INGEGNERIA DEL SOFTWARE

DESIGN

Avvertenza: gli appunti si basano sul corso di Ingegneria del Software tenuto dal prof. Picco della facoltà di Ingegneria del Politecnico di Milano (che ringrazio per aver acconsentito alla pubblicazione). Essendo stati integrati da me con appunti presi a lezione, il suddetto docente non ha alcuna responsabilità su eventuali errori, che vi sarei grato mi segnalaste in modo da poterli correggere.

e-mail: webmaster@morpheusweb.it
web: http://www.morpheusweb.it

IL GOAL

L'attività di design produce l'architettura software (il nostro design software)
L'architettura di un sistema software definisce il sistema in termini di componenti computazionali ed interazioni tra questi componenti

INTERAZIONE TRA COMPONENTI

Possono essere definite su diversi livelli di astrazione

Identifichiamo due livelli:

- **meccanismi**: quali sono i componenti e come interagiscono (più vicini all'implementazione)

  Ad esempio l'architettura client-server, non specifico il modo in cui realizzo il software, ho una prima separazione in componenti ed interazioni

- **stile**

  Posso parlare di architettura pensando a come funziona l'applicazione, dando uno schema generale (principalmente usando viste statiche)

MECCANISMI

- Quali sono i moduli?
- Qual è la loro interfaccia?
- Quali sono le relazioni utili tra moduli?

Metodo di distribuzione: Quali sono i criteri per decomporre i sistemi in moduli?

Documentazione: Come documentare il catalogo dei moduli e relazioni?

MODULI

Un modulo è una parte di un sistema che fornisce un set di servizi agli altri moduli.
I servizi sono elementi computazionali che ogni modulo può usare

INTERFACCIA

Definisce un legame tra il modulo ed i suoi utenti
Un modulo è formato dalla sua interfaccia ed il suo corpo (implementazione, segreti)
Gli utenti vedono il modulo soltanto attraverso la propria interfaccia (definisco ciò che gli utilizzatori conoscono e ciò che non devono conoscere).

RELAZIONI

- Utilizziamo le seguenti relazioni:
- USES: un modulo usa i servizi esportati da un altro
- IS_COMPONENT_OF: descrive le aggregazioni di moduli in moduli di livello superiore
- INHERITS: per sistemi orientati agli oggetti

MODELLO FORMALE

Possiamo modellare le relazioni in modo formale.

S = {M1, M2, . . ., Mn} set di moduli

Una relazione è un prodotto cartesiano $S \times S$

Se Mi e Mj Sono in S, la relazione <Mi, Mj> può essere scritta come: **Mi r Mj**

CHIUSURA TRANSITIVA

Mi r+ Mj

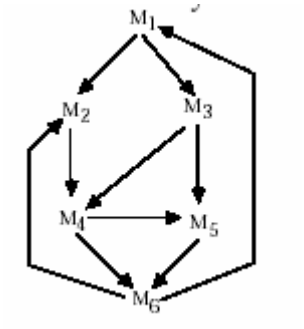
O sono in relazione direttamente oppure tramite altri moduli.

(Mi r Mj) or $\exists$ Mk in S : (Mi r Mk) and (Mk r Mj)

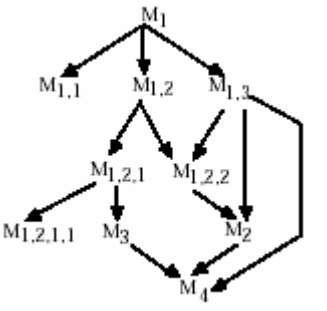
GERARCHIA

r è una gerarchia se e solo se: $\nexists$ Mi, Mj : (Mi r Mj) $\wedge$ (Mj r Mi)

La relazione d'uso è unidirezionale.



Non è una gerarchia



E’ una gerarchia.

Tipicamente una gerarchia è rappresentata da un DAG (grafo diretto aciclico)

USES

- A uses B
  A può accedere ai servizi esportati da B attraverso l’interfaccia
  Definita staticamente
  Spetta a B fornire i servizi

- A is a client of B

IS COMPONENT OF

Usato per descrivere un modulo di livello più alto come costituito da un certo numero di moduli di livello inferiore
E’ la composizione.
A is component of B: B consiste di diversi moduli, uno dei quali è A

INHERITS FROM

Se il sistema è sviluppato ad oggetti, l’ereditarietà consente ad un componente di estendere un altro.
Un erede può accedere ad alcuni dei segreti del suo antenato

STILE

I componenti sono delle cose tipo clients, servers, database, filtri e livelli in un sistema gerarchico.
Le interazioni tra i componenti possono essere semplici come una chiamata di procedura e l'accesso ad una variabile condivisa.
Ma possono essere complesse e ricche semanticamente come un protocollo client-server.

MECCANISMI VS STILE

Il meccanismo descrive come un'architettura è costruita

Lo stile è ciò che caratterizza un'architettura rispetto ad un'altra

> ***Esempio****: coupè vs station vagon*

Ci sono due viste dello stesso mondo, la distinzione può essere sfumata

DESIGN GOALS

Design for change

- anticipare possibili cambiamenti
- non concentrarsi sui bisogni di oggi, ma pensare alla possibile evoluzione

Program family

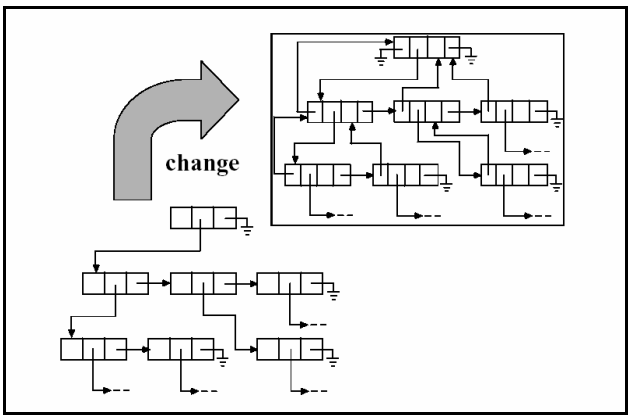
- pensare un programma come facente parte di una famiglia

POSSIBILI CAMBIAMENTI

- **Cambiamenti in algoritmi**
  (es. da bubblesort a quicksort)
- **Cambiamenti nelle strutture di dati**
  (circa il 17% del costo di manutenzione)
- **Cambiamenti nella macchina astratta sottostante**
  (periferiche hardware, OS, DBMS… nuove release, problemi di portabilità)
- **Cambiamenti nell'ambiente**
  (ad esempio introduzione dell'euro)
- **Cambiamenti dovuti a strategie di sviluppo**
  (prototipi che si evolvono)

Esempio:

Ogni nodo dell'albero punta al figlio ed al fratello.



Ogni nodo dell'albero punta al figlio ed al fratello
→
Cambiamento: voglio che il figlio punti al padre

FAMIGLIA DI PROGRAMMI

Pensare ad un programma ed a tutte le sue varianti come membri di una famiglia

Il goal è progettare l'intera famiglia, non del singolo membro separatamente.

ESEMPIO

Sistema di prenotazione

- Per alberghi (prenotazione stanze, ristorante, sala conferenze, proiettori…)
- Per un'università (molte funzionalità sono simili, altre differenti)

Lo scopo è di progettare l'intera famiglia, non il singolo membro e differenziarli più tardi possibile

PRINCIPI DI DESIGN

Come selezionare i moduli?

Come definire le interfacce dei moduli?

Come definire le relazioni USE?

COME SELEZIONARE I MODULI

- Un modulo è un'unità autosufficiente
- Interconnessioni di tipo USE devono essere minimizzate

PRINCIPIO: massimizzare l'accoppiamento. Se tra due moduli ci sono troppe relazioni d'uso diverse, vuol dire che i due moduli vanno accoppiati.

COME DISTINGUERE TRA MODULI ED INTERFACCE

- Distinguere tra ciò che il modulo fa e ciò che offre agli altri (i suoi segreti).
- Minimizzare il flusso di informazioni ai client per massimizzare la modificabilità
- Le interfacce dei moduli devono essere minime e stabili.

PRINCIPIO: Information Hiding. I moduli servono a nascondere l'informazione.
Voglio nascondere ciò che è passibile di cambiamento.

PRINCIPI E CONCETTI CHIAVE DELLA PROGETTAZIONE

- Decomposizione
- Astrazione
- Information Hiding
- Modularità
- Estendibilità
- Macchina virtuale
- Gerarchie
- Famiglie di programmi e sottoinsiemi

La meta principale di questi concetti è:

- Gestire la complessità dei sistemi software
- Migliorare i fattori di qualità del software
- Facilitare il riuso sistematico

COMPONENTI

- Che realizzano **operazioni astratte** (ad esempio i sort, i merge...)
  E' datata e non si usa più; porta in maniera prematura a progettare i passi sequenziali

- **Oggetti astratti**: sono moduli che rappresentano la struttura dati ed esportano un set di operazioni per cambiare lo stato degli oggetti (istanziazioni delle classi in C++)

- **Tipi di dati astratti**: componente che definisce un tipo e non ha oggetto. Dal tipo possiamo generare tanti oggetti

COME DEFINIRE USES TRA MODULI

Bisogna fare in modo che la relazione USES definisca una gerarchia (E 'più semplice da comprendere e da verificare)

E' sconsigliata la ricorsione mutua tra moduli.

Se la relazione di USE non è gerarchica, è più difficile testare i singoli moduli.

La gerarchia definisce un sistema attraverso "livelli di astrazione"

PROGETTAZIONE ORIENTATA AD OGGETTI

Il modulo è di per se una risorsa; è usata da altri per generare istanze

Introduce la relazione di ereditarietà

DESING BY CONTRACT

Raffina un principio di design adeguato alla progettazione orientata agli oggetti

Un CONTRATTO è accordo tra un cliente ed un contraente

CONTRATTO PER UN MODULO AD OGGETTI

Definisce precondizione e postcondizione

PRECONDIZIONE: cosa è richiesto da ciascun metodo (vincolo per il cliente)

POSTCONDIZIONE: cosa ogni metodo fornisce (vincolo per il contraente)

Possono essere espresse entrambe usando la logica.

Esempio:

Operation insert(element) in a table

Precondizioni: n°elementi<dimensione
Postcondizzioni: n°elementi'=n°elementi + 1

PERCHE' LE PRECONDIZIONI?

Può una routine essere preparata per gestire tutti i possibili input? La risposta è NO.

- Precondizioni WEAK: true significa che non ci sono vincoli
- Precondizioni STRONG: false significa che non può essere invocata

La scelta delle precondizioni è una decisione di progetto; non c'è una regola assoluta. E' preferibile scrivere routine semplici che soddisfino un contratto ben definito piuttosto che routine che cercano di soddisfare ogni possibile soluzione

Comunque non sostituiscono il controllo sull'input o la gestione delle eccezioni.

PRECONDIZIONI E POSTCONDIZIONI

PRECODIZIONI

Il client deve garantire la proprietà

if *p è la precondizione per un metodo m*
    write *(for ogni oggetto x)*
        if *(x.p) x.m(...)*
        else *...trattamento speciale*

POSTCONDIZIONI

Il contraente deve garantirle nell'implementazione del metodo.

IN JAVA

Vengono chiamate asserzioni.

```
m( ){
  …
  assert x<20;
  …
}
```

Prima di procedere, l'interprete verifica che sia x<20.
Se è vera, procede, altrimenti genera un'eccezione.

PROPRIETA' INTERNE DI UNA CLASSE

Possiamo specificare una proprietà che tutte le istanze devono soddisfare come ***invariante***

L'invariante è vero dopo la creazione, e prima e dopo ogni operazione

Esempio: $0 \leq n°elementi \leq dimensione$

L'invariante definisce un obbligo in più; l'implementazione della classe deve soddisfarlo.

CORRETTEZZA DELLE CLASSI

OPERAZIONE DI CREAZIONE: $\{pre_c\}$ constructor {INV'}

ALTRE OPERAZIONI: $\{pre_{op} \wedge INV\}$ constructor $\{ post_{op} \wedge INV' \}$

IL RUOLO DELLE ECCEZIONI

Dovrebbero essere sollevate se una delle seguenti condizioni è violata

- precondizioni
- postcondizioni
- invarianti

Quando il controllo lascia una routine, sia la precondizione che l'invariante sono veri oppure viene sollevata un'eccezione (le eccezioni possono essere elencate nell'interfaccia)

EREDITARIETA'

Le sottoclassi possono aggiungere attibuti e metodi

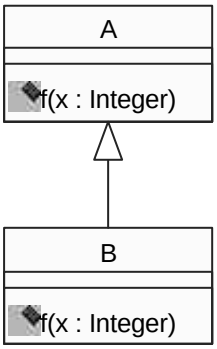
Possono ridefinire metodi

vincoli sintattici (covarianza del risultato e controvrianza dei parametri)

Vicoli semantici

$\mathbf{pre}_{\mathbf{class}} \rightarrow \mathbf{pre}_{\mathbf{subclass}}$
$\mathbf{post}_{\mathbf{class}} \rightarrow \mathbf{post}_{\mathbf{subclass}}$

Esempio:



B ridefinisce f()

Posso scrivere:

```
A a = new B();
a.f(x);
```

seleziona la f del tipo dinamico (che è B)

Consideriamo le seguenti precondizioni:

A→ x>0 B→x>5

Non funziona perché se ho x=2, se ragiono in termini del tipo statico posso accettarlo, se mi riferisco al tipo dinamico no, infatti per il tipo dinamico deve essere x>5.

DESIGN STYLES

Sono un modo comune di astrarre per uno specifico prodotto e tirar fuori gli aspetti essenziali.

COMPONENTI E CONNETTORI

COMPONENTI

- clienti
- server
- filtri
- livelli
- database
- ...

CONNETTORI (come il client è connesso al server)

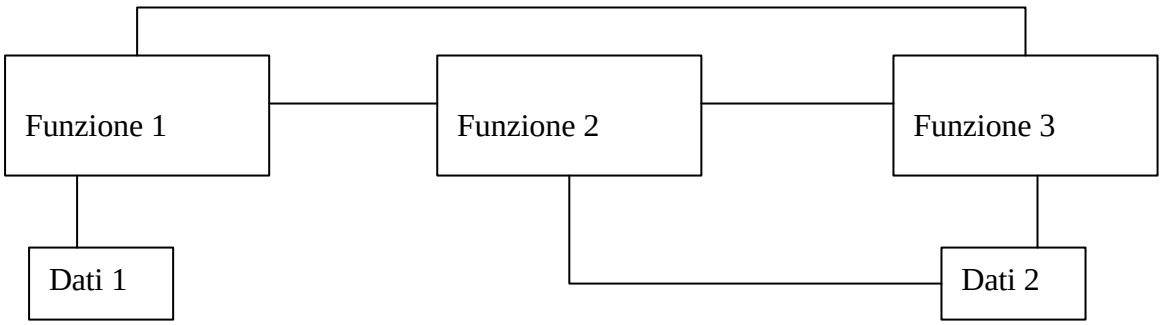
- chiamate di procedura
- eventi broadcast
- protocolli di database
- pipes
- ...

ARCHITETTURA FUNZIONALE

I componenti realizzano astrazioni funzionali

I connettori sono *chiamate di funzione* ed i *dati di ritorno*

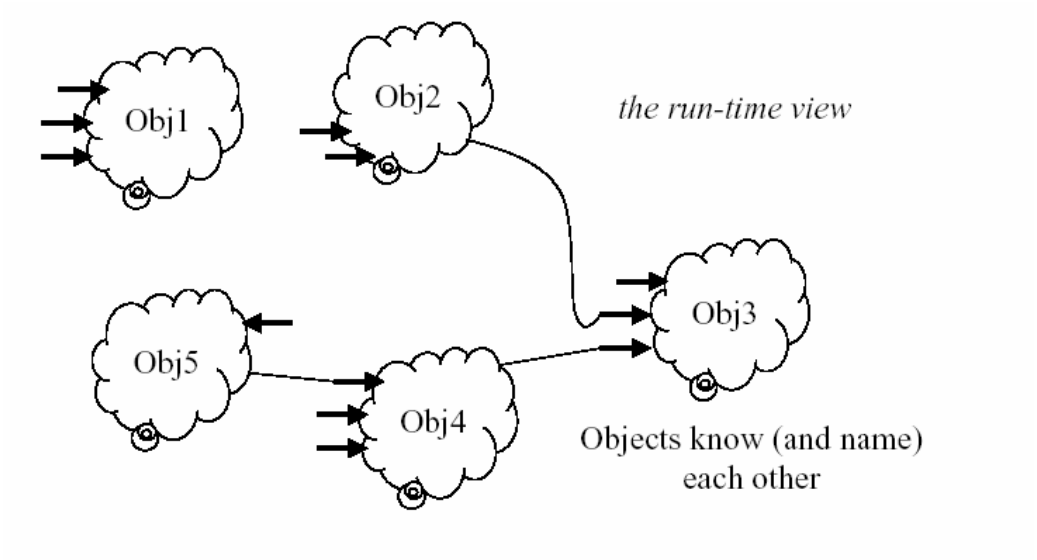
Esiste un altro meccanismo di connessione mediante i *dati condivisi*



I componenti devono conoscere i nomi l'uno dell'altro, poiché un componente invoca i servizi di un altro.

ARCHITETTURA AD OGGETTI

A run-time il sistema è composto da oggetti. I componenti non sono funzioni, ma hanno al loro interno lo stato dell'applicazione. Ogni componente è combinazione di stato e servizi esportati. Gli oggetti hanno un interfaccia e comunicano mediante scambio di messaggi

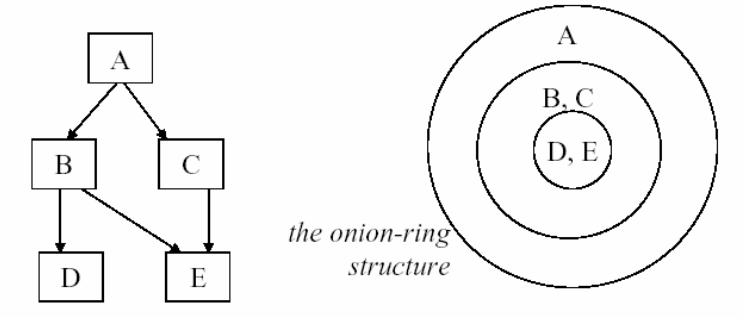


ARCHITETTURA A LIVELLI

I componenti sono organizzati in una gerarchia di astrazione

I livelli sono rispetto ad una relazione

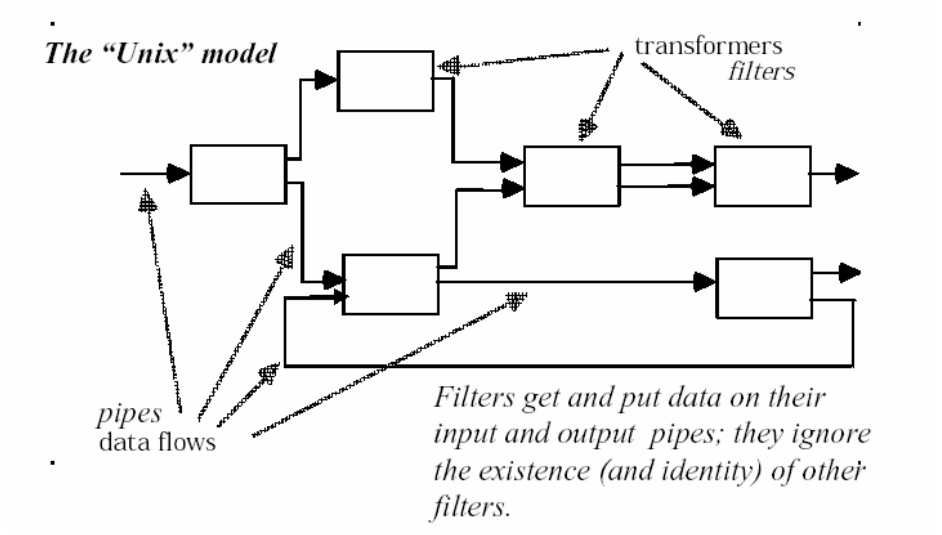
Esempio:
A use B, C...
B use D, E...



Ogni livello realizza dei servizi per il livello superiore

ARCHITETTURA PIPES & FILTER

Ereditata da Unix.
E' simile alla funzionale.



Si realizzano i componenti mediate PIPE (condotti)

I componenti sono visti come filtri

Quando progetto un componente non so a priori come sono collegati.

Attraverso il meccanismo del pipe stabilisco come sono interconnessi

I dati sono un flusso continuo, l'idea è di avere dei componenti piccoli, specializzati e con un'interfaccia semplice

VANTAGGI

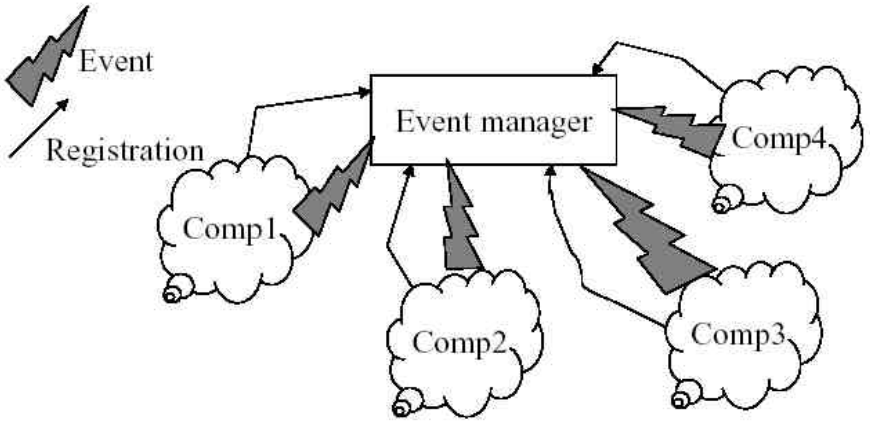
- composizionale: il comportamento complessivo è composizione di comportamenti individuali
- orientata al riuso
- Modifiche sono semplici

SVANTAGGI

- *non c'è persistenza* (i dati fluiscono in un condotto e non c'è un deposito dati)
- replicazioni (se ho bisogno della stessa funzioni in punti diversi, devo replicarla)
- tendenza all'organizzazione batch

SISTEMI BASATI SUGLI EVENTI

C'è un event manager che fa da connettore. I componenti non si conoscono tra di loro.



I componenti possono registrarsi ed informare il gestore degli eventi quali sono gli eventi di interesse. Possono inoltre notificare al gestore di aver osservato un evento, in questo caso il gestore degli eventi fa una notifica ai componenti che si erano registrati.

Non c'è nessun nome esplicito associato ai componenti (particolarmente importante per componenti mobili) poiché ciascun componente conosce solo il gestore degli eventi.

VANTAGGI

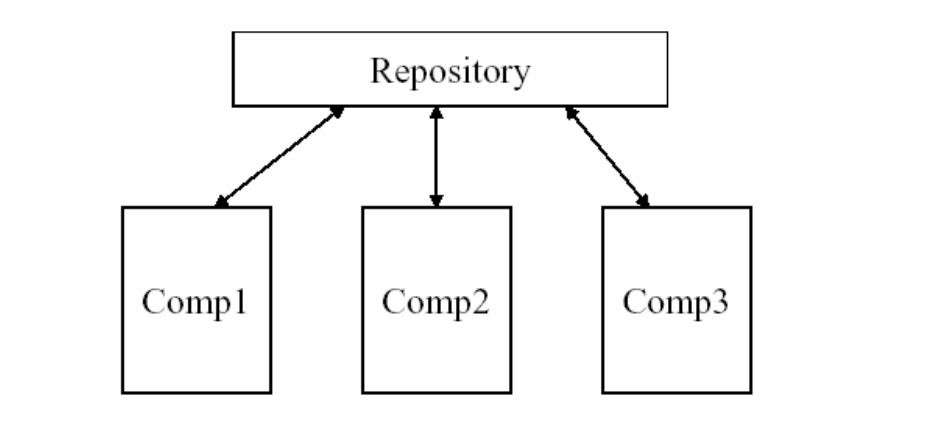
- sempre più usata per la moderna integrazione
- facile inserimento/cancellazione di eventi

SVANTAGGI

- potenziali problemi di scalabilità (si tende a risolverlo usando un'architettura distribuita per l'event manager)
- ordine degli eventi (l'ordine di esecuzione non è sempre quello di arrivo)

ARCHITETTURA BASATA SU MAGAZZINO

I sistemi basati su eventi non hanno nessun tipo di persistenza, lo stato è fuori.
I componenti comunicano attraverso un magazzino di dati.



Esempio: il caso del database

I componenti sono attivi, il database è passivo
Un ulteriore coponente (transaction handler o gestore delle transazioni) legge le transazioni in ingresso e chiama le funzioni appropriate

BLACKBOARD

Esiste una variante detta BLACKBOARD (lavagna)

I componenti leggono e scrivono sulla lavagna.

La lavagna è attiva, lo stato della lavagna fa da trigger e risveglia determinati componenti