Progettazione

- É il ponte fra la specifica e la codifica
- É la fase in cui si decide come passare da “che cosa” deve essere fatto a “come” deve essere fatto
- La sua uscita si chiama architettura (o progetto) del sw

Architettura = definizione del sistema sw in termini di componenti e loro reciproche interconnessioni

Obiettivi della modularizzazione

o, meglio, dei principi che la guidano:

- dominare la complessità (la somma delle complessità delle singole parti deve essere minore della complessità originaria)
- ridurre i tempi di produzione (grazie alla distribuzione e parallelizzazione del lavoro)
- favorire il riuso sistematico
- migliorare i fattori di qualità del sw

Componenti e interazioni: due livelli di astrazione

- Meccanismi
- Stili

Entrambi forniscono una vista statica (topologica) dell'architettura e talvolta la distinzione fra i due è sottile

Meccanismi

Consentono di rispondere alle seguenti domande:

- Che moduli?
- Quali interfacce per i moduli?
- Quali relazioni fra moduli?

Prospettive circa i meccanismi

- Metodologica: quali sono i criteri di scomposizione in moduli?
- Documentazione: come documentare il catalogo di moduli e relazioni?

Cos’è lo stile?

È ciò che caratterizza un’architettura rispetto a un’altra

Esempio tratto dal dominio della progettazione civile:

> Componenti di un’abitazione: stanze, corridoi, scale, seminterrati, mansarde, taverne, ecc.
>
> Stili di un’abitazione: disposizione tipica delle stanze, aspetto esteriore (es. classico, rustico, moderno), ecc.

Nel dominio della progettazione sw lo stile

- influenza l’individuazione dei moduli
- caratterizza l’interazione fra moduli che può spaziare da modalità semplici, quali chiamata di procedura e accesso a variabili condivise, a modalità complesse e semanticamente ricche, come protocolli client-server, diffusione di eventi asincroni e flussi in pipeline

Stili di progettazione

La comprensione di forme di progettazione comuni e un vocabolario condiviso, codificato nei manuali, sono tipici degli ambiti ingegneristici maturi
Gli stili di progettazione sw vanno in questa direzione, anche se in questo ambito c'è minore maturità

Meccanismi

Meccanismi: moduli

- Modulo = parte di un sistema sw che fornisce un insieme di servizi ad altri moduli
- Servizio = elemento computazionale che gli altri moduli possono usare

Meccanismi: interfacce

Un modulo consiste di:

- Corpo = implementazione e suoi segreti
- Interfaccia = insieme dei servizi esportabili, definisce un contratto fra il modulo e i moduli suoi utenti; gli utenti conoscono solo l'interfaccia di un modulo

Meccanismi: relazioni (tutte irriflessive)

- USA: un componente usa i servizi esportati da un altro
- È-UN-COMPONENTE-DI: descrive l'aggregazione di moduli in moduli di livello più altro
- EREDITA-DA: per sistemi OO

Relazione USA

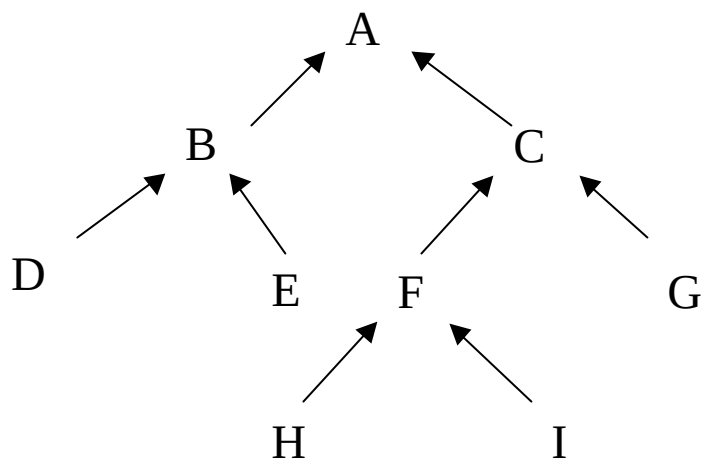
A *USA* B (si dice anche che A è un cliente di B) se:

- A accede ai servizi esportati da B
- Nel fornire i suoi servizi, A dipende da B (ad es. A chiama una routine esportata da B)

È un grafo

Relazione È-UN-COMPONENTE-DI

È una gerarchia



Relazione EREDITA-DA

- Il modulo che eredita può estendere quello ereditato
- L'erede può accedere – in tutto o in parte – ai segreti del progenitore

Principi di modularizzazione

Progetto per il cambiamento

- Anticipare i cambiamenti
- Non concentrarsi sulle esigenze attuali ma prevedere la loro evoluzione (prototipazione evolutiva)
- Pensare al programma come al membro di una famiglia

Cambiamenti: verosimiglianza

- Degli algoritmi (ad es. per ragioni di efficienza)
  - Es. da ordinamento per affioramento a ordinamento per selezione
- Delle strutture dati (sia per ragioni di efficienza, sia per cambiamenti dei requisiti)
  - Es. dati dell'utente: 17% dei costi di manutenzione
- Della macchina astratta sottostante
  - Es. periferiche hw, OS, DBMS, ecc. ($\rightarrow$ nuovi rilasci, problemi di portabilità, ecc.)
- Dell'ambiente
  - Es. anno 2000, euro
- Della strategia di sviluppo
  - Es. prototipo evolutivo
- Funzionali (cambiamenti dei requisiti)
  - Es. l'introduzione dell'applicazione genera nuovi bisogni e modifica quelli esistenti, oppure si scoprono errori nei requisiti

Cambiamenti: rischi

Passare da un progetto ordinato, ben documentato, riusabile, con codice pulito a un progetto contorto, non riusabile, con documentazione inconsistente, con codice “spaghetti”

Cambiamenti: dove e come

- L’interfaccia di un modulo è un contratto con i clienti e, come tale, deve essere stabile
- Se le parti modificabili sono quelle segrete del modulo, il loro cambiamento non deve avere effetti per i clienti
- Minimizzare il flusso di informazioni di un modulo verso i clienti
- Minimizzare la presenza della relazione USA

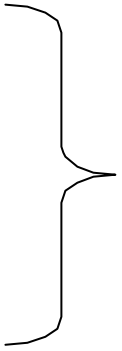
Cambiamenti: un esempio

Una TABELLA in cui è possibile inserire, cancellare, modificare e stampare voci (secondo un ordine prestabilito)

Interfaccia: INSERT, DELETE, MODIFY e PRINT

Si possono cambiare liberamente:

- Le strutture dati
- La politica di gestione dei dati (mantenere le voci ordinate oppure ordinarle solo prima di stamparle)
- Gli algoritmi (ad es. l'algoritmo di ricerca)



Questi sono i segreti di un modulo

Famiglia di programmi

Obiettivo del fatto di pensare al programma come al membro di una famiglia: progettare l'intera famiglia anziché ogni membro separatamente

Ad es. sistema di biglietteria

- Ferroviaria: formulazione interattiva del piano del viaggio (magari suddiviso in più tratte), scelta della classe e del posto da parte dell'utente, calcolo del prezzo, eventuale acquisto on-line, eventuale prenotazione, scelta fra modalità di pagamento diverse, eventuale recapito a domicilio
- Marittima: molte funzionalità sono simili ma alcune diverse (ad es. possibilità di portare l'auto al seguito)

Principi di modularizzazione

Occultamento delle informazioni (Parnas 1974): “define what you wish to hide and design a module around it”

- Un modulo è un’unità logica autocontenuta
- Entro questa unità è necessario distinguere fra ciò che un modulo fa per gli altri e come lo fa (i suoi segreti)
- Il modulo deve essere un firewall intorno ai suoi segreti
- I segreti devono essere incapsulati e protetti, ovvero l’accesso agli stessi deve essere filtrato dall’interfaccia

Altri principi e concetti chiave della modularizzazione

- Massima coesione e minimo accoppiamento
- Progetto per il riuso
- Scomposizione
- Astrazione
- Modularità
- Estensibilità
- Strutturazione a macchine virtuali

Astrazione

Tipo di modulo	Equivalente
Operazione astratta	Procedure della programmazione funzionale
Oggetto astratto (ad es. il modulo TABELLA) – Un modulo che incapsula una struttura dati – Esporta un insieme di operazioni – L'applicazione delle operazioni modifica lo stato dell'oggetto astratto	Classe della programmazione OO
Tipo di dato astratto – Un modulo che consente l'istanziazione di oggetti astratti	Interfaccia

Circa la relazione USA

Utilizzare solo una gerarchia di USA, non un grafo, perché è più facile

- da comprendere
- da sviluppare e verificare (se non è una gerarchia, si sviluppa un sistema in cui non funziona niente fino a quando non funziona tutto)

I livelli della gerarchia diventano i livelli di astrazione del sistema

Tecniche di modularizzazione

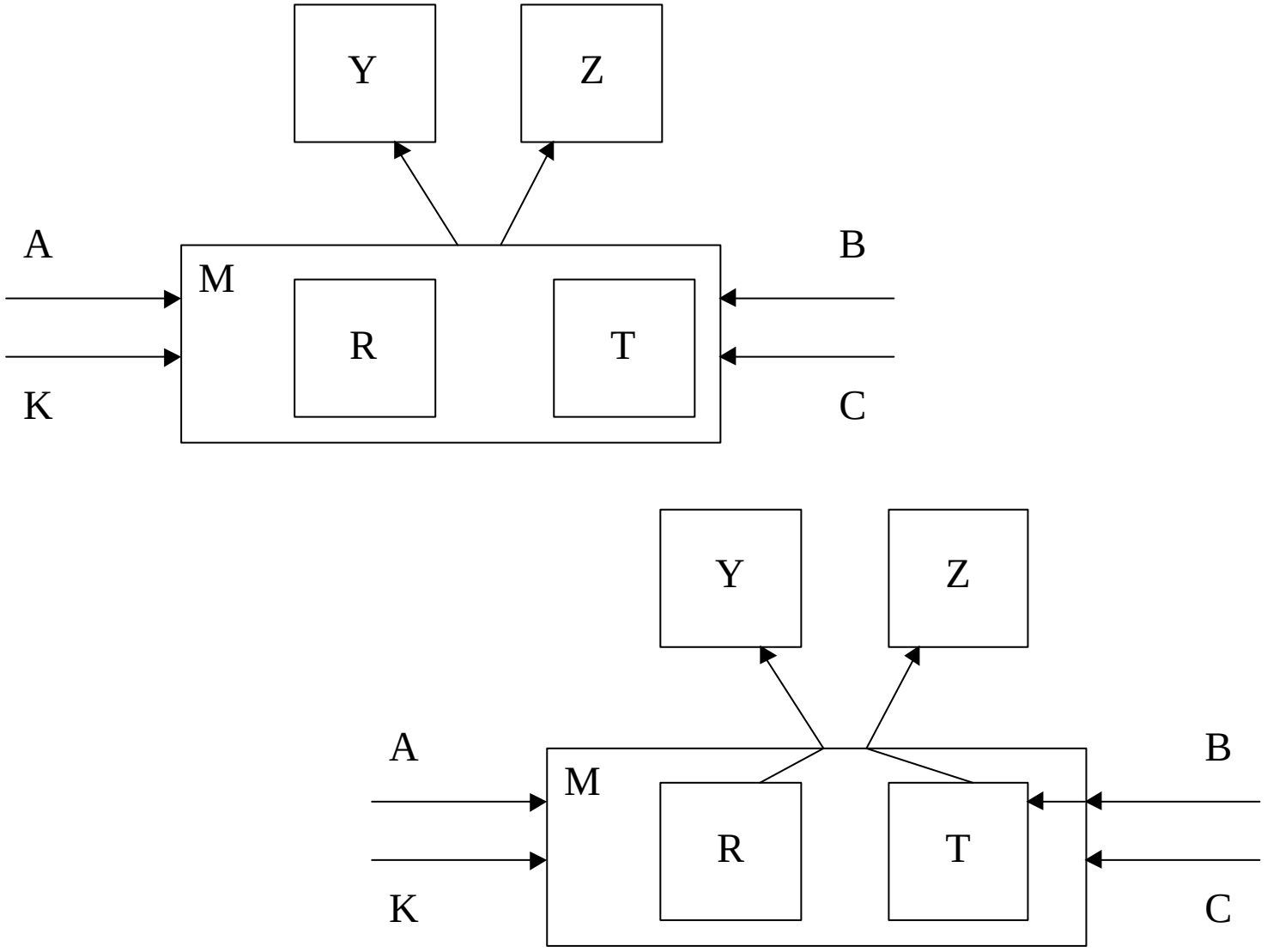
- top down
- bottom up
- miste

	Top down	Bottom up
Vantaggi	Scomposizione funzionale	▪ Sviluppo di moduli riusabili ▪ Flessibilità nel coordinamento dei moduli ▪ Incapsulamento di dati e algoritmi ▪ Elevata coesione e basso accoppiamento
Svantaggi	▪ Limitato incapsulamento ▪ Limitato riuso ▪ Interdipendenza tra i moduli di livello più basso ▪ Approccio puramente algoritmico al problema	Difficoltà di combinazione dei moduli per la costruzione dei moduli di più alto livello

Textual Design Notation (TDN)

module M;
uses Y, Z **imports** L, S;
-- importazione selettiva
exports var A: integer; K : record ...end;
type B : **array** (1..10) **of** real;
procedure C (D: **in out** B; E: **in** integer; F: **in** real);
-- qui c'è l'eventuale spiegazione in linguaggio naturale di ciò che sono
-- effettivamente A, B e C, nonché di possibili vincoli e proprietà che i clienti
-- devono conoscere; per es. potrebbe essere necessario che gli oggetti di tipo B
-- passati alla procedura C siano inizializzati dal cliente e non debbano mai
-- contenere tutti zero
implementation
-- qui ci sono eventuali commenti circa le ragioni che motivano le scelte di
-- modularizzazione, suggerimenti sull'implementazione, ecc.
is composed of R, T;
end M;

Graphical Design Notation (GDN)



Progettazione OO

- Un modulo (classe) può essere usato (relazione USA) da altri per generare istanze
- La classe da cui si eredita (relazione EREDITA-DA) fattorizza la parte comune (es. famiglia di programmi)
- Le variazioni (delta) sono definite nelle classi che ereditano
- L'ereditarietà aggiunge ulteriori interdipendenze fra moduli

Progettazione per contratto

- Tecnica sviluppata da Bertrand Meyer nel 1992 come caratteristica del linguaggio Eiffel
- Rifinisce un principio di progettazione noto, particolarmente adatto per la progettazione OO

Contratto = accordo fra un cliente e un contraente

Progettazione per contratto	Progettazione OO
Modulo contraente	Classe contraente
Contratto	Interfaccia della classe contraente
Modulo cliente	Classe cliente (che USA la classe contraente)

Un esempio di contratto nel mondo reale

	Obblighi	**Benefici**
Cliente	♦ Fornire un terreno di dimensioni minime prefissate ♦ Mettere a disposizione un punto di prelievo dall'impianto idraulico ♦ Pagare una certa somma secondo modalità di versamento prefissate	Disporre di una piscina funzionante, di dimensioni e caratteristiche note, allacciata all'impianto idraulico
Contraente	♦ Costruire una piscina funzionante, di dimensioni e caratteristiche note, effettuando l'allacciamento all'impianto idraulico ♦ Nessun obbligo se il punto di prelievo dall'impianto idraulico non è disponibile	Ricevere, secondo le modalità prefissate, i versamenti della somma di denaro pattuita

Il contratto per un metodo OO

	Obblighi	**Benefici**
Metodo cliente	Precondizioni = ciò che è richiesto dal metodo contraente	♦ Il servizio computazionale reso dal modulo contraente ♦ Conoscere le condizioni soddisfatte dai risultati, senza bisogno di controllarle
Metodo contraente	Postcondizioni = ciò che deve essere fornito dal metodo contraente	Non dovere considerare tutte le possibili configurazioni d'ingresso (principio di progettazione)

Precondizioni

Se `p` è la precondizione per il metodo `m`, o scriviamo (per ogni oggetto `x`)

```
if x.p
   then x.m
   else …{trattamento speciale}
```

o, ragionando sul programma, assicuriamo che `p` sia vera (per ogni oggetto `x`) prima della chiamata

- Scelta delle precondizioni = decisione di progetto per cui non esistono regole guida
- Precondizioni deboli implicano che tutte le complicazioni sono delegate al metodo ($\rightarrow$ difficoltà di sviluppo)

Postcondizioni

Servono a specificare cosa fa un'operazione senza dire come lo fa, cioè a separare l'interfaccia dall'implementazione

Invariante

Invariante = proprietà che ogni istanza di una classe deve soddisfare

- dopo la sua creazione
- sia prima, sia dopo ogni operazione compiuta sull'istanza stessa (ma non necessariamente durante)

Rappresenta un ulteriore obbligo che la classe deve soddisfare con la sua implementazione

Proprietà interne di una classe

Esempio: classe tabella

- Invariante: n_elementi ≤ capacità
- Operazione *inserisci(elemento)*
  Precondizione: n_elementi < capacità
  Postcondizioni:
  ✓*elemento* è presente nella tabella
  ✓n_elementi' = n_elementi + 1

Questi tre tipi di asserzioni (= espressioni booleane che diventano false solo in presenza di errori di programmazione)

- esplicitano le responsabilità dei moduli, evitando così ridondanza di controlli o controllo insufficienti
- risolvono i conflitti di responsabilità in presenza di errori

Eccezioni

Si deve sollevare un'eccezione se è violata una delle seguenti condizioni:

- precondizioni
- postcondizioni
- invariante

I moduli clienti potranno rilevare e gestire opportunamente tale eccezione o trasferirla a loro volta ai chiamanti

Precondizioni, postcondizioni ed ereditarietà

Le sottoclassi possono

- Aggiungere attributi e metodi, rispettando però il vincolo semantico
  $inv_{sottoclasse} \rightarrow inv_{classe}$
- Ridefinire i metodi, soddisfando però i seguenti vincoli semantici:
  $post_{sottoclasse} \rightarrow post_{classe}$
  $pre_{classe} \rightarrow pre_{sottoclasse}$

In questo modo le asserzioni impediscono che le operazioni ridefinite o aggiunte nella sottoclasse siano inconsistenti con quelle della classe padre

Uso ideale ed effettivo delle asserzioni

Ideale

- Le asserzioni dovrebbero essere incluse nel codice come parte della definizione di interfaccia
- I compilatori dovrebbero attivare il controllo delle asserzioni durante il debugging e rimuoverlo per generare le versioni definitive

Effettivo

- Le asserzioni vengono incluse nel codice per il debugging e il testing e vengono rimosse quando si compila il codice finale
- Il programma deve ignorare la presenza del controllo delle asserzioni
- L'uso delle sole precondizioni spesso consente di rilevare il massimo numero di errori, con il minimo spreco di risorse computazionali
- Solo Eiffel supporta le asserzioni come parte integrante del linguaggio

Componenti e connettori

Componenti:

- client
- server
- basi di dati
- filtri
- layer

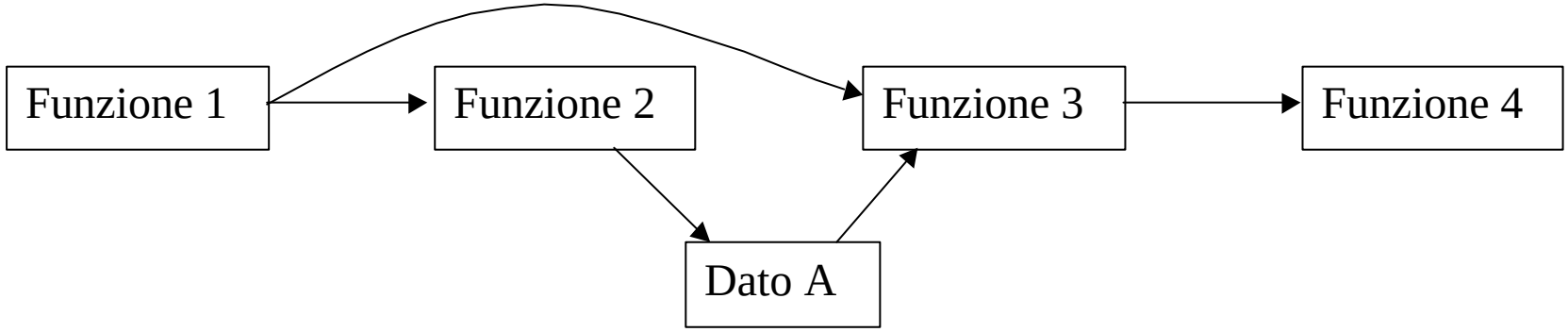
Connettori:

- chiamate di procedure
- diffusione di eventi
- protocolli delle basi di dati
- pipe(line)

Stile architetturale (Garlan & Shaw, 1996): organizzazione di componenti e connettori

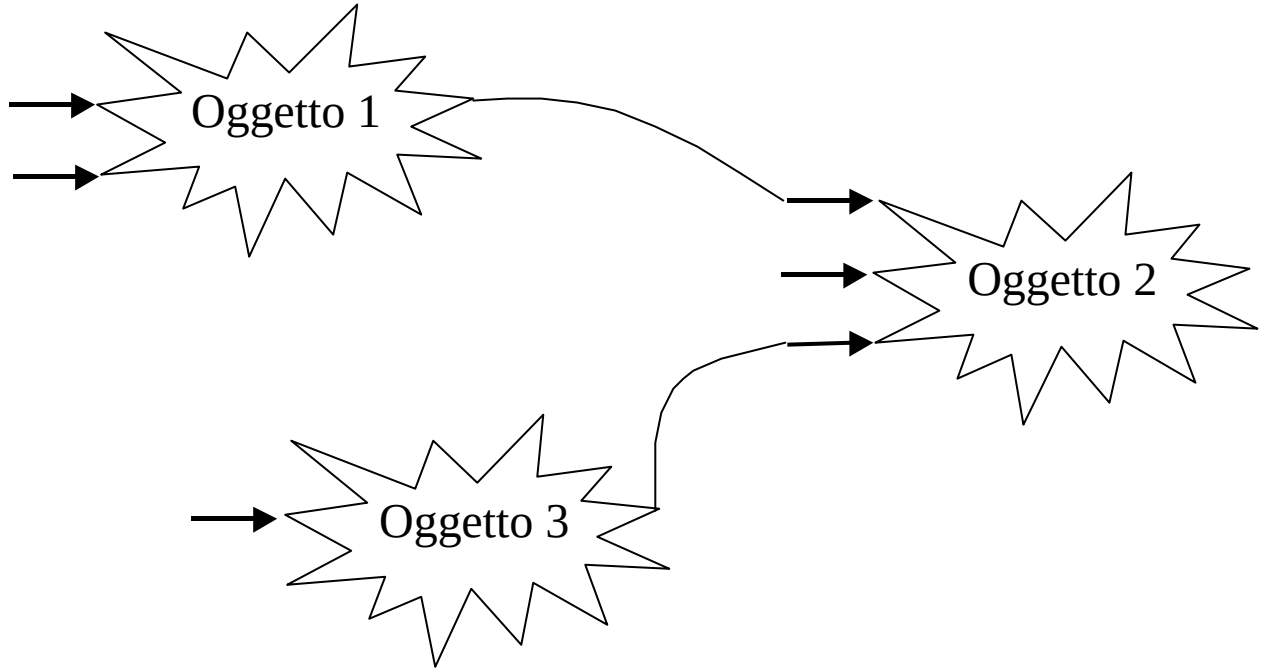
Architettura funzionale

- Il sistema è suddiviso in operazioni astratte
- Le operazioni si conoscono l'un l'altra
- L'interazione fra operazioni avviene mediante chiamate di procedure e dati condivisi



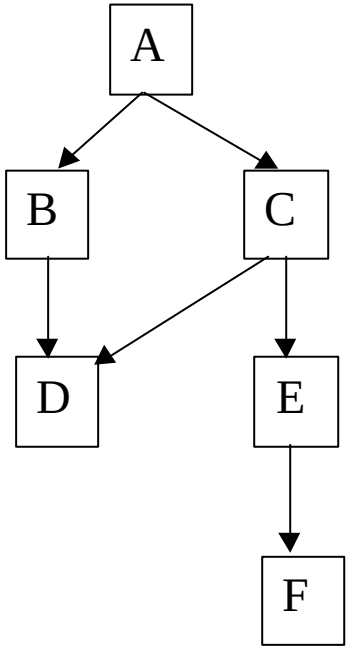
Architettura OO

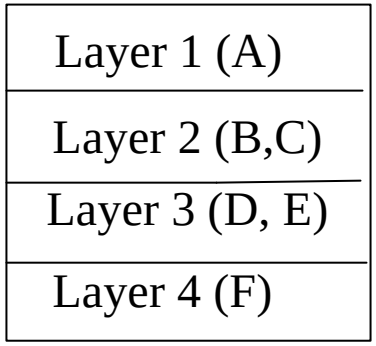
- Non c’è più separazione fra processo e dati
- Gli oggetti si conoscono l’un l’altro



Architettura a livelli

- Il sistema è organizzato in più livelli di astrazione delle funzionalità offerte, come una gerarchia di macchine astratte
- La gerarchia è data dalla relazione USA





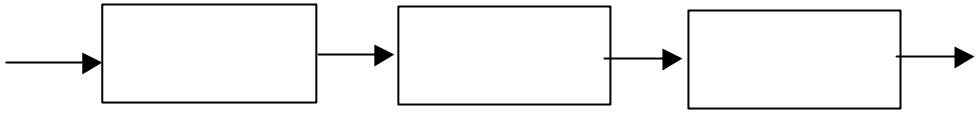
Architettura “pipe & filter”

È quella adottata da Unix. Ciascun filtro

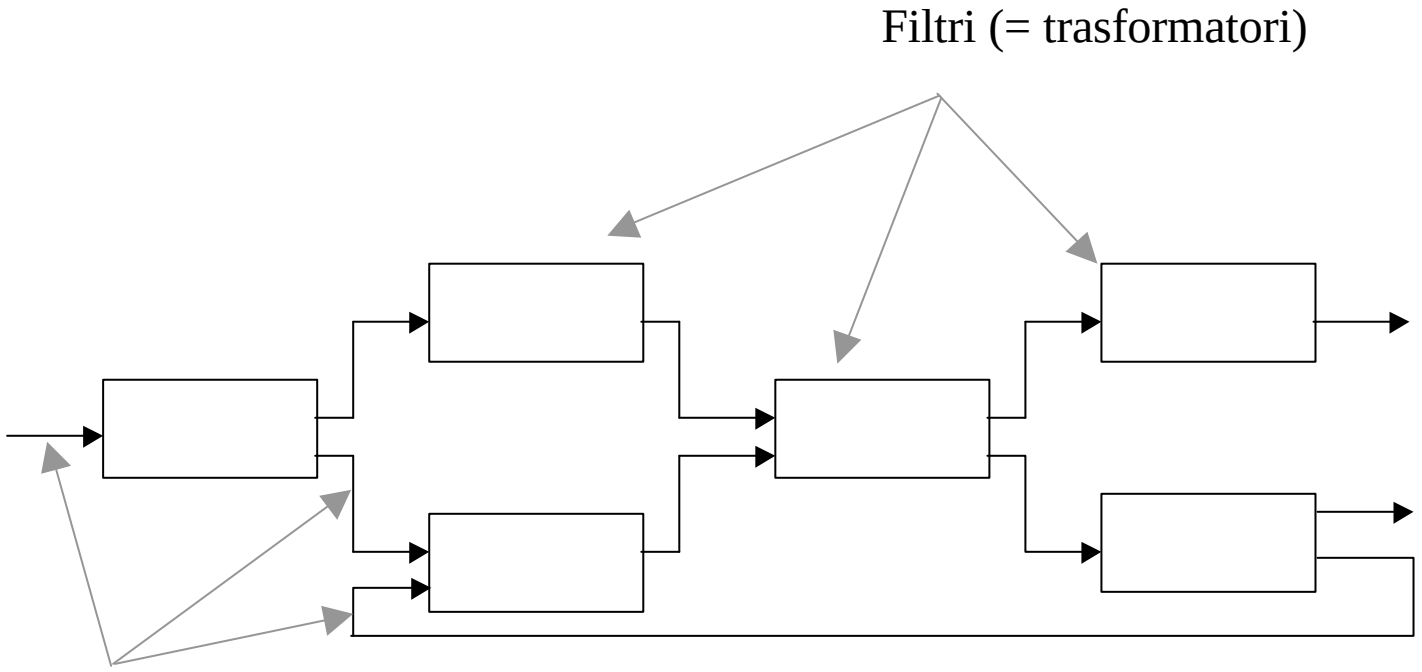
- acquisisce i dati dai suoi pipe di ingresso e fornisce i risultati sui suoi pipe di uscita
- ignora l’esistenza (e l’identità) di altri filtri

Architettura “pipe & filter” (cont.)

Pipe & filter sequenziale (= pipeline)



Pipe & filter concorrente

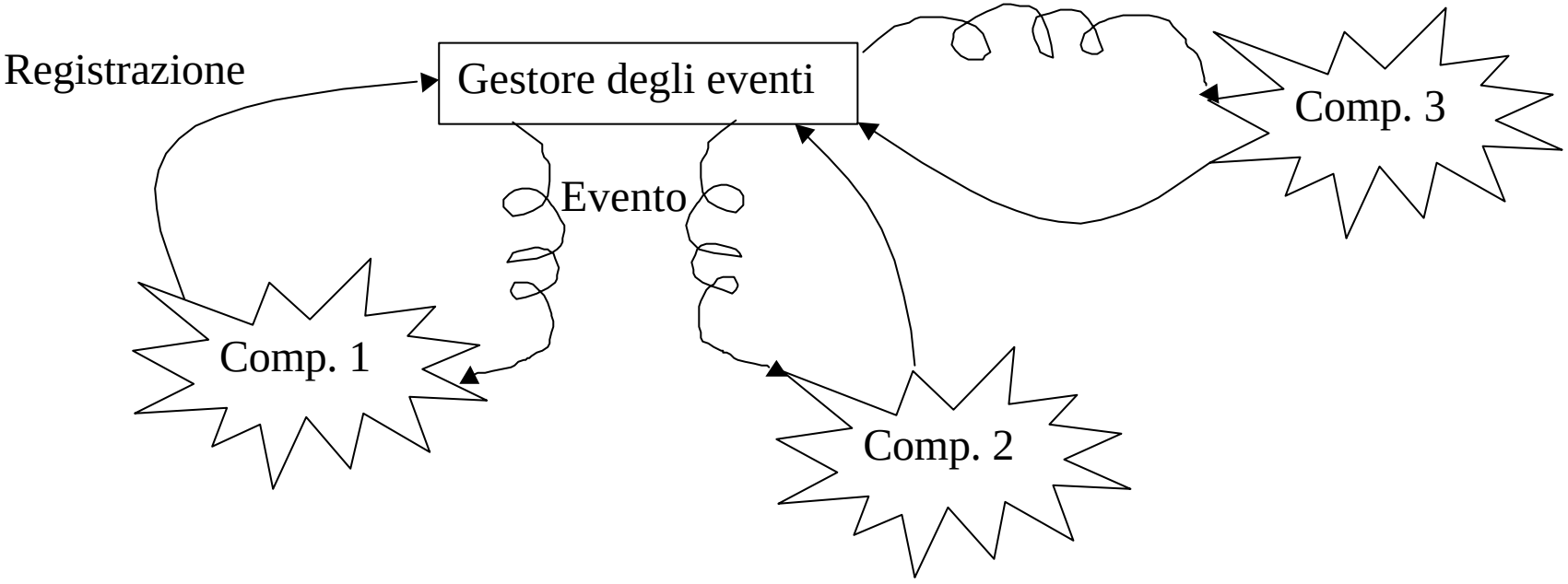


Architettura pipe & filter: pro e contro

+ Compositiva: il comportamento globale è la composizione di comportamenti individuali
+ Orientata al riuso dei filtri
+ Orientata al cambiamento: si possono facilmente aggiungere o sostituire filtri
– Nessuna persistenza (l'architettura non persiste immodificata ai cambiamenti dei requisiti)
– Repliche (per effettuare elaborazioni analoghe su dati diversi in punti diversi del flusso)
– Tendenza a una organizzazione batch

Architettura a invocazione implicita (event driven)

- Gli eventi sono trasmessi (diffusi) a tutti i componenti registrati
- Nessuna identificazione esplicita (mediante nome) dei componenti destinatari
- Deve esistere la possibilità di condividere le strutture dati



Sistemi event driven: pro e contro

+ Orientati al cambiamento: si possono facilmente aggiungere, togliere, sostituire componenti
– Problemi di “scalabilità”
– La dipendenza dal contesto (registrazione agli eventi) e dalla sequenza (ordinamento) degli eventi ne rende difficile la progettazione e il test

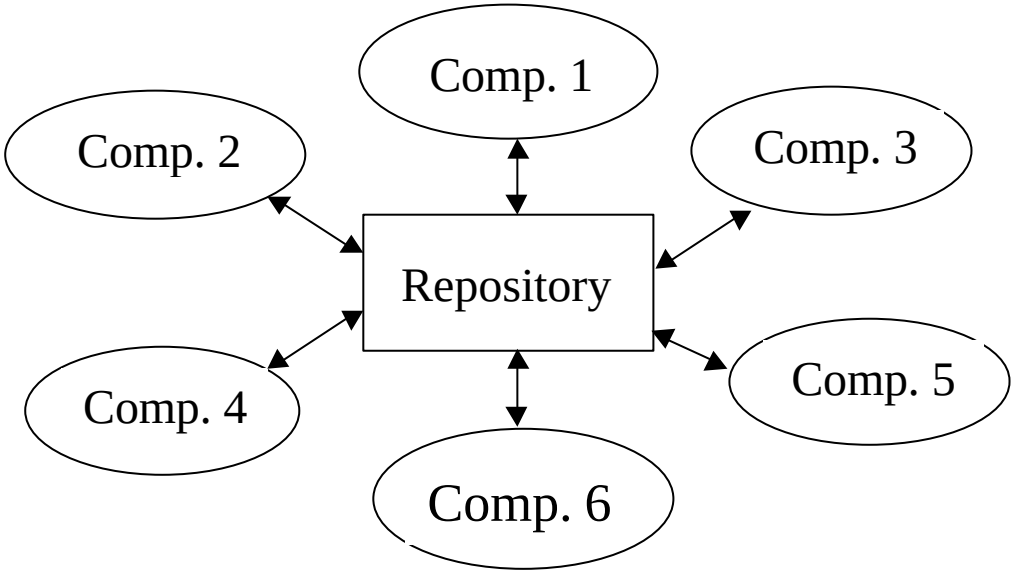
Un confronto

	Architettura funzionale	Architettura OO	Architettura pipe&filter	Architettura event driven
Cambiamenti algoritmici	-	-	+	+
Cambiamenti nelle strutture dati	-	+	-	-
Cambiamenti funzionali	+	-	+	+
Prestazioni	+	+	-	-
Riuso	-	+	+	±

Architettura a repository

I componenti comunicano solo attraverso il repository. Casi particolari:

- Basi di dati
- Architettura a lavagna



Architettura a repository (cont.)

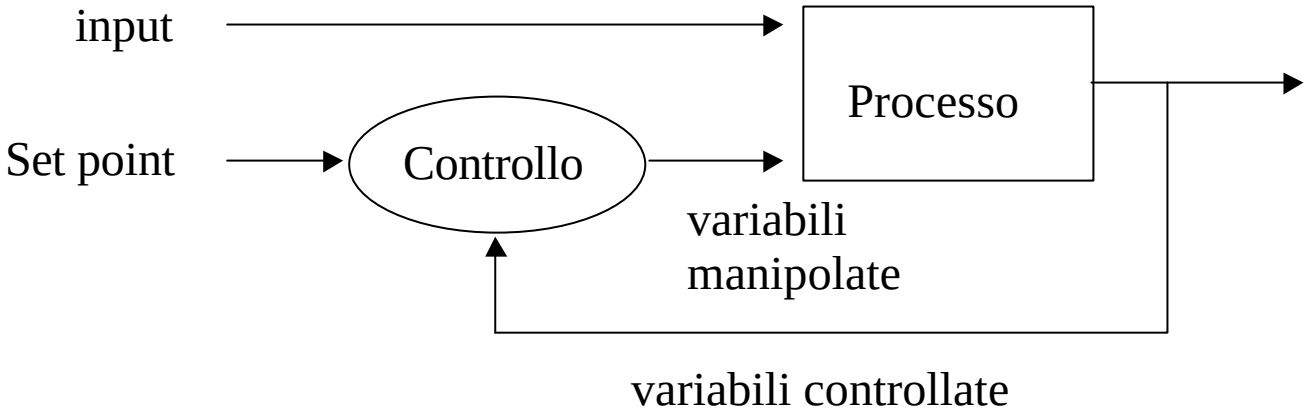
Basi di dati

- Componenti attivi
- Repository passivo
- Esiste un componente particolare (manipolatore delle transazioni) che legge le transazioni d'ingresso e invoca le funzioni appropriate

Architettura a lavagna

I cambiamenti di stato della lavagna innescano l'attivazione dei componenti (cioè, la lavagna è una base di dati attiva)

Architettura per il controllo di processo



Sw basato sui componenti

Visione secondo cui i sistemi sw possono essere costruiti a partire da componenti binari forniti da un produttore, detti COTS (Components Off The Shelf), che

- sono stati sviluppati e testati indipendentemente uno dall'altro
- possono essere assemblati rapidamente in diverse configurazioni "scalabili"
- sono dotati di interfacce esplicite

Es. Java beans, CORBA objects, ActiveX

Sw basato sui componenti: motivazioni e rischi

+ I produttori possono vendere a un mercato di massa (→ sorta di standardizzazione)
+ I produttori forniscono automaticamente gli upgrade
+ Lo sviluppo in loco viene ridotto
+ Gli stessi componenti possono essere riusati in molti sistemi
– I produttori continueranno a supportare i componenti?
– Minore flessibilità nella definizione dei requisiti
– Test di integrazione