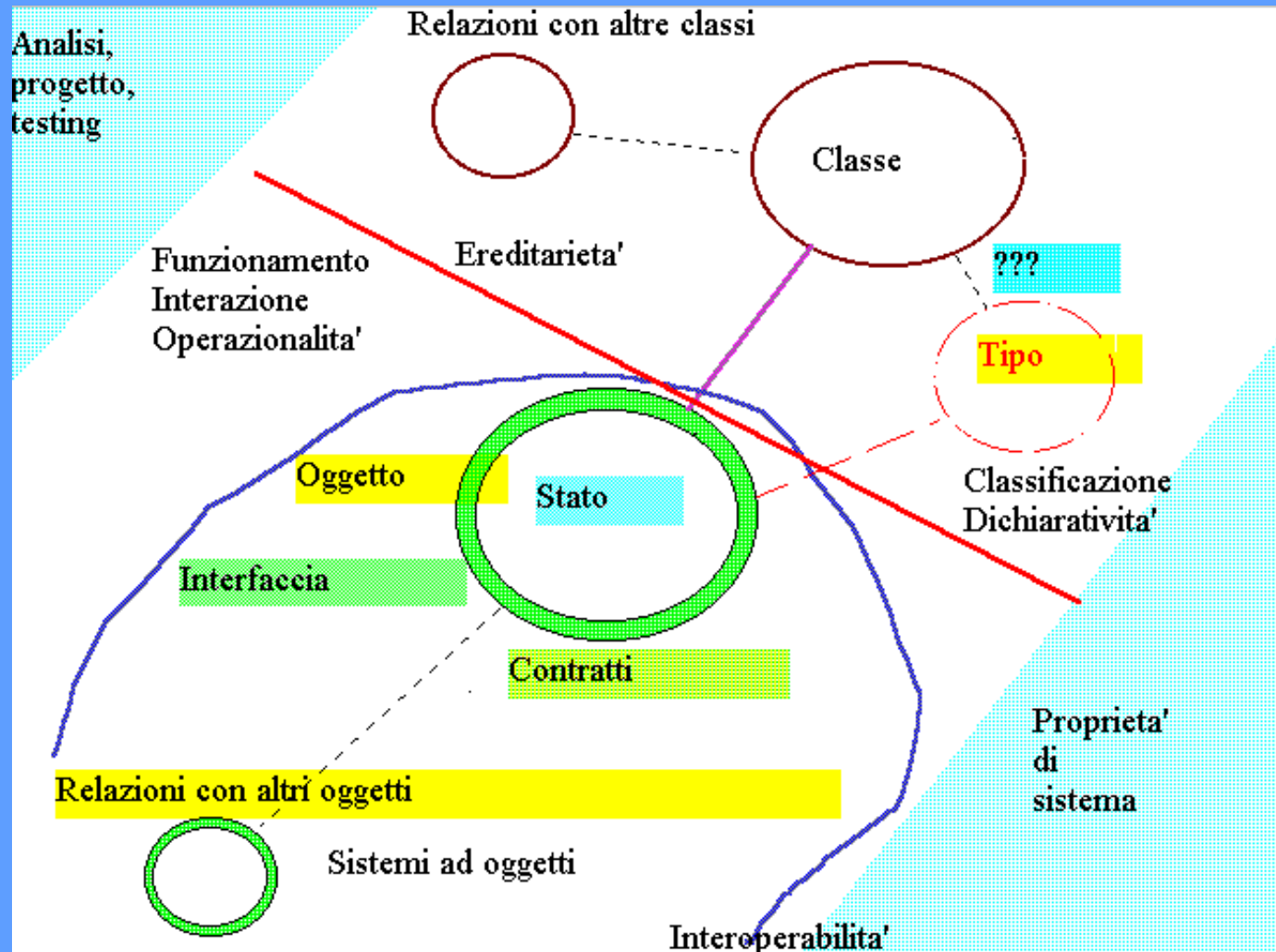
Peter Wegner:

- “gli **oggetti** sono la classe più usata di ***interaction machines***”
- “la **tecnologia object-based** è divenuta dominante perché **interattiva**, e quindi per ciò stesso **più espressiva** di una specifica algoritmica.”

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 un *information space* molto ricco e orientato alla gestione della complessità.

INFORMATION SPACE OO



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.

IL CONCETTO DI CLASSE

- Una **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 vs. ALGORITMI

- Gli algoritmi hanno un funzionamento osservabile che *non dipende dal tempo*.
- Gli oggetti posseggono sia proprietà trasformazionali sia *proprietà temporali*.
- Il funzionamento osservabile di un oggetto è descritto da ***interaction histories*** che specificano *i messaggi ricevuti* e *i messaggi inviati*.

OGGETTI vs. ALGORITMI

- Un algoritmo calcola eseguendo una sequenza finita di azioni interne.
- Un algoritmo è un **sistema off-line**, che specifica il flusso completo di una computazione **prima che essa inizi**.
- Un oggetto calcola attraverso un **pattern di interazioni** (*interaction history*) iniziate dall'esterno, *al di fuori del controllo dell'oggetto*.
- Un oggetto è un **sistema on-line**.

OGGETTI vs. ALGORITMI

- Un algoritmo è un *sistema chiuso*, non soggetto a interazioni con l'esterno durante il suo funzionamento.
- Problemi di natura algoritmica ammettono specifiche formali, sono implementati da programmi la cui correttezza può essere dimostrata e hanno una precisa nozione di complessità.
- Un oggetto è un *sistema aperto*.
- Problemi interattivi come la prenotazione dei posti aerei *non hanno specifiche complete*, né una definizione precisa di correttezza o di complessità.

OGGETTI vs. ALGORITMI

- Il funzionamento di algoritmi può essere specificato da un *contratto* che *si conclude in un dato momento*, come un ***contratto di vendita***.
- Un algoritmo trasforma un ingresso prespecificato (*pre-condizioni*) in una uscita (*post-condizioni*): se i clienti forniscono un valore che soddisfa le *precondizioni*, si garantisce un risultato che soddisfa le *postcondizioni*.
- Gli oggetti richiedono la specifica di un *contratto* che *si prolunga nel tempo*, come un ***contratto di matrimonio***.

OGGETTI vs. ALGORITMI

- Gli algoritmi hanno come *requisito primario* la **correttezza** e come requisito secondario l'**efficienza**.
- Gli algoritmi sono progettati mediante un raffinamento *top-down* delle specifiche.
- I **sistemi software** hanno come *requisito primario* un **basso costo del ciclo di vita**.
- Il progetto di un sistema ad oggetti parte da una *descrizione bottom-up* dell'ambiente di interazione.

RELAZIONI TRA OGGETTI

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

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*

EREDITARIETÀ

- Una *relazione*

- **Ereditarietà**
condivide
mento definito

- **Ereditarietà**

condivide la struttura e/o il funzionamento definito *in varie altre*

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 solo incapsulamento (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?*
- SÌ... 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 ?

<i>Classificazione delle operazioni</i>	<i>Categorie</i>
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 decomposizione

IL PUNTO DI VISTA UTENTE

- Dal punto di vista dell'utente, vi sono:
 - operazioni di costruzione (*costruttori*)
 - 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*

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

- L'implementatore, invece, opera *diverse classificazioni* in relazione all'aspetto che intende mettere in luce

- operazioni di costruzione / configurazione

Costruzione: un'operazione di *definizione*, che alloca memoria e eventualmente inizializza.

Configurazione: un'operazione che inizializza un oggetto.

IL PUNTO DI VISTA DELL'IMPLEMENTAZIONE

- L'implementazione **classifica** le operazioni in base a come esse intendono mettere a

Non primitive:

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

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

Primitive:

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

operano su
o all'altro

e private o pubbliche,
pubblici, etc.

IL PUNTO DI VISTA DELL'IMPLEMENTATORE

- L'implementatore, in *classificazioni* in relazione a *intende mettere in luce*
 - operazioni di costruzione / configurazione
 - operazioni primitive / non primitive
 - operazioni private / pubbliche
- **Private:**
possono essere invocate *solo da altre operazioni* del medesimo oggetto
- **Pubbliche:**
tutti possono invocarle per operare sull'oggetto
- *classificazione operano su*
rispetto all'altro
primitive private o pubbliche,
privati e pubblici, etc.