**PROGETTAZIONE DEL SOFTWARE**

Secondo lo standard ISO9126 la qualità è definita secondo 6 caratteristiche (ed ognuna di esse è poi definita in termini di sotto-caratteristiche):

1. Funzionalità: l’insieme di attributi che riguardano l’esistenza di un complesso di funzioni e delle specifiche proprietà. Le funzioni sono quelle che soddisfano i requisiti stabili o impliciti. Nella prima versione dello standard era chiamata correttezza e riguarda il che cosa deve fare un’applicazioni (requisiti funzionali) e le altre caratteristiche riguardano il come (requisiti non funzionali);
2. Affidabilità: l’insieme di attributi che riguardano la capacità del prodotto software di mantenere il livello di prestazioni con condizioni e limiti di tempo fissati; è differente dai sistemi fisici in quanto il software non si logora. Un software si può guastare e la sua frequenza di guasti dipende o dal prodotto software o dal modo in cui il prodotto software viene utilizzato;
3. Usabilità: l’insieme di attributi che riguardano lo sforzo necessario all’uso del prodotto e alla valutazione individuale di tale uso relativamente a un insieme di utenti prefissato o implicito. Le categorie di utenti possono essere: utenti finali, operatori, installatori, …;
4. Efficienza: l’insieme di attributi che riguardano la relazione esistente tra il livello delle prestazioni del prodotto software e la quantità di risorse necessarie nell’ambito di condizioni fissate. Le tipiche risorse sono lo spazio, lo spazio di memoria e l’occupazione di banda;
5. Manutenibilità: l’insieme di attribuiti che riguardano lo sforzo necessario ad eseguire modifiche; la manutenzione può essere di 3 tipi: correttiva (corregge gli errori), perfettiva (migliora il prodotto) e adattiva (migliora la compatibilità del prodotto con la piattaforma);
6. Portabilità: l’insieme di attributi che riguardano la capacità del prodotto software di essere trasferito da un ambiente all’altro. Le differenze tra gli ambienti possono: basarsi sull’organizzazione, basarsi sul software o basarsi sull’hardware.

Un insieme di numeri interi può essere rappresentato in molti modi diversi:

|  |
| --- |
| public class InsiemeDiInteri {  public final static int CAPACITA = 10;  public int n;  public int elenco[] = new int [CAPACITA];  Non mettere i dati public! |

Se si cambiano solo i dettagli della struttura dati e si mantiene il significato inalterato saranno inevitabili cambiamenti anche nelle altre applicazioni:

|  |
| --- |
| public class NodoListaInteri {  public int dato;  public NodoListaInteri next;  …  }  public class InsiemeDiInteri {  public NodoListaInteri primoNodo;  …  } |

|  |  |
| --- | --- |
| Prima | Dopo |
| public class GestioneTarghe {  public InsiemeDiInteri insiemeTarghe;  …  public int cercaTarga(int numeroTarga) {  int i;  while (i < insiemeTarghe.n && numeroTarga  != insiemeTarghe.elenco[i] ){  i++; }  if(numeroTarga != insiemeTarghe.elengo[i] )  { return -1; }  else  { return 1; }  } | public class GestioneTarghe {  public InsiemeDiInteri insiemeTarghe;  …  public NodoListaInteri cercaTarga(int numeroTarga) {  NodoListaInteri rifNodo = insiemeTarghe.primoNodo;  while (rifNodo != **nul**l && numeroTarga != rifNodo.dato) {  rifNodo = rifNodo.next; }  return rifNodo;  } |

Sarà quindi necessario cambiare tutte le istruzioni che facevano riferimento alla vecchia struttura dati e solo le istruzioni che facevano riferimento alla vecchia struttura dati. I problemi che si presentano sono il tempo e lo sforzo mal spesi e la possibilità di introdurre nuovi errori. Se più funzioni possono accedere ai dati liberamente, crescono le possibilità di avere problemi di correttezza nell’uso delle informazioni e nella modifica delle informazioni.

Il riutilizzo di un software può creare problemi ma porta anche dei benefici:

1. Tante rappresentazioni diverse possono avere un unico tipo (Es. posso creare più programmi che si basano sul tipo integer);
2. Un tipo di dato astratto: i metodi devono riportare gli stessi risultati indipendentemente dalla rappresentazione scelta;
3. Un tipo di dato può essere riusato da tanti programmi senza modifiche.

Soluzioni ad alcuni problemi possono riguardare:

1. Il contenuto (l’insieme di dati cui si vuole accedere);
2. L’interfaccia (ovvero la modalità di accesso ai dati di un tipo di dato astratto).

Altri benefici che ci possono essere grazie al riuso: una maggiore affidabilità, un minore costo e un minore tempo di sviluppo.

C’è differenza tra le classi e gli oggetti:

1. Una classe definisce un tipo, ossia;
2. L’insieme dei valori dati;
3. L’insieme delle operazioni ammesse su quei dati;
4. La dimensioni di quei dati.
5. Una classe fornisce servizi ai clienti della classe, ovvero modalità per la definizione e l’uso dei dati membro;
6. Un oggetto è una variabile il cui tipo è una classe.
7. L’istanziazione di una classe vs istanziazione di un oggetto:
8. In Java, la dichiarazione: *identificatore\_classe identificatore* è la dichiarazione di un riferimento, non di un oggetto;
9. Gli oggetti devono essere **creati** (identificatore = new identificatore\_classe();
10. Un oggetto è anche detto istanza (= un esemplare) di una classe, anche se non tutte le classi vengono poi effettivamente istanziate in oggetti e non tutte le classi possono essere istanziate in oggetti (classi astratte).

Classi ed oggetti contengono come elementi:

1. Dati, detti dati membro;
2. Metodi, detti metodi membro.
3. Dati e metodi esterni a una classe si dicono non membro quando ci si riferisce alla classe o a un oggetto istanza della classe;
4. Elementi (dati e metodi)
5. Di istanza: ne esiste una copia indipendente per ogni oggetto istanza della classe;
6. Di classe (static): ne esiste una copia unica comune a tutti gli elementi della classe.

**Esercizio:**

|  |
| --- |
| Siano dati 2 file così definiti:  //primo file  package A;  public class C1 {  public void m1() { }  protected void m2() { }  private void m3() { }  }  //secondo file  package B;  import A.\*;  Metodo m1() ridichiarato (adombramento del precedente  public class C2 extends C1 {  public void m1() { System.out.print(“Salve”); m2(); m3(); }  protected void m2() { System.out.print(“, mondo”); }  private void m3() { System.out.print(“!”); }  }  Dire se tale definizione è scorretta. Se non è scorretta, dire che cosa produce in output il frammento di codice:  Utilizzo new(costruttore) perciò punto “x” alla classe C2. “x” è il riferimento alla classe C2.  m1() è il metodo della classe a cui “x” fa riferimento. “x” in sostanza, è un costruttore  import A.\*;  import B.\*;  …  C2 x = new C2();  x.m1();  Con x.m1() chiamo il metodo dinamico. m1() sarà quello di C2  --------------------------------------------------------------------------------  La soluzione è corretta e quindi compila.   * Scriverà il metodo m1() di C2 e prosegue scrivendo m2() di C2 perché “protected” permette l’utilizzo del metodo da tutte le classi dello stesso package (non permette l’uso del metodo del package A perché è un package differente). * Non posso richiamare l’m3() di C1 perché in C1 è privato, perciò richiamo m3() di C2.   L’output finale è: “Salve, mondo!”.  **Modificatori di visibilità:**  public -> metodo o attributo visibile a tutte le classi.  protected -> metodo o attributo visibile solo da classi dichiarate nello stesso package.  private -> metodo o attributo visibile solo dalla classe in cui è dichiarato. |

Elementi **pubblici** di una classe/un oggetto sono:

1. Dati membro (**ASSOLUTAMENTE SCONSIGLIATO**);
2. Metodi.

Sono visibili a tutte le classi e a tutti gli oggetti e sono qualificati dalla parola chiave ***public***. Il meccanismo di incapsulamento serve per nascondere i dati e permettere di accedervi solo tramite le funzionalità dell’interfaccia. La parte pubblica non dovrebbe perciò contenere dati membro.

Elementi **protetti** di una classe/un oggetto sono quelli a cui possono accedere soltanto i metodi:

1. Della classe/degli oggetti della classe;
2. Delle sue sottoclassi/degli oggetti delle sue sottoclassi;
3. Delle classi appartenenti allo stesso package della classe.

Sono qualificati dalla parola chiave ***protected*** e a differenza degli elementi di una classe/un oggetto con visibilità limitata al package, viene esplicitata l’intenzione di rendere questi elementi riutilizzabili dalle sottoclassi che ereditano dalla classe.

Elementi di una classe/un oggetto **con visibilità limitata al package** sono:

1. Dati membro (**ASSOLUTAMENTE SCONSIGLIATO**);
2. Metodi.

Sono visibili a tutte le classi appartenenti allo stesso package della classe e ai loro oggetti. ***Non sono qualificati da alcuna parola chiave***. Il meccanismo di incapsulamento serve per nascondere i dati e permettere di accedervi solo tramite le funzionalità dell’interfaccia. La parte pubblica non dovrebbe perciò contenere dati membro.

Elementi **privati** di una classe/un oggetto sono:

1. Dati membro (meglio che non siano pubblici);
2. Metodi.

Non sono visibili all’esterno della classe/dell’oggetto. Sono qualificati dalla parola chiave ***private***.

Categorie dei metodi membro:

1. Estensore (metodo “set”):
   1. Metodo membro non privato che modifica (definisce) il valore dei dati membro;
2. Selettore (metodo “get”):
   1. Metodo membro non privato che accede al valore (usa il valore) dei dati membro senza modificarli;
3. Metodo di servizio (metodo “utility”):
   1. Metodo di membro privato;
4. Il tipo restituito da un metodo membro può essere qualsiasi, inclusa la classe stessa cui il metodo appartiene.

I costruttori sono metodi chiamati automaticamente durante l’esecuzione all’atto della creazione di un oggetto per inizializzare i dati membro dell’oggetto, hanno lo stesso nome della classe, ma senza il tipo ritornato (nemmeno void). Ci sono due tipi di costruttori:

1. Costruttori con parametri “generici”;
2. Costruttori per default.

Il costruttore può essere attivato, solo una volta, all’atto della creazione di un nuovo oggetto. I costruttori non possono essere chiamati esplicitamente dal programmatore su un oggetto dopo la creazione dell’oggetto. I dati membro di una classe/un oggetto sono inizializzati automaticamente a un valore “neutro” in mancanza di un valore esplicito (i dati membro int sono inizializzati a 0).

|  |  |
| --- | --- |
| Costruttore per default:   1. Non ha parametri; 2. Chiamata di un costruttore per default:   identificatore\_classe oggetto;  puntatore\_a\_oggetto = new identificatore\_classe();   1. Se non viene dichiarato nessun costruttore per una classe, si può prendere quello della superclasse più vicina della gerarchia di ereditarietà che contiene il costruttore per default (eventualmente risalendo fino alla classe *Object*). | Costruttore con parametri “generici”:   1. Hanno parametri; 2. Possono essere sottoposti a overloading; 3. Chiamata di un costruttore con parametri “generici”:   riferimento\_a\_oggetto = new identificatore\_classe(lista\_paramentri\_effettivi); |

Il finalizzatore è un metodo chiamato automaticamente durante l’esecuzione all’atto della distruzione di un oggetto del garbage collector, che è il processo che permette di rilasciare la memoria allocata per oggetti quando gli oggetti non servono più. Esprime le “ultime volontà” dell’oggetto prima che venga distrutto. La distruzione di un oggetto avviene tipicamente quando:

1. L’oggetto perde significato perché si è usciti dall’ambito in cui è dichiarato l’oggetto;
2. Si è eliminata la possibilità di accedere all’oggetto, ad esempio, dopo aver posto a *null* l’unico riferimento esistente per un oggetto.

Il finalizzatore è un metodo void il cui identificatore è *finalize*. Il finalizzatore viene chiamato dal garbage collector*,* il momento di esecuzione non viene controllato dal programmatore. Il finalizzatore può essere chiamato esplicitamente dal programmatore per distruggere un oggetto. Se non viene dichiarato il finalizzatore di una classe, si prende quello della superclasse più vicina della gerarchia di ereditarietà che contiene il finalizzatore. L’identificativo *finalize* non è una parola chiave del linguaggio. Il finalizzatore deve essere scritto con attenzione per non causare danni al funzionamento del programma, soprattutto nel caso di strutture dati diamiche.

Ogni oggetto è raggiungibile tramite riferimento, indicato dalla parola chiave *this* quando si scrive la dichiarazione della classe. La parola chiave *this* indica un riferimento che contiene l’indirizzo dell’oggetto stesso cui appartengono: i dati membro che si stanno manipolando e i metodi membro che si stanno utilizzando. La parola chiave *this* è implicita per i dati membro e i metodi membro, ossia si può omettere quando si menzionano dati membro e metodi membro qualora non vi siano pericoli di ambiguità. Alcuni utilizzi tipici ci sono:

1. Al termine di un metodo, restituire il riferimento all’oggetto che si è manipolato: *return this;* che permette di concatenare più chiamate di metodi;
2. All’interno di un metodo, per permettere di accedere a quei dati membro di una classe/un oggetto che sono nascosti dai dati locali del metodo che hanno lo stesso nome.

Gli operatori predefiniti di assegnamento e uguaglianza:

1. Effettuano l’assegnamento;
2. Controllano l’uguaglianza.

Dei riferimenti (assegnamento e uguaglianza superficiali) e non degli oggetti (assegnamento e uguaglianza profondi). Bisogna Definire comunque metodi in modo tale che sia possibile effettuare l’assegnamento e l’uguaglianza (metodo *equals*). Non è necessario definire un metodo apposito per l’assegnamento, ma si può utilizzare un costruttore “per copia”.

*Oggetto o1, o2;*

*o2 = new Oggetto( 01 );*

con la chiamata di

*public Oggetto( Offetto altroOggetto)*

che costruisce un oggetto per copia di *altroOggetto*.

I dati di membro *static* sono dati e metodi membro di cui deve esistere una sola copia per classe.

Dati membro:

1. *numeroOggetti* inizializzato con 0;
2. *intero* inizializzato con 14;
3. *oggetto1* inizializzato con *null*;
4. *oggetto2* inizializzato con il costruttore per default della classe *AltraClasse*.

I dati *static* vengono allocati indipendentemente dall’esistenza di oggetti della classe non appena inizia l’eleborazione.

I metodi *static* possono essere chiamati con:

*identificatore\_classe.identificatore\_metodo\_statico(lista\_parametri\_effettivi)* (**CONSIGLIATO**);

*identificatore\_oggetto.identificatore\_metodo\_statico(lista\_paramentri\_effettivi)* (**MEGLIO DI NO**);

Dati e metodi *static* non fanno riferimento a nessun oggetto, per cui non possono utilizzare il riferimento *this*.

I metodi *static* possono utilizzare solo dati e metodi *static* della stessa classe (oltre a metodi non *static* di oggetti di altre classi).

L’ereditarietà consente agli oggetti di una classe di essere oggetti anche di un’altra classe. Ad esempio gli oggetti di classe *Studente* sono casi particolari degli oggetti di classe *Persona.* In questo caso la classe *Studente* si dice sottoclasse della classe *Persona* mentre la classe *Persona* si dice superclasse della classe *Studente*.

In questo caso gli oggetti della classe *Studente*:

1. Hanno automaticamente tutte le proprietà (dati e metodi membro) degli oggetti della classe *Persona*;
2. Possono avere altre proprietà (dati e metodi membro) in più degli oggetti della classe *Persona*;
3. Possono ridefinire alcune proprietà degli oggetti della classe *Persona* per meglio adattarle alla natura degli studenti.

Si dice che la sottoclasse “eredita” dalla superclasse le proprietà:

*class SottoClasse extends SuperClasse {*

l’ereditarietà permette di non dovere riscrivere il codice tuttavia deve essere riutilizzata con cautela per evitare di commettere errori che possono rivelarsi in fase di compilazione o di esecuzione. Nella costruzione di una classe si possono sia aggiungere nuovi elementi rispetto alla superclasse sia ridefinire gli elementi protetti e pubblici della superclasse. La costruzione della sottoclasse non comporta alcuna modifica della superclasse inoltre nell’aggiunta e nella ridefinizione di elementi non si usa nella sottoclasse alcuna sintassi particolare.

Per riferirsi alla superclasse si utilizza la parola chiave *super*; è la chiamata al metodo *metodo* della superclasse. Non si può andare alla superclasse di una superclasse: *super.super.metodo(…);* è un errore di sintassi. All’inizio dell’esecuzione di un costruttore di una sottoclasse viene sempre chiamato un costruttore della superclasse. Se c’è una chiamata esplicita (*super(lista\_parametri\_effettivi)*) viene chiamato il costruttore corrispondente della superclasse individuato dalla lista dei parametri effettivi. Questa chiamata deve sempre essere la prima istruzione del costruttore della sottoclasse. Se non c’è una chiamata esplicita a un costruttore della superclasse viene chiamato implicitamente e automaticamente il costruttore per default della superclasse.

La gerarchia di ereditarietà tra le classi è un albero dove:

1. Una classe può ereditare da una sola classe;
2. L’ereditarietà multipla da classi non è consentita;
3. L’ereditarietà multipla è consentita solo per “ereditare” dalle *interface*;
4. La classe *Object* è la radice dell’albero della gerarchia (fa parte del package *java.lang* importato implicitamente);
5. Ogni classe implicitamente o esplicitamente eredita direttamente o indirettamente da *Object*.

Con la parola chiave *final* si designano:

1. Metodi che non possono essere ridefiniti nelle sottoclassi di una classe;
2. Classi che non possono essere ridefinite in sottoclassi;
3. I metodi *static*  sono anche *final* e non possono essere ridefiniti;
4. I metodi *private* possono essere ridefiniti ma comunque non sono visibili nelle sottoclassi.

Con la parola chiave *abstract* si designano:

1. Metodi che devono essere ridefiniti nelle sottoclassi di una classe;
2. Classi che devono essere ridefinite in sottoclassi;
3. Non è possibile di istanziare oggetti di classe *abstract*;
4. È sufficiente che un metodo sia *abstract* perché l’intera classe sia *abstract*.

Cona la parola chiave *interface* si designano “scheletri di classi” in cui i metodi sono tutti *public abstract* e i dati sono obbligatoriamente *public final static* anche se non sono scritti esplicitamente.

Le interfacce sono legate tra di loro dalla relazione di ereditarietà e possono ereditare da più interfacce.

Le interfacce sono legate alle classi dalla relazione di implementazione e una classe può estendere una classe e implementare più interfacce. Una classe non può ereditare più classi. Una classe non è sostituibile a più classi (al massimo una sola) ma può essere sostituibile a più interfacce.

Un oggetto di una sottoclasse può essere trattato come un oggetto di una delle sue superclassi mentre un riferimento ad un oggetto di una sottoclasse può essere assegnato al riferimento ad un oggetto di una delle sue superclassi. Un oggetto di una sottoclasse può essere trattato come un oggetto di una delle sue superclassi. Il riferimento a un oggetto di una superclasse è una variabile che può puntare a oggetti diversi durante l’esecuzione di un programma. Il riferimento a un oggetto di una superclasse può in ogni momento puntare a un oggetto della superclasse oppure a un oggetto di una qualunque delle sue sottoclassi. Ogni riferimento può puntare a un oggetto che assume “molte forme” (polimorfismo).

Il riferimento a un oggetto di una sottoclasse viene convertito implicitamente in un riferimento a oggetto di una delle sue superclassi. Quando si assegna il riferimento a un oggetto di una sottoclasse a un oggetto di una superclasse, bisogna stare attenti a quali dati e metodi si usano/possono usare quando si usa il riferimento all’oggetto della superclasse.

Se si effettua la conversione esplicita di un riferimento a un oggetto di una sottoclasse a un riferimento a un oggetto di una superclasse tramite una conversione (cast) di tipo, non cambia il risultato. Il riferimento a un oggetto di una superclasse può essere convertito solo esplicitamente in un riferimento a oggetto di una delle sue sottoclassi tramite una conversione (cast) di tipo.

|  |
| --- |
| Sia dato il seguente frammento di codice. Indicare gli errori a compile-time. Eliminare le istruzioni che generano errore a compile-time, e dire se il codice genera errori a runtime. Eliminare anche le istruzioni che generano errore a runtime, e dire che cosa produce in output il programma.  Package C;  A e B sono i package dell’esercizio 1  C1, c2, o sono i riferimenti  import A.\*; //vedere esercizio precedente  import B.\*;  public class C3 {  public static void main(String[] s) {  C1 c1; C2 c2; Object o;  c1 = new C1(); //1  c1.m1(); //2  c2 = new C2(); //3  c2.m2; //4  Staticamente è corretto  c1 = c2; //5  c1.m1(); //6  c2 = new C1(); //7  o = new C1(); //8  c2 = (C2) o; //9  o = new C2(); //10  Staticamente c1 a sinistra è compatibile con il C1 di destra e in runtime non genera errore in quanto dentro o c’è c2 che è un sottotipo di c1.  c1 = (C1) o; //11  c1.m1();  }  }  --------------------------------------------------------------------------------  Errori in fase di compile-time:  //4 genera errore perché m2 è “protected”.  //7 genera errore perché c2 è una sottoclasse di C1 e quindi servirebbe un cast.  Errori in fase di runtime:  //9 in o in quel momento c’è un C1 che è un sottotipo di C2  Il programma produce in output: “Salve, mondo!”.  //2 c1.m1() è void (Package A).  //4 c1.m1() è “Salve, mondo!” (Package B).  //12 c1.m1() è “Salve,mondo!” (Package B). |

L’attività di design produce l’architettura del software. I componenti possono essere definiti con diversi livelli di astrazione. Ci sono 2 livelli:

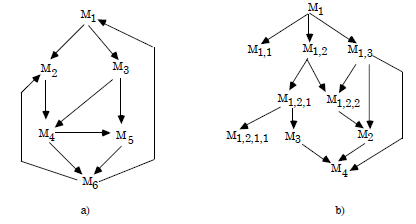
1. Livello 1: il meccanismo (da cosa sono costituiti e come sono aggregati);
2. Livello 2: lo stile.

Il meccanismo descrive com’è costruita una architettura; lo stile è quello che caratterizza un’architettura rispetto ad un’altra. Ad ogni livello si dovrebbe avere la possibilità di ragionare sull’architettura e sulle proprietà del sistema. Entrambi i livelli forniscono una descrizione “statica” dell’architettura. Un modulo è una parte del sistema che provvede a fornire servizi agli altri moduli. I servizi sono elementi computazionali che gli altri moduli possono utilizzare.

L’insieme di servizi forniti da un modulo costituiscono l’interfaccia del modulo. L’interfaccia definisce un contratto tra il modulo e il suo utilizzatore.

Le relazioni di un meccanismo:

1. USES: un modulo usa i servizi esportati dagli altri;
2. Is\_COMPONENT\_OF: descrive l’aggregazione di più moduli in un livello più alto di moduli;
3. INHERITS: serve per i sistemi object-oriented.

Le relazioni possono essere rappresentate come grafi

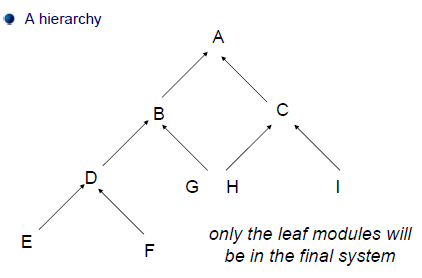
Vengono rappresentati i metodi delle classi, la foto (b) è migliore perché è ciclica e crea una gerarchia dove viene indicata una classe e una sottoclasse e i metodi che derivano dagli altri metodi.

La relazione USES:

1. A uses B: A può accedere ai servizi esportati da B attraverso la sua interfaccia, è staticamente definito. A dipende da B per fornire i suoi servizi. Si dice anche che A è cliente di B.

La relazione IS\_COMPONENT\_OF:

1. Viene usata per descrivere un modulo di alto livello costituito da più moduli di basso livello;
2. A IS\_COMPONENT\_OF B: B è costituito da diversi moduli, uno dei quali è A. B comprenda A.

tutte le classi sono astratte tranne le foglie quindi solo i moduli foglia risulteranno nel sistema finale. I moduli foglia sono E, F, G, H e I.

Se il sistema è sviluppato con uno stile object-oriented, la relazione di ereditarietà consente a un componente di estenderne un altro. Un erede può accedere ad alcuni componenti del predecessore.

Per sviluppare un software bisogna pensare a un programma e tutte le sue varianti come membri di una famiglia, l’obiettivo è di disegnare tutta la famiglia assieme, non ogni membro separatamente.

Ci sono 3 principi di design:

1. Come scegliere i moduli;
2. Come definire le interfacce dei moduli;
3. Come definire le USE relations.

Come scegliere i moduli:

1. Un modulo è un unità autonoma;
2. L’uso di interconnesioni con gli altri moduli dovrebbe essere minimo;
3. In principio: massimizza la coesione e minimizza l’accoppiamento.

Come definire le interfacce dei moduli:

1. Distinzione tra cosa fa il modulo per gli altri moduli e come li fa;
2. Minimizza il flusso di informazioni con i clienti e ne massimizza la modificabilità;
3. L’interfaccia è un contratto con i clienti e deve essere stabile;
4. Il principio d’oro si basa sul nascondere le informazioni (Parnas 1974), definisce cosa desideri nascondere e sviluppa un modulo attorno ad esso.