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 *scor-relati*: nulla indica che le una 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 DIMENSIONALE"

Programmi di grandi dimensioni (segue)

- 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 limi-tato 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 fun-zioni 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"

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.*

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"

Tecnologia a componenti

INTERFACCIA E IMPLEMENTAZIONE

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

di 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

- <u>Incapsulamento</u> significa che la *rappresentazione concreta* di un dato
  può essere modificata
- *senza che vi siano ripercussioni* sul resto del programma.

ESEMPIO

- <u>Tavola</u> con *rappresentazione concreta:*
  – mediante vettore, oppure
  – mediante file
- *l'interfaccia non varia*

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 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

<u>Scopo</u>: definire il concetto di *contatore*
<u>Specifica</u>:

- 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

<u>Scopo</u>: de[...]
<u>Specifica</u>:

Non importa come è realizzato *dentro*: importa il suo *<u>comportamento osservabile</u>*.

- un comp[...] istante d[...]
- 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

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:** mancanza di vero incapsulamento (chiunque può in realtà accedere allo stato interno)

ESEMPIO: IL CONTATORE

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

- 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) "contator[...]
  `type[...]`
- ...con le s[...]

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

```
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 *<u>irrilevante</u>* 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

<u>Implementazione</u>

- La struttura interna del contatore diventa *<u>rilevante</u>* quando giunge il momento di *realizzarlo.*
- Se <u>ora</u> 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

<u>Una implementazione alternativa</u>

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

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 <u>volontario</u>*

Se un cliente "ill-behaved" decide di *aggira-re le operazioni previste* e di accedere direttamente allo stato interno degli oggetti, *<u>nessuno può impedirlo</u>*

  – *conoscono la struttura interna* dell'ADT
  – e possono *violare il protocollo di accesso*

IL CONTATORE COME ASTRAZIONE DI DATO

- Definire il contatore come *<u>singola risorsa protetta</u>* 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 ASTRAZIONE DI DATO

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

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

CONTATORE COME ASTRAZIONE DI DATO: 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();
}
```

ASTRAZIONE DI DATO

- Separa *interfaccia e implementazione*
- Rende il cliente *indipendente dalla struttura interna del dato* (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*

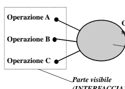
L'OBIETTIVO

L'ideale sarebbe:

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

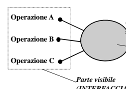
IL CONCETTO DI OGGETTO

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



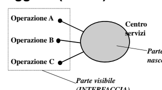
IL CONCETTO DI OGGETTO

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



IL CONCETTO DI OGGETTO

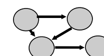
Ogni cliente può *creare* (*istanziare*) *tanti oggetti quanti gliene occorrono* a partire da una sorta di "modello" dell'oggetto (classe)



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 <u>*interattiva*</u>, e quindi per ciò stesso <u>*più espressiva*</u> 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.

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 <u>*istanza*</u> e <u>*oggetto*</u> sono intercambiabili.

IL CONCETTO DI CLASSE

- Una *classe* descrive la *struttura interna* e il *funzionamento* di un oggetto.
- <u>Oggetti appartenenti a una stessa classe</u> 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 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 *relazi...*
- Ereditariet... condivide ... mento defi...
- Ereditarietà ... condivide la struttura e/o il funziona-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 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 ?

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 UTENTE

- Dal punto di vista dell'utente, vi sono:
  - operazioni di <u>costruzione</u> (*costruttori*)
  - 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 *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 <u>costruzione</u> / <u>configurazione</u>
  - operazioni <u>primitive</u> / <u>non primitive</u>
  - operazioni <u>private</u> / <u>pubbliche</u>
- 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 <u>costruzione</u> / <u>configurazione</u>
- Q... *dimensioni dive...*
  - possono quindi ... operatori di cos...

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

Configurazione: un'operazione che inizializza un oggetto.

IL PUNTO DI VISTA DELL'IMPLEMENTATORE

- L'implementa... *classificazio...* intende mette...
  - operazioni di <u>costruzione</u> / <u>confi...</u>
  - operazioni <u>primitive</u> / <u>non primitive</u>
  - opera... <u>private / pubbliche</u>
- ... *operano su* ... all'altro
  - ... 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>gurazione</u>
  - operazioni <u>primitive</u> / <u>non p...mitive</u>
  - 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