

DESIGN DELLE COMPONENTI **(Low level design):** **FASI E PRINCIPI**

Ingegneria del Software a.a. 2023-24

AGENDA

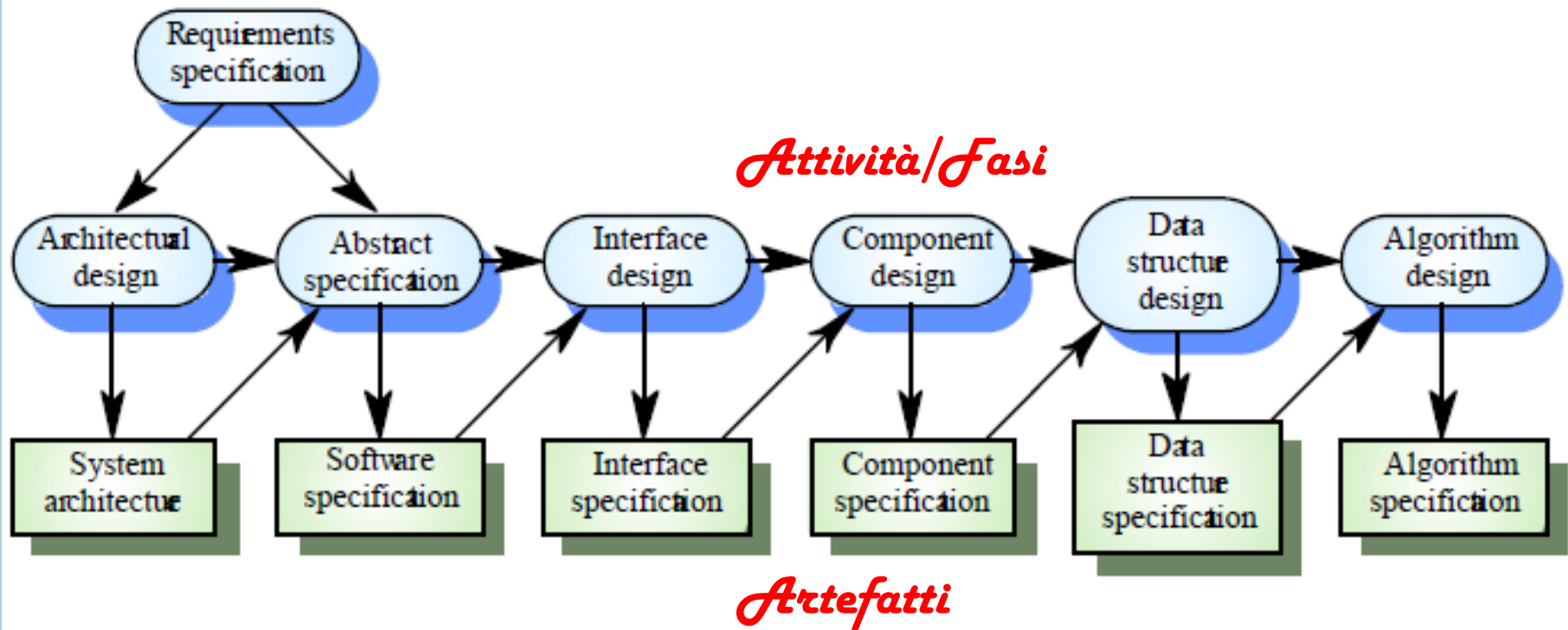
- Fasi del processo di design
- **Design by contract**: solo nozione
- Progettazione degli algoritmi
 - **PDL** o pseudocodice
- Principi di (buona) progettazione
 - Astrazione
 - Decomposizione
 - Modularità
 - Information Hiding
 -



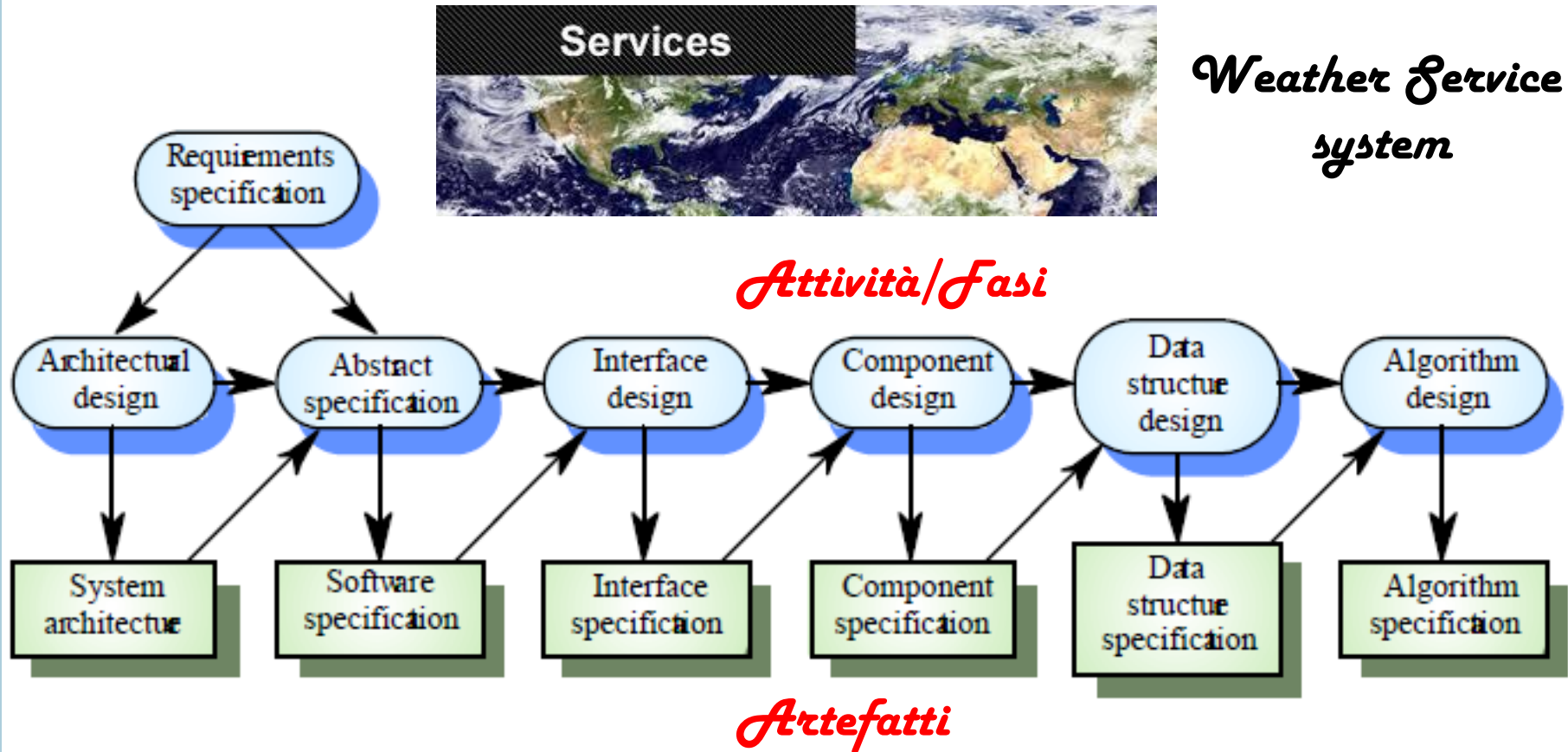
Bertrand Meyer



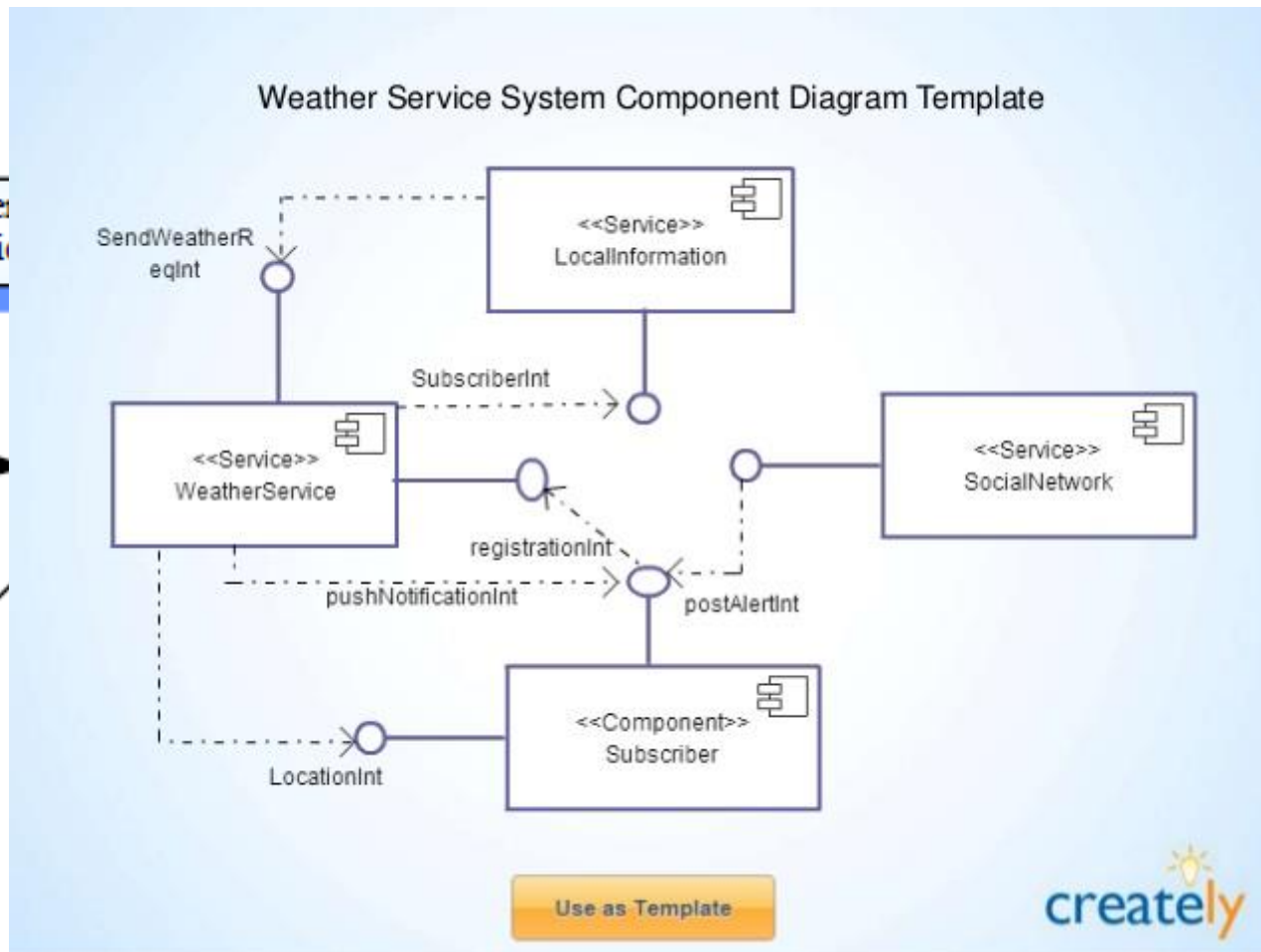
FASI DEL PROCESSO DI DESIGN



FASI DEL PROCESSO DI DESIGN



FASI DEL PROCESSO DI DESIGN



Requirements
specification

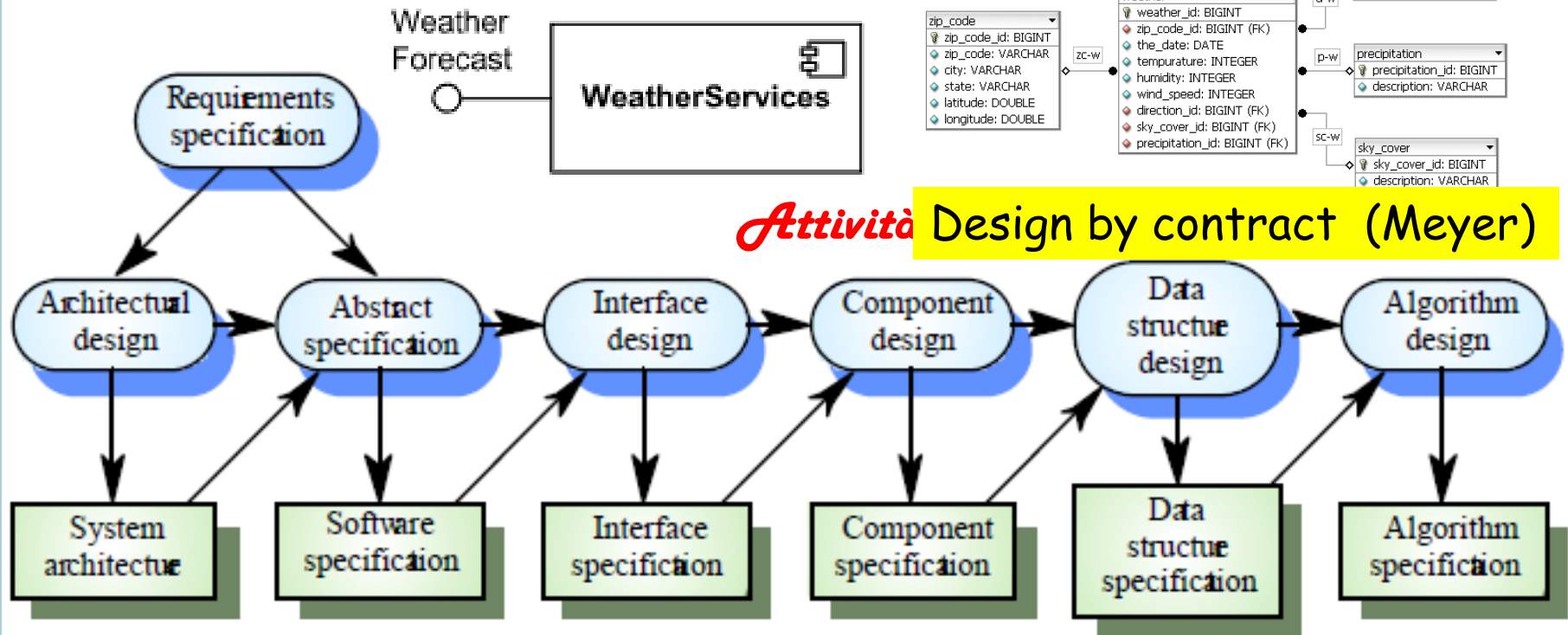
Architectural
design

System
architecture

Algorithm
design

Algorithm
specification

FASI DEL PROCESSO DI DESIGN



Weather Forecast interface:

+GetHistoricalTemperature(date:Date, hour:Hour)
+GetTomorrowForecastRain(unitOfMeasure:Unit)

Design by contract (Meyer)

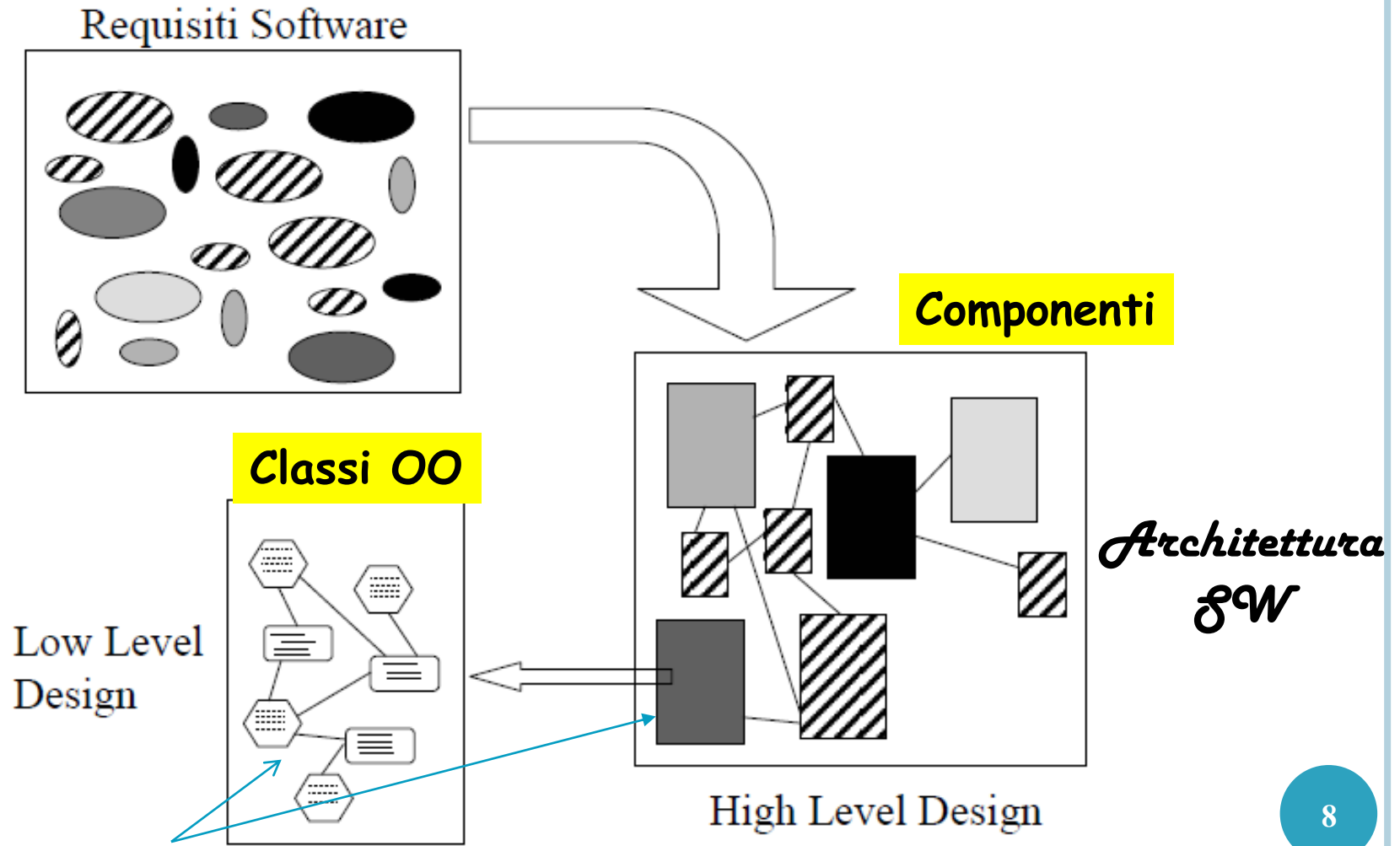
List:

ArrayList, Vector o LinkedList?
(nel caso di design platform dependent)

Sort:

QuickSort o Bubble Sort?

HIGH LEVEL DESIGN E LOW LEVEL DESIGN

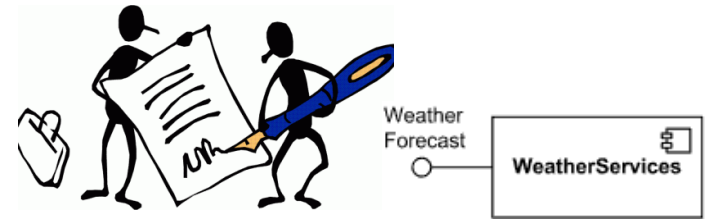




DESIGN BY CONTRACT

DESIGN BY CONTRACT

- Metodo di design per il software che ha come obiettivo quello di migliorarne la qualità
- Prescrive che il progettista debba definire **specifiche precise delle interfacce dei classi/componenti software**, basandosi sulla metafora di un **contratto** legale



- L'idea centrale è che una componente software ha degli obblighi nei confronti delle altre componenti
- Un “contratto”, viene creato per ogni componente del sistema **prima che sia codificato**

ELEMENTI DI UN CONTRATTO (Classi OO)

Pre-condizione.
rappresentante **le a**
prima che venga ese

- È un errore invocare l'operazione radice quadrata su un numero negativo e le conseguenze di tale azione sono indefinite: il risultato è non predicibile (errato, eccezione, ...)

Es. radice quadrata: `int sqrt(int x)`
pre: $x \geq 0$

Post-condizione. Espressione a valori booleani riguardante lo '**stato del mondo**' dopo l'esecuzione di un'operazione

Es. radice quadrata: `int sqrt(int x)`
pre: $x \geq 0$
post: $x = \text{ris} * \text{ris}$

Invariante di classe. Condizione che ogni oggetto della classe deve soddisfare quando è '**in equilibrio**'

- In equilibrio: non 'in mezzo' ad una transizione (esecuzione operazione); in ogni momento in cui è possibile eseguire un'operazione

PRE E POST CONDIZIONI: ESEMPIO CONTO CORRENTE

```
class Account {  
    private int balance;  
    private List<Integer> deposits;  
    private List<Integer> withdraws;  
  
    Account(int initialAmount) {}  
  
    public void deposit(int value) {}  
  
    public void withdraw(int value) {}  
  
    public boolean mayWithdraw(int value) {}  
  
    public int getBalance() {}  
  
}
```

saldo

liste di depositi e prelievi
Semplificazione!!!!

Precondition:

value >= 0

value <= balance

Postcondition:

balance = balance@pre-value

Object Constraint Language (OCL)

INVARIANTE:

ESEMPIO CONTO CORRENTE

```
class Account {  
    private int balance;  
    private List<Integer> deposits;  
    private List<Integer> withdraws;  
  
    Account(int initialAmount) {}  
  
    public void deposit(int value) {}  
  
    public void withdraw(int value) {}  
  
    public boolean mayWithdraw(int value) {}  
  
    public int getBalance() {}  
  
}
```

saldo

liste di depositi e prelievi
Semplificazione!!!!

Balance = somma di tutti i depositi – somma di tutti i prelievi

Invariant (Class):

balance >= 0 and

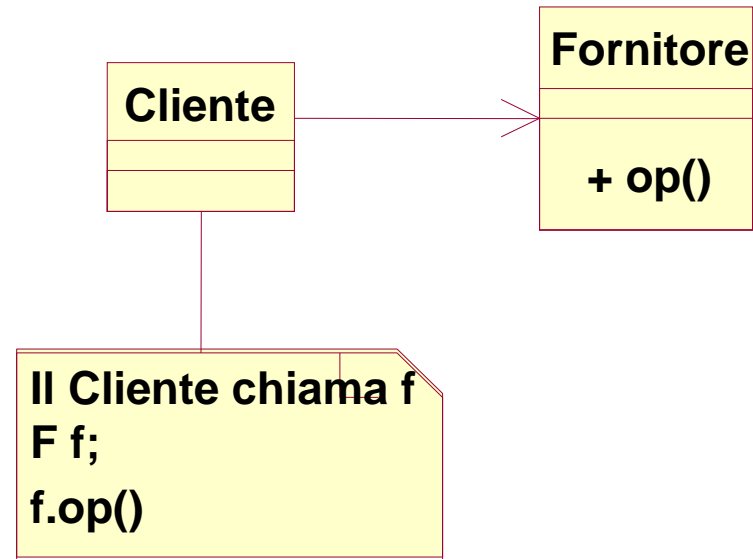
balance = (deposits.value --> sum() - withdraws.value --> sum())

LE PRECONDIZIONI SONO UTILI???

- Radice quadrata: `int sqrt(int x)`
pre: $x \geq 0$
- A prima vista **aggiungere una pre-condizione** potrebbe sembrare **inutile** in quanto è sempre possibile mettere dei controlli che assicurino che `sqrt` sia sempre chiamata in modo appropriato ...
- **Il problema è:** chi è responsabile di questi controlli?
- Senza una dichiarazione esplicita di responsabilità potremmo avere:
 1. **Troppi pochi controlli (ognuno delega l'altro)**
 2. **Troppi controlli (entrambe le parti li eseguono)**
 - 'Defensive' programming
 - Non va bene perchè si duplica il codice

DESIGN BY CONTRACT

- Il fornitore **garantisce** che se *pre* & *inv* valgono allora, dopo l'esecuzione di **op**, vale *post* (& *inv*)
- Il fornitore **chiede** che valgano *pre* & *inv* (altrimenti non garantisce nulla!)
- Pre = **Richiesta** che il fornitore fa al chiamante/cliente
- Post = **Assicurazione, garanzia** che il fornitore dà



La responsabilità dei controlli è esplicitata!

DI CHI È LA RESPONSABILITÀ DI UN FAILURE IN UN SISTEMA SOFTWARE?

○ Le responsabilità sono chiare:

- **Prima** della chiamata di un operazione è responsabilità del **Cliente**
 - E' responsabilità di chi chiama $\text{sqrt}(X)$ verificare che sia $X > 0$
- **Durante** l'esecuzione dell'operazione è responsabilità del **Fornitore**

Se accade un fallimento e:

Pre non valida \Rightarrow errore del chiamante!

Post o inv non valida \Rightarrow errore del fornitore!
(se Pre era valida)

VANTAGGI DEL DESIGN BY CONTRACT

- **Codifica.** È una guida per lo sviluppatore durante la fase di codifica
 - Sviluppatore deve seguire le specifiche!
- **Migliorano la qualità del software.** Definisce quale componente **è responsabile** ad effettuare i controlli. Aiuta a scrivere operazioni semplici che soddisfino un contratto ben definito
 - e non delle “op” che cercano di gestire tutti i casi possibili
- **Documentazione.** Pre, Post e Invarianti documentano in modo preciso cosa fa una componente/classe
- **Testing.** Guida alla generazione di casi di test “black-box”
- **Debugging.** Se è implementato nel codice ci permette di trovare “il colpevole” di un malfunzionamento
 - Le eccezioni si sollevano quando il contratto è violato
 - **Esistono framework** per diversi linguaggi



PROGETTAZIONE DEGLI ALGORITMI

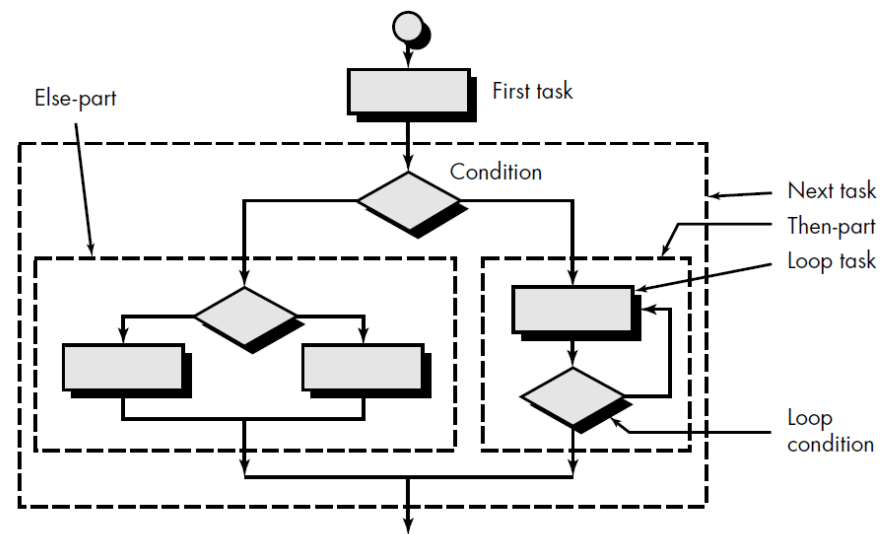
DESIGN DEGLI ALGORITMI



- È l'attività più vicina alla codifica
 - Spesso viene lasciata in parte o totalmente agli sviluppatori
- Di solito si seguono i **seguenti passi**
 1. Si analizza la descrizione di design della **classe “target”**
 2. Se esistono una o più operazioni che necessitano di un algoritmo
 3. Se è possibile **selezionare** un algoritmo si seleziona
 - Es. Sort → ‘Quick sort’
 4. Se invece occorre **definire** un algoritmo si sceglie una **notazione**
 5. Utilizzando la nozione di “**stepwise refinement**” si sviluppa l'algoritmo nella notazione scelta
 6. Si usano i **metodi formali** per provare la correttezza dell'algoritmo proposto
 - Triple di Hoare (correttezza alla Hoare)
 - Solo quando è il caso: es. sistemi safety-critical

NOTAZIONI

- Esistono diverse notazioni utilizzate per rappresentare un algoritmo
- Si dividono in visuali e testuali
 - **Visuali:** Activity diagram di UML, Diagrammi di flusso (flowchart), Box diagram, Structured Chart, Decision table, ...
 - **Testuali:** Program Design Language (PDL) o Pseudo-codice



VI RICORDATE??

```

type
  SET = ↑ nodetype;

```

Now we can specify fully the function MEMBER, in Fig. 5.2. Notice that since SET and “pointer to nodetype” are synonymous, MEMBER can call itself on subtrees as if those subtrees represented sets. In effect, the set can be subdivided into the subset of members less than x and the subset of members greater than x .

```

function MEMBER ( x: elementtype; A: SET ) : boolean;
  { returns true if x is in A , false otherwise }
begin
  if A = nil then
    return (false) { x is never in  $\emptyset$  }
  else if x = A ↑.element then
    return (true)
  else if x < A ↑.element then
    return (MEMBER(x, A ↑.leftchild))
  else { x > A ↑.element }
    return (MEMBER(x, A ↑.rightchild))
end; { MEMBER }

```

Fig. 5.2. Testing membership in a binary search tree.

PDL o PSEUDOCODICE (VEDERE LIBRO PRESSMAN)

- Il Program design language (PDL) o pseudo-codice è un **linguaggio semplificato** che usa il vocabolario di un linguaggio (es. Inglese) e la sintassi di un altro (es. linguaggio di programmazione come Java)

La narrativa permette vari livelli di astrazione ed abilita il **stepwise refinement**

- La differenza tra PDL e un linguaggio reale di programmazione è nell'utilizzo della “**narrativa**” direttamente dentro i comandi PDL

ESEMPIO: STEPWISE REFINEMENT

$$6, \textcolor{red}{2}, \textcolor{red}{9} = 7$$

Dati tre numeri interi calcolare la **differenza** tra il più **grande** e il più **piccolo**

1. “Acquisire dall'esterno valori A, B, C”
2. “Calcolare il più grande di A,B,C”
3. “Calcolare il più piccolo di A,B,C”
4. “Calcolare la differenza tra i risultati ottenuti in 2. e in 3.”
5. Rendere disponibile all'esterno il risultato”



```
Var A, B, C, BIG, SMALL, RESULT: Integer
1. “Acquisire dall'esterno valori A, B, C”
2. BIG <- CALCOLA_MASSIMO(A, B, C);
3. SMALL <- CALCOLA_MINIMO(A, B, C);
4. RESULT <- BIG – SMALL;
5. “Rendere disponibili all'esterno RESULT”
```

CALCOLA_MASSIMO(A, B, C)

Var BIG: Integer

1. “Se A è più grande di B allora metti in BIG il valore di A altrimenti quello di B”
2. “Se C è più grande di BIG allora metti in BIG il valore di C”
3. “restituisce BIG”



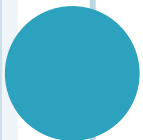
CALCOLA_MASSIMO(A, B, C)

Var BIG: Integer

```
If A>B then
    BIG <- A;
else
    BIG <- B;
endif
If C>BIG then
    BIG <- C;
endif
return BIG;
```

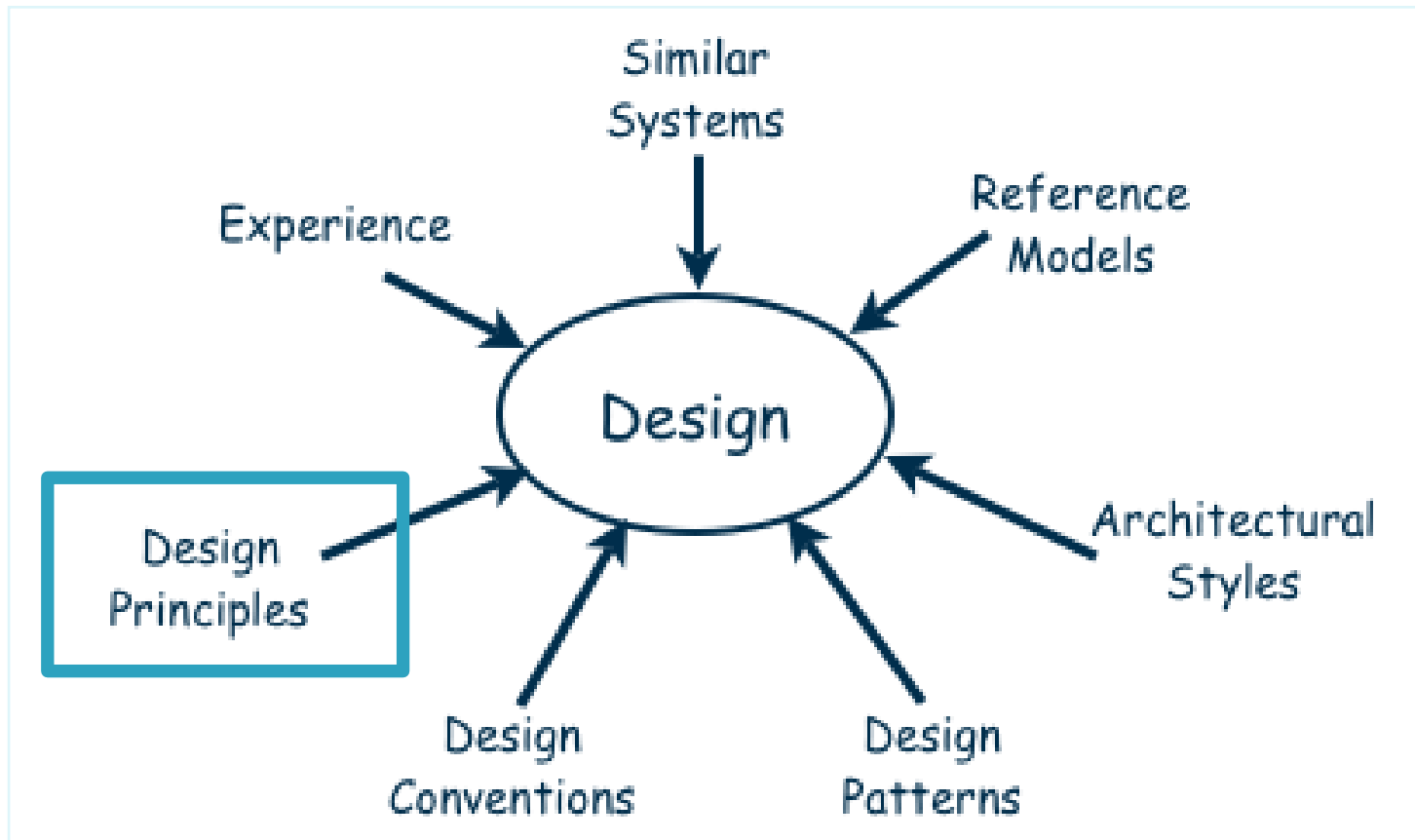


Passo di raffinamento



PRINCIPI DI PROGETTAZIONE

DESIGN: UN PROCESSO CREATIVO?



PRINCIPI DI (BUONA) PROGETTAZIONE

I principi di progettazione guidano verso il raggiungimento degli obiettivi di **qualità** per il progetto

- Seguire i principi:

- Astrazione
- Decomposizione
- Modularità
- Information Hiding
-

*Non possiamo vederli tutti,
Davis (1995) ne presenta 201 ...*

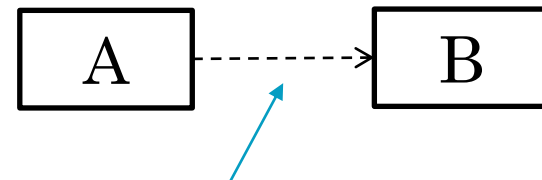
- porta a produrre software: **manutenibile**,
comprensibile, **semplice** da **testare**,
riusabile, **riparabile** e **portabile** ...

Si possono applicare per tutti i sistemi non solo OO!

MODULO

Definizione molto generale!!!

- È un'entità SW, identificata da un **nome**
 - che può **fornire servizi** software:
 - contiene istruzioni, strutture dati, controllo
 - **può essere incluso** in un altro modulo
 - **può usare** un altro modulo o parte di esso
 - relazione 'dipende da'



A dipende da B

- Esempi di moduli:
 - una macro, un programma, un sottoprogramma, una funzione, uno script, un gruppo di programmi, un gruppo di sottoprogrammi, una classe, un metodo, un package,

1.ASTRAZIONE

Permette di concentrarsi su un problema ad un determinato livello di astrazione, senza perdersi in dettagli irrilevanti

- Riduce la complessità
- Permette di descrivere il comportamento di un modulo senza preoccuparsi dei dettagli
 - **Nasconde informazioni** che a quel livello non servono

```
Sort L in  
nondecreasing order
```

1.ASTRAZIONE

Permette di concentrarsi su un problema ad un determinato livello di astrazione, senza perdersi in dettagli irrilevanti

- Riduce la complessità
- Permette di descrivere il comportamento di un modulo senza preoccuparsi dei dettagli
 - **Nasconde informazioni** che a quel livello non servono

```
DO WHILE I is between 1 and (length of L)-1
  Set LOW to index of smallest value in L(I), ..., L(length of L)
  Interchange L(I) and L(LOW)
```

```
END DO
```

Selection Sort

1.ASTRAZIONE

Permette di concentrarsi su un problema ad un determinato livello di astrazione, senza perdersi in dettagli irrilevanti

- Riduce la complessità
- Permette di descrivere il comportamento di un modulo senza preoccuparsi dei dettagli
 - **Nasconde informazioni** che a quel livello non servono

```
DO WHILE I is between 1 and (length of L)-1
    Set LOW to current value of I
    DO WHILE J is between I+1 and (length of L)
        IF L(LOW) is greater than L(J)
            THEN set LOW to current value of J
        ENDIF
    END DO
    Set TEMP to L(LOW)
    Set L(LOW) to L(I)
    Set L(I) to TEMP
END DO
```

FORME DI ASTRAZIONE

◦ Funzionale:

- definizione di una **funzionalità** indipendentemente dall'algoritmo che la implementa

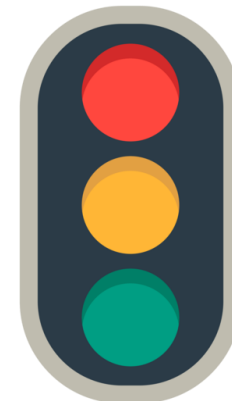
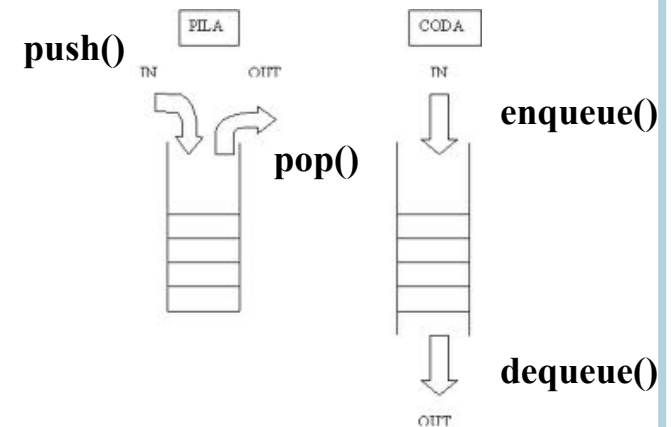
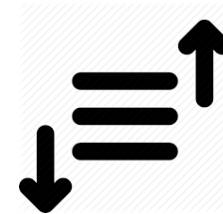
◦ di Dati:

- definizione di un **tipo di dato** in base alle **operazioni** che su di esso possono essere fatte, senza definirne una struttura concreta

◦ di Controllo:

- definizione di un **meccanismo di controllo** senza indicarne i dettagli interni
 - ad esempio **semafori**, per sincronizzare l'accesso a risorse condivise tra task (realizza mutua-esclusione)

Sort L in decreasing order



2. DECOMPOSIZIONE

Cercare di risolvere un problema in una volta sola è in genere più difficile che risolverlo per parti (cioè scomponendolo)

P = problema

$C(P)$ = complessità di P

$E(P)$ = effort (sforzo) per la risoluzione (software) di P

Legenda

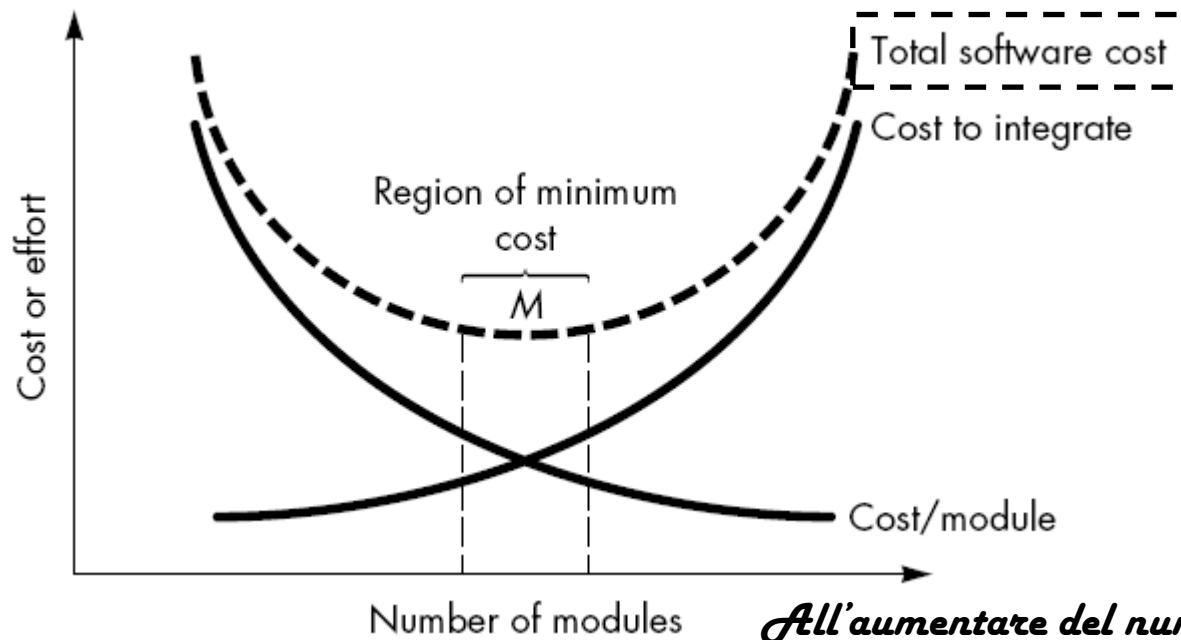
Dati due problemi $P1$ e $P2$

- Se $C(P1) > C(P2)$ allora vale $E(P1) > E(P2)$
- $C(P1+P2) > C(P1) + C(P2)$ *vale empiricamente*

quindi: **$E(P1+P2) > E(P1) + E(P2)$**

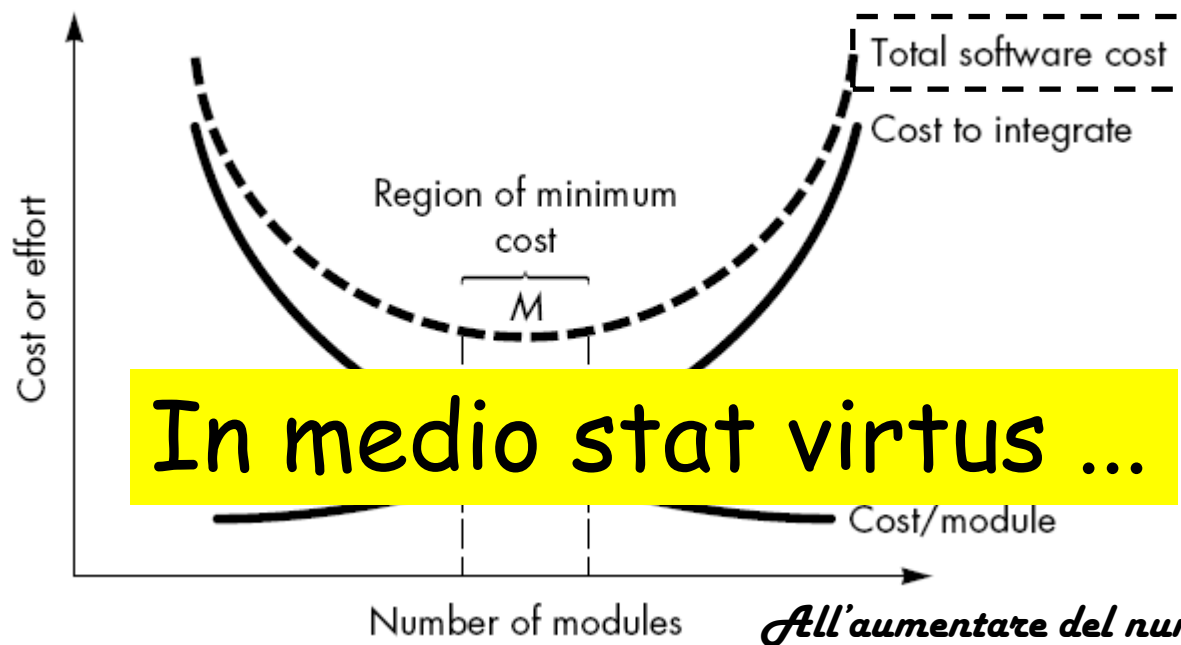
DISEQUAZIONE SBAGLIATA?? $E(P1+P2) > E(P1) + E(P2)$

- La disequazione precedente porterebbe alla conclusione sbagliata **che se noi dividessimo il problema e quindi il software infinite volte l'effort di sviluppo diventerebbe nullo** ...
- Purtroppo entrano in gioco altre variabili: aumentando i moduli **l'effort di integrazione** aumenta



DISEQUAZIONE SBAGLIATA?? $E(P1+P2) > E(P1) + E(P2)$

- La disequazione precedente porterebbe alla conclusione sbagliata **che se noi dividessimo il problema e quindi il software infinite volte l'effort di sviluppo diventerebbe nullo** ...
- Purtroppo entrano in gioco altre variabili: aumentando i moduli **l'effort di integrazione** aumenta



3. MODULARITÀ (SEPARATION OF CONCERNS)

È una conseguenza del principio di decomposizione ...

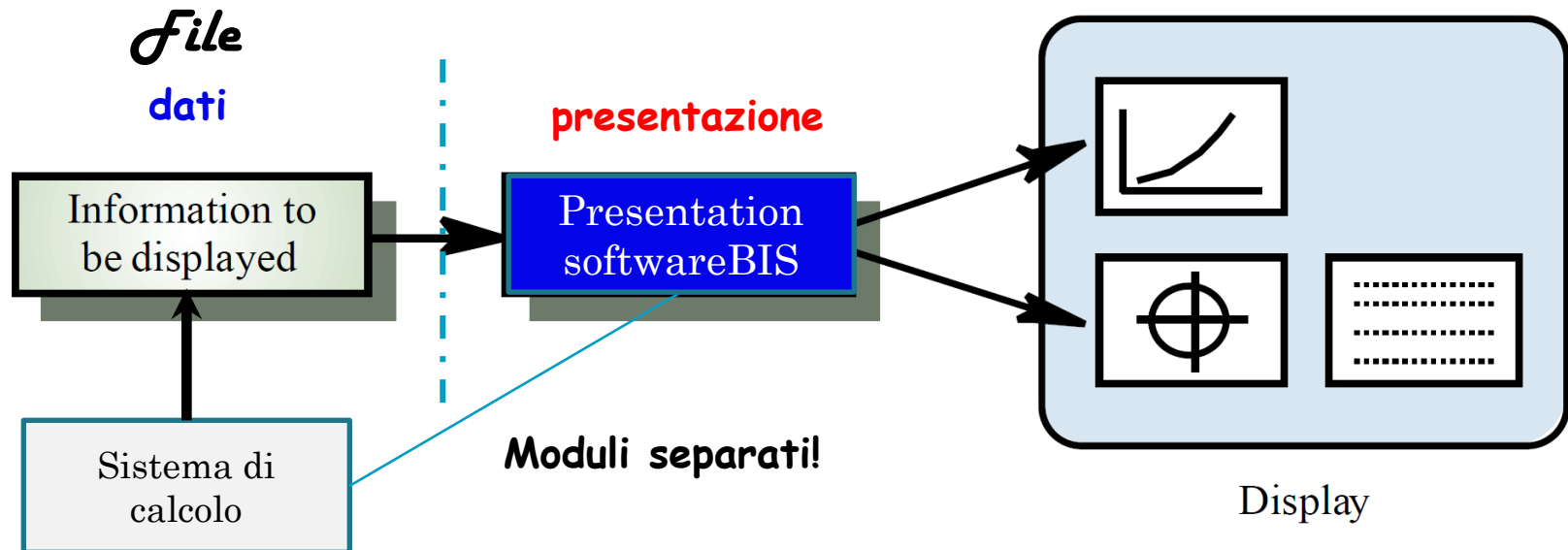


*Non basta però decomporre
il SW in moduli
occorre farlo bene ...*

- Il principio di decomposizione consiglia di dividere un problema (software) in tanti piccoli problemi (moduli)
- Questo principio invece ci dice **come dividere** un problema (software)
- **Separation of concerns**: l'idea è quella di tenere separati gli aspetti “unrelated” di un software

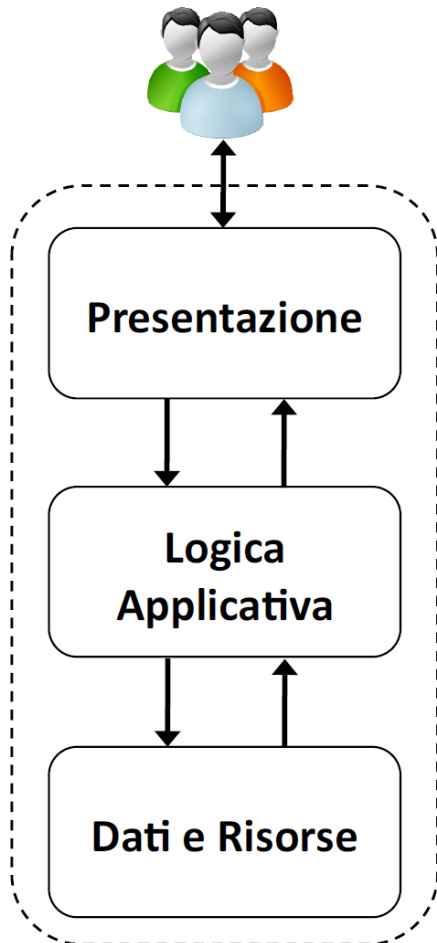
ESEMPIO DI “SEPARATION OF CONCERNS”

- Una valida linea guida è tenere separati il SW di generazione **dati** dal SW necessario alla loro **presentazione**
- Permette di cambiare la rappresentazione sullo schermo senza dover modificare il **sistema di calcolo**



ALTRO ESEMPIO DI 'SEPARATION OF CONCERNS ...'

- La struttura di una applicazione software, è **spesso** caratterizzata da tre livelli



- **Presentazione:** insieme dei moduli che gestiscono l'**interazione con l'utente**

- **Presentation logic**

- **Logica Applicativa:** insieme dei moduli che realizzano la logica applicativa, implementano le funzionalità richieste e gestiscono il flusso dei dati

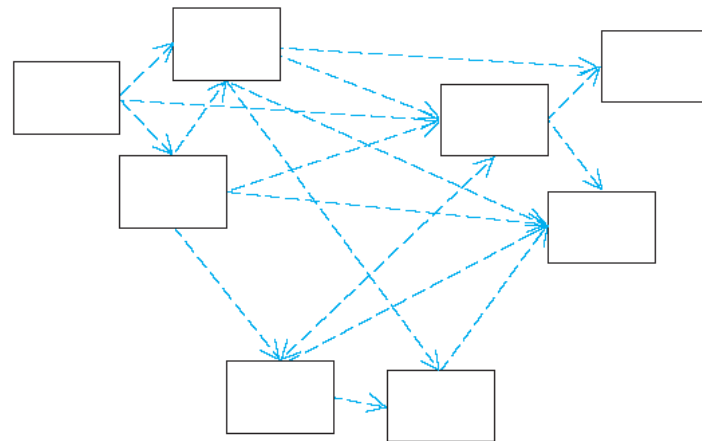
- **Business logic**

- **Dati e Risorse:** insieme dei moduli che gestiscono i dati che rappresentano le informazioni utilizzate

- **Data logic**

CRITERI PER LA MODULARIZZAZIONE

- Sostanzialmente **massimizzare la coesione** e **minimizzare l'accoppiamento**
 - Fare in modo che ogni modulo esegua un singolo compito (**The Single Responsibility Principle**)
 - Minimizzare il numero e la complessità delle interconnessioni fra moduli



High coupling

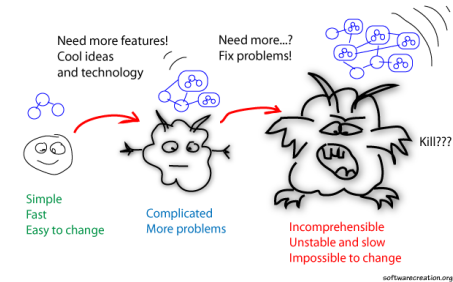


Moduli coesi e poco accoppiati

- facili da comprendere
- riusabili
- semplici da modificare
- semplici da testare

MODULARITÀ: EVITARE I MOSTRI ...

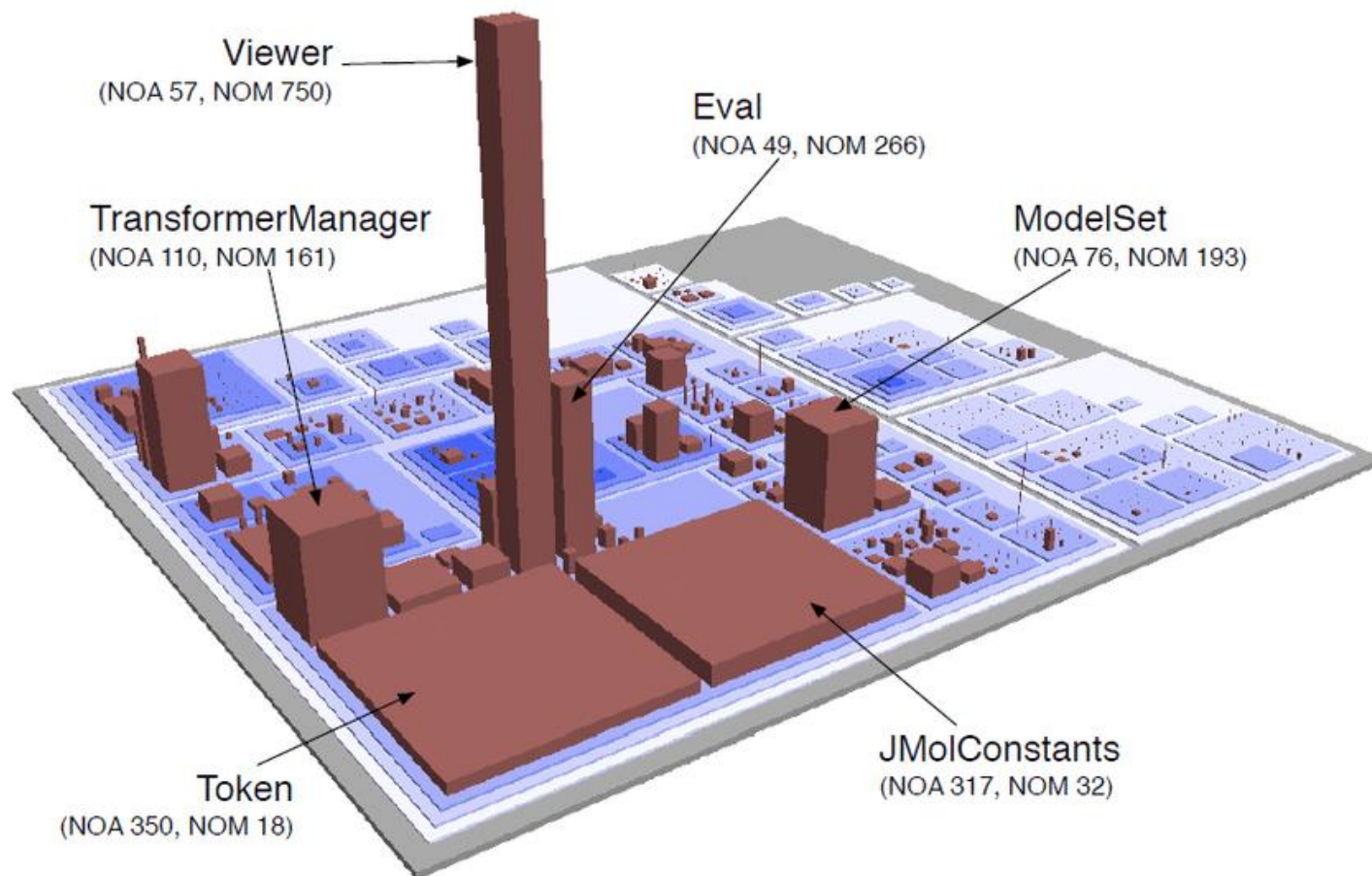
- Evitare moduli (es. classi e metodi in OO) “**monster**” e trovare la “giusta” misura
- “Large pieces” di codice sono impossibili da capire e mantenere



- Come evitare i “mostri”?

- Seguire i criteri per una buona modularizzazione
 - Moduli piccoli (no God class!)
 - Separation of concerns
 - Alta coesione e basso accoppiamento!
- Usare delle **metriche** o delle **view** per vedere se i criteri sono stati applicati
 - Esistono tool ad esempio **Stan4J** e **CodeCity**

CODECITY



Sistema SW: **Jmol** è un visualizzatore open source che permette di visualizzare le strutture molecolari in 3D

COESIONE

Una modulo coeso svolge **un unico compito** richiedendo poche interazioni con altri componenti

- Un modulo avrà elevata coesione se contiene al suo interno elementi correlati fra di loro e “lascia fuori” il resto
- **Coesione = un modulo deve esprimere una sola astrazione** (una funzione, un oggetto, un oggetto generico, una politica, un controllo)
- La coesione permette di comprendere e modificare meglio ogni singolo modulo

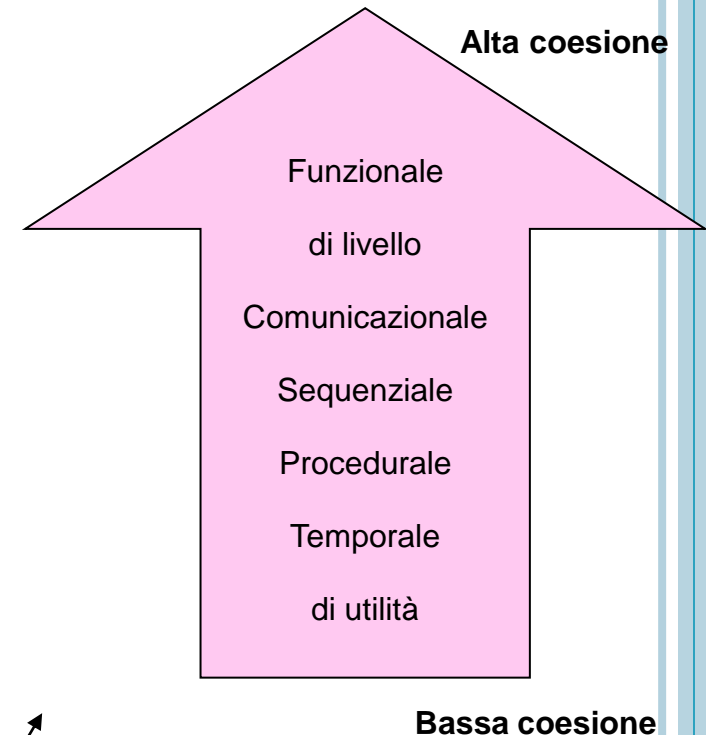
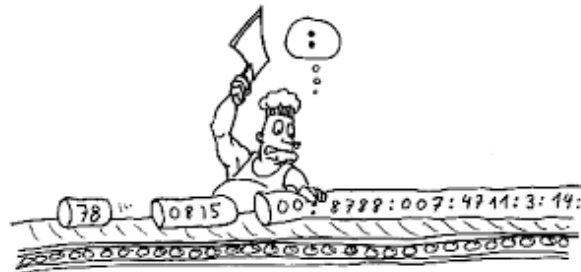
TIPOLOGIE DI COESIONE

- Di solito quando si parla di coesione si intende quella

Funzionale:

- Tutti gli elementi del modulo contribuiscono ad un **singolo ben definito task**

- Es. String Tokenizing, gestire una coda o pila, gestire conto corrente, ...



Esistono diversi tipi di coesione: vedere Pfleeger

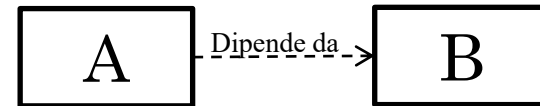
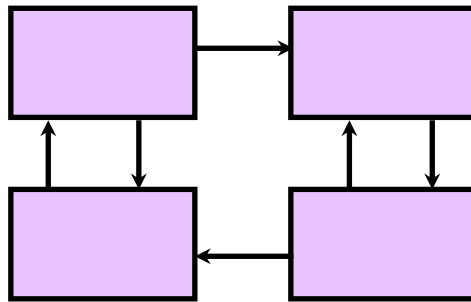
ESEMPI

- Coesione (di utilità)
 - Classe **Math** con i seguenti metodi/attributi:
Sin(), Cos(), Asin()
Sqrt(), Pow(), Exp()
Math.PI, Math.E
- Bassa coesione**
- Bassissima coesione o meglio nulla ;-)
 - Classe **Magic** con i seguenti metodi:

```
public void PrintDocument(Document d);  
public void SendEmail(  
    string recipient, string subject, string text);  
public void CalculateDistanceBetweenPoints(  
    int x1, int y1, int x2, int y2)
```

ACCOPPIAMENTO (1)

Ogni modulo non deve dipendere da troppi altri moduli, né dipendervi in modo troppo forte



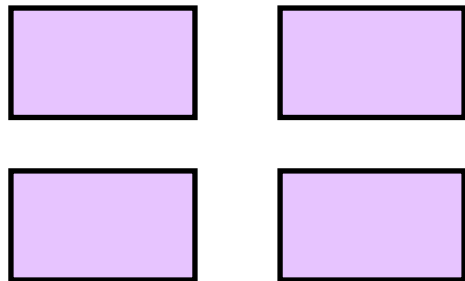
Il coupling misura (informalmente) il **grado di dipendenza di un modulo dagli altri**

- Dipendenza: es. chiamo una routine/procedura/funzione/metodo

Se ci sono dipendenze, spesso:

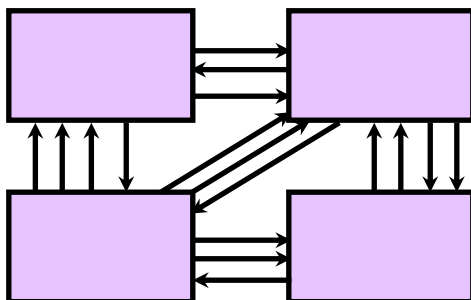
- le modifiche in un punto richiederanno modifiche anche altrove
- è difficile capire come un modulo lavora effettivamente
- non si può riusare un modulo senza tutti gli altri

ACCOPPIAMENTO (2)



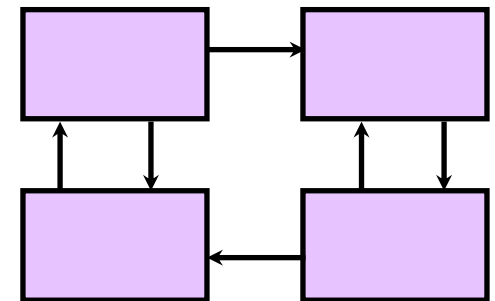
Uncoupled -
no dependencies

Irrealistico



Tightly coupled -
many dependencies

Giusto compromesso



Loosely coupled -
some dependencies

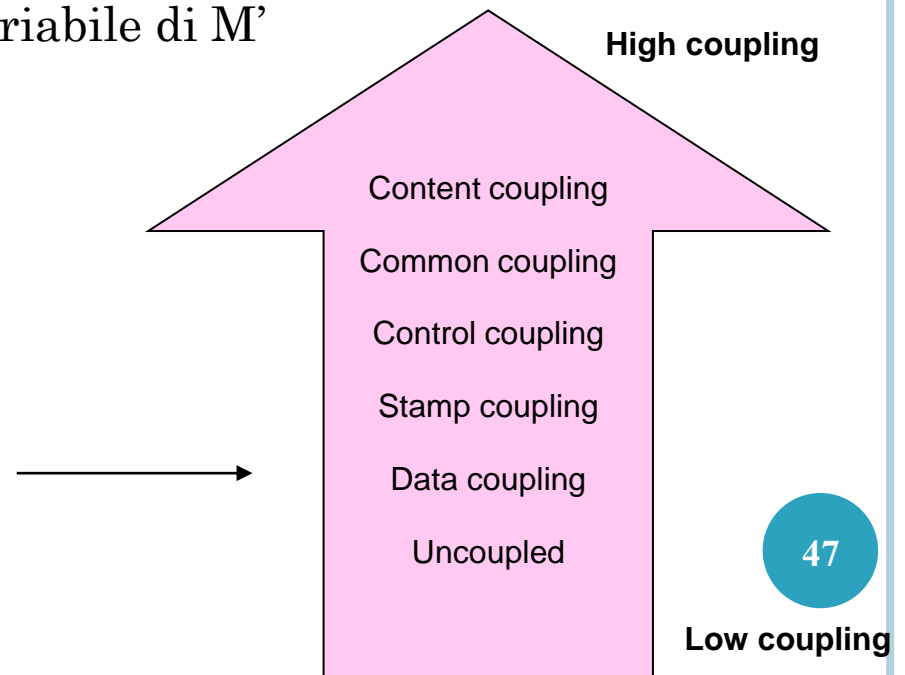
Da evitare

In medio stat virtus ...

ACCOPPIAMENTO 'BUONO' E 'CATTIVO'

- Esistono diversi tipi di accoppiamento tra moduli: **alcuni sono ammessi altri no** perchè rendono il sistema troppo difficile da capire, modificare e riusare
 - Esempi:
 - 'Buono'**: chiamata di routine/metodo/funzione di altro modulo
 - 'Cattivo'**: content coupling
 - M modifica il valore di una variabile di M'

Esistono diversi tipi di accoppiamento: vedere Pfleeger



ACCOPPIAMENTO 'BUONO'

- **Chiamata di funzione:** quando una funzione (o un metodo) ne chiama un'altra
 - la prima dipende dal comportamento dell'altro
 - è una forma di accoppiamento **inevitabile**
- **Uso di tipo:** quando un modulo usa un tipo di dato definito in un altro modulo
 - se cambia la definizione del tipo, anche i clienti del tipo possono dover cambiare
- **Inclusione o importazione:** quando un componente importa un package (Java) o quando un componente include una libreria (C/C++)
 - il componente che include o importa componenti dipende da tutto ciò che si trova nel componente incluso o importato

```
Public class A {  
    private B b;  
    public void doX() {  
        b.doA()  
        b.doB()  
    }  
}
```

```
Public class A {  
    public void doX() {  
        private B b  
        ...  
    }  
}
```

ACCOPPIAMENTO 'CATTIVO'

- **Content:** un modulo riesce a:
 - modificare dati interni ad un altro modulo oppure
 - “saltare” all’interno di un altro modulo oppure
 - alterare uno statement in un altro modulo (**Reflection**)
- Va evitato perché complica **enormemente** la comprensione e la modifica
- In un sistema OO si riduce *incapsulando* tutti i campi di una classe (inoltre non esiste GOTO!)
 - dichiarandoli privati
 - fornendo i metodi di get e set
- Nei linguaggi ‘Legacy’ (es. Fortran, Cobol, C, ...) le cose sono più complicate: GOTO e puntatori

Information hiding

ESEMPIO DI ACCOPPIAMENTO 'CATTIVO'

```
class MathParams
{
    public static double operand;
    public static double result;
}
class MathUtil
{
    public static void Sqrt()
    {
        MathParams.result = CalcSqrt(MathParams.operand);
    }
}
class MainClass
{
    static void Main()
    {
        MathParams.operand = 64;
        MathUtil.Sqrt();
        Console.WriteLine(MathParams.result);
    }
}
```

Content coupling

ESEMPIO DI ISTRUZIONE GOTO

```

400 REM
401 REM Comando di download
402 REM
403 OPEN FILE$ FOR INPUT AS #2
405 PRINT: PRINT: INPUT " Premere return, quando è pronto ", BS$
410 ON ERROR GOTO 1100
450 IF EOF (2) THEN GOTO 720
500 LINE INPUT #, D$
510 STATUS%=INP(&3FE)
520 STATUS%=STATUS% AND TS%
600 IF STATUS% = 0 THEN GOTO 510 ELSE PRINT #1, D$;
CHR$(13);CHR$(10);
610 PRINT D$
650 FOR J=0 TO 300: NEXT J
700 GOTO 450
710 PRINT #1,CHR$(26);CHR$(13);CHR$(10);
720 CLOSE #2
730 GOTO 1000
750 IF INSTR(D$, "END") <> 0 THEN GOTO 720
1000 ON ERROR GOTO 0: RETURN
1100 RESUME: GOTO 600
2000 REM
2001 REM Comando di Upload
2002 REM

```

leggi un'altra linea

4. INFORMATION HIDING

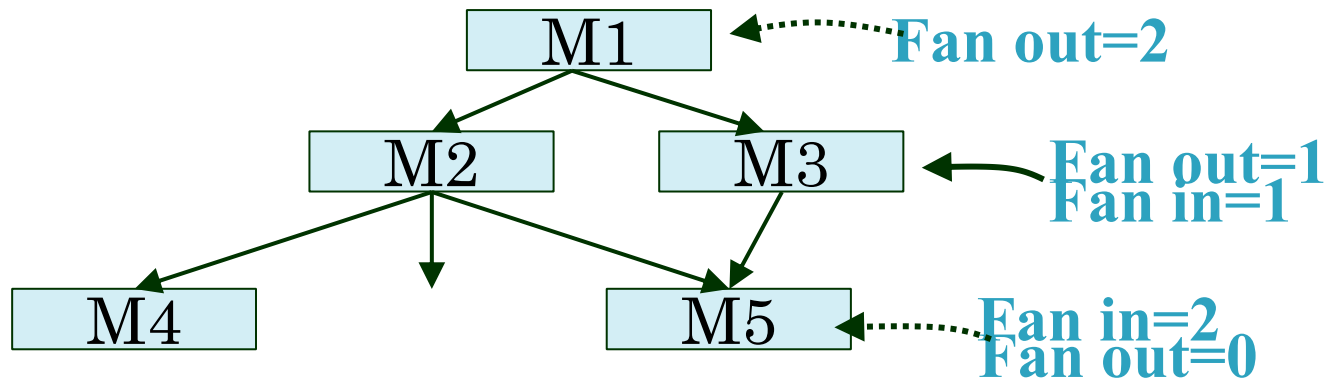
I moduli dovrebbero essere specificati e progettati in modo tale che le **informazioni** (algoritmi e dati) **risultino nascoste** agli altri moduli che non hanno bisogno di tali informazioni



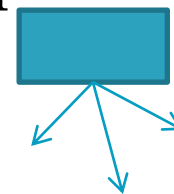
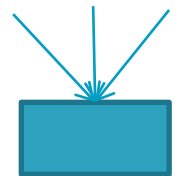
- Ogni modulo incapsula una decisione di design separata che potrebbe essere modificata in futuro
 - Es. Implementare uno stack con un array (modifico con una lista)
- Le **interfacce** sono utilizzate per descrivere ogni modulo in termini delle sue proprietà visibili esternamente
- Un vantaggio dell’information hiding è che i moduli risultano **debolmente accoppiati**
 - Se cambio algoritmo o struttura dati del modulo M ma lascio invariato il behavior, i moduli che usano M non cambiano

5. ALTO FAN-IN E BASSO FAN-OUT

- E' possibile costruire il grafo degli usi (**o dipendenze**) in cui gli archi rappresentano la relazione dipende da:
 - Fan-in = numero di archi entranti in un modulo
 - Fan-out = numero degli archi uscenti da un modulo

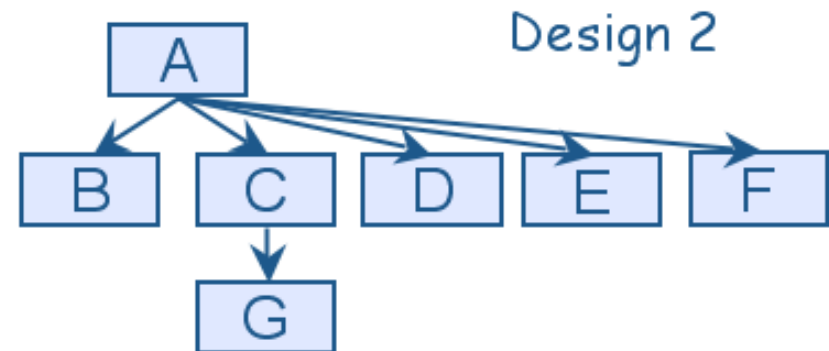
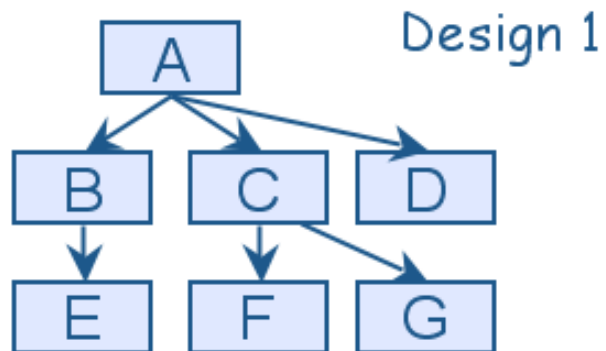


- Un alto numero di Fan-in indica buon riuso
 - se eccessivamente alto può indicare un problema di 'collo di bottiglia'
- Un alto numero di Fan-out indica eccessiva dipendenza e che il modulo "fa troppo" (va decomposto)



ESEMPIO

- I seguenti sono due possibili design per lo stesso sistema SW



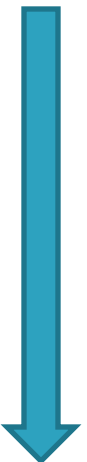
- Il Design 1 è **migliore** del Design 2 perché il modulo A nel Design 2 ha fan-out=5 (alto)
 - Nel Design 2 il modulo A andrebbe decomposto

6. GENERALITÀ

- È la proprietà di design che “migliora” il **riuso** di un modulo in altri progetti (in futuro)
 - Si cerca di rendere un modulo il più generale possibile in modo da **poterlo usare in più contesti**

- Esempio:

Warning: principio controverso perché 'peggiora' un altro principio quello della 'semplicità' (vedi dopo)



```
PROCEDURE SUM (a, b, c: INTEGER): INTEGER;  
POSTCONDITION: returns sum of parameters
```

```
PROCEDURE SUM (a[]: INTEGER; len: INTEGER): INTEGER  
PRECONDITION: 0 <= len <= size of array a  
POSTCONDITION: returns sum of elements 1..len in array a
```

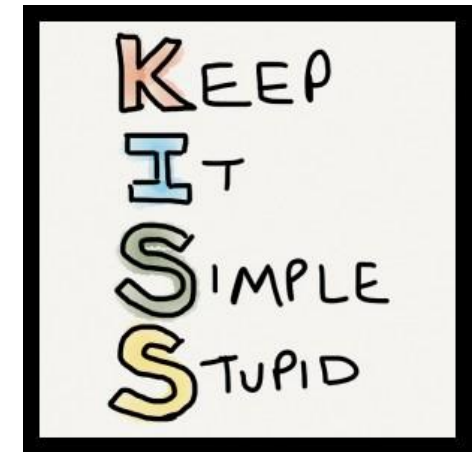
```
PROCEDURE SUM (a[]: INTEGER): INTEGER  
POSTCONDITION: returns sum of elements in array a
```

+ generale

PRINCIPIO KISS

- Il design dovrebbe essere il più semplice possibile

- Semplici i moduli
- Semplici le strutture dati
- Semplici le interfacce
- Semplici i metodi/procedure
- Semplice la comunicazione tra moduli
- No componenti generiche ...

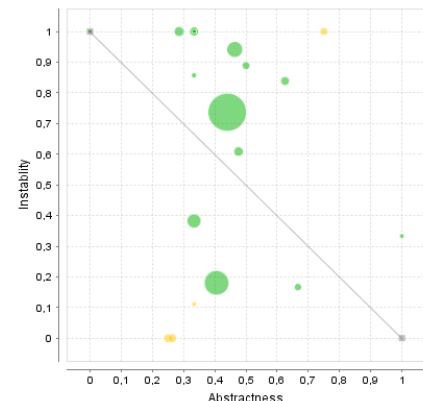


- Principio **KISS** (vale anche per la codifica)
 - **Keep It Simple, Stupid!**
 - **Keep It Short and Simple**

Versione moderna del rasoio di Occam ...

COME CAPIRE CHE STIAMO SEGUENDO I PRINCIPI?

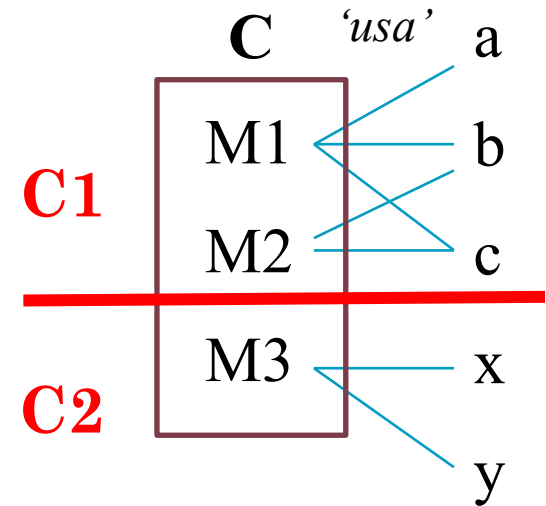
- **Esistono delle metriche del SW** che ci permettono di misurare l'aderenza a (parte dei) principi che abbiamo visto
 - Metrica SW = (IEEE Std 610.12-1990) **misura quantitativa** del grado di possesso di uno specifico attributo da parte di un sistema, un componente, o un processo
 - Es. Complessità ciclomatica (McCabe), LCOM (Lack of Cohesion)
- **Esistono dei tool** che forniscono metriche e viste
 - Es. Stan4J
 - Analizza il codice sorgente!!!



LCOM - ESEMPIO

- Si consideri una classe C con

- tre metodi: M1, M2, M3
- cinque field: a, b, c, x, y



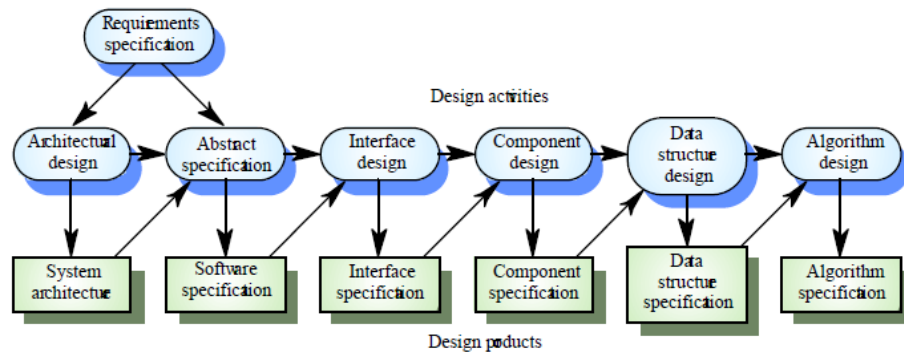
- M1 e M2 sono coesi perché usano le stesse variabili o meglio l'intersezione delle variabili 'usate' è non vuota
- M3 non è invece coeso con M1 e M2

$LCOM \sim n^{\circ} \text{ intersezioni vuote} - n^{\circ} \text{ intersezioni non vuote}$

- Quindi:

- $I1 = \{a, b, c\}$, $I2 = \{b, c\}$ e $I3 = \{x, y\}$
- Si ha: $I1 \cap I2 \neq \emptyset$ ma $I1 \cap I3 = \emptyset$ e $I2 \cap I3 = \emptyset$
- $LCOM = 2 - 1 = 1$ (**$LCOM > 0 \rightarrow$ classe non coesa**)

CONCLUSIONI



```

SORT (TABLE, SIZE OF TABLE)
  IF SIZE OF TABLE > 1
    DO UNTIL NO ITEMS WERE INTERCHANGED
      DO FOR EACH PAIR OF ITEMS IN TABLE (1-2, 2-3, 3-4, ETC.)
        IF FIRST ITEM OF PAIR > SECOND ITEM OF PAIR
          INTERCHANGE THE TWO ITEMS
  
```

PDL

