**OBJECT ORIENTED PARADIGM**

Nei linguaggi imperativi nessun modulo che accede ad una variabile globale può essere sviluppato indipendentemente dal contesto in cui si trova. David Parnas riconobbe questo problema, e notò che sistemi software complessi diventavano molto difficili da gestire. È nata così la disciplina dell’INFORMATION HIDING, sul quale è fortemente basato il paradigma di programmazione orientato agli oggetti.

Rispetto al paradigma imperativo, quello OO costituisce:

* Una evoluzione 🡪 Consente agli oggetti di essere cittadini di prima classe;
* Una rivoluzione 🡪 information hiding ed incapsulamento sono principi cardine nel paradigma orientato agli oggetti.

**CLASSIFICAZIONE DI WENGER (1987) DEI LINGUAGGI DI PROGRAMMAZIONE**

* Object-based 🡪 Supportano la notazione di oggetto;
* Class-based 🡪 Supportano la notazione di classe, e gli oggetti sono sempre istanze di classi;
* Object-Oriented 🡪 Supportano la notazione di oggetto, classe ed ereditarietà. Gli oggetti sono cittadini di prima classe.

**OGGETTI**

Oggetto = OID + stato + comportamento.

Aliasing 🡪 Variabili distinte che fanno riferimento al medesimo oggetto. Gli oggetti possono essere confrontati in due modi:

* Per riferimento: se due oggetti puntano allo stesso oggetto;
* Per confronto: se il contenuto della locazione di memoria dei due oggetti è uguale.

**CLASSE**

Una classe è la descrizione di una famiglia di oggetti che condividono la stessa struttura (attributi) e lo stesso comportamento (operazioni).

* Componente statica di una classe 🡪 Attributi di istanza (tempo di vita pari a quello della istanza) + Attributi di classe (Il tempo di vita è lo stesso della classe, STATIC);
* Componente dinamica di una classe 🡪 Metodi (costruttori, di accesso, di trasformazione, distruttori). I metodi di accesso e trasformazione possono essere di classe o di istanza.

Classi Entity – Classi Boundary – Classi Control

**EREDITARIETÀ**

Una classe è considerata come un repertorio di conoscenze, a partire dalla quale è possibile specificare delle sottoclassi che estendono e completano le conoscenze della classe madre.

*EREDITARIETÀ PER ESTENSIONE*

La sottoclasse introduce nuove caratteristiche (metodi o attributi) non presenti nella superclasse e non applicabili ad istanze della superclasse. Ottima soluzione per scrivere del codice estendibile.

*EREDITARIETÀ PER VARIAZIONE FUNZIONALE*

La sottoclasse ridefinisce una operazione (*overriding*) modificandone l’implementazione ma non la segnatura. La ridefinizione non è incrementale, quindi se apportiamo delle modifiche alla implementazione della classe madre, dobbiamo aggiungerle manualmente anche nelle sottoclassi. Non è facile ricordarselo ed è per questo che l’ereditarietà per variazione funzionale è vista come potenziale fonte di errori. Molti esperti ne sconsigliano l’uso. Una soluzione che potrebbe permetterne l’uso e mitigarne i difetti, sarebbe quello di attenersi alla anticipazione del cambiamento in fase progettuale (secondo me, è comunque dispendioso e non vale la pena applicare questo tipo di ereditarietà).

*EREDITARIETÀ PER RESTRIZIONE*

Una sottoclasse restringe il campo di azione della superclasse. Nel costruttori della sottoclasse è necessario specificare quali sono le restrizioni da apportare (esempio di una classe madre rettangolo, e di una sottoclasse quadrato. Nel costruttore di quadrato dobbiamo dire che la base è uguale all’altezza è che l’area magari si calcola facendo lato^2).

*EREDITARIETÀ DI IMPLEMENTAZIONE*

Una classe mette a disposizione la sua realizzazione per implementare la struttura di una sottoclasse. Immaginiamo un dato vettore che mette a disposizione la sua struttura per realizzare il dato pila. Una pila sarà basata sulla realizzazione del dato vettore ma non ne erediterà le operazioni, solo la implementazione. In UML si utilizza lo stereotipo <<implementation>>. L’ereditarietà di implementazione non è compatibile con il principio di sostituibilità, perché il principio di sostituibilità prevede che una sottoclasse non rinunci a nessun metodo della superclasse. In questo si rinuncia idealmente a tutti i metodi, perché la sottoclasse ha interesse ad ereditare esclusivamente l’implementazione della superclasse.

**LISKOV SUBSTITUTION PRINCIPLE (Terzo principio Solid)**

Data un dichiarazione di una variabile il cui tipo è dichiarato come X, qualunque istanza di una classe discendente da X può essere assegnata a tale variabile, e può essere usato come valore effettivo per quella variabile. Si deduce una conseguenza ovvia del principio di sostituibilità: una sottoclasse non può rimuovere o rinunciare ad alcun metodo della superclasse. Il principio di sostituibilità non è compatibile con l’ereditarietà di implementazione. Esempio: Dizionario d = new DizionarioCollegato();

**EREDITARIETA SINGOLA** 🡪 Catena di antenati;

**EREDITARIETA MULTIPLA** 🡪 Grafo aciclico orientato;

**CLASSI ASTRATTE**

Sono classi dalle quali non è possibile generare istanze. Qualunque classe astratta possiede almeno un metodo astratto, si dice infatti astratta perché non è completamente specificata. In UML il nome della classe e il nome dei metodi astratti vanno indicati in corsivo. Sono strumenti per fattorizzare proprietà comuni tra classi simili e poterle organizzare in una gerarchia di ereditarietà.

**CLASSI FINALI**

Una classe è detta finale o foglia quando non può essere ulteriormente specificata. Sono utilizzate per ragioni di affidabilità, e permette la generazione di codice ottimizzato, perché siamo sicuri che i metodi non verranno mai sovrascritti in sottoclassi.

**INTERFACCE**

Una interfaccia descrive il comportamento di una classe di oggetti senza specificarne una implementazione. È una collezione di operazioni, cioè di servizi messi a disposizione, che possono essere richiesti da delle classi concrete che implementano l’interfaccia. Ogni classe concreta specificherà la propria implementazione per tali servizi, cioè ogni classe concreta implementerà il suo metodo. Una interfaccia è simile ad una classe astratta, però i metodi sono tutti astratti e non può disporre di attributi. Servono appunto a disaccoppiare la definizione delle operazioni dalla loro implementazione. È consentito stabilire relazioni di ereditarietà fra interfacce, e differentemente dalle classi, l’ereditarietà multipla non causa problemi, perché non ci interessa l’implementazione delle operazioni.

**METACLASSI**

Sono classi che definiscono la struttura di altre classi. Le istanze di oggetti metaclasse sono tutti istanze della classe chiamata METACLASS. In Java è diverso, tutti ereditano dalla classe Object.

**AGGREGAZIONE**

Relazione debole. A in aggregazione con B se almeno una istanza di B contribuisce a formare una parte di una istanza di A. Le istanze di B posso avere vita propria. Quando usarla:

* Contenimento fisico (es. pagina di un libro);
* Appartenenza (es. membership);
* Composizione funzionale (es. composizione funzionale).

**COMPOSIZIONE**

Relazione forte. A è in composizione con B se una istanza di B può esistere solo se esiste A. Il tempo di vita di una istanza di B è la stessa della istanza “contenitore” di A.

**EREDITARIETÀ vs AGGREGAZIONE**

Nell’ereditarietà ogni istanza di una classe specializzata contiene tutti i campi definiti nella superclasse, quindi può accedervi direttamente. Nella aggregazione/composizione la classe aggregante contiene un campo per ogni classe aggregata e può accedere ai suoi campi esclusivamente tramite i metodi di accesso. L’ereditarietà può consentire il polimorfismo. La scelta di progetto è totalmente diversa: con l’ereditarietà possiamo definire una gerarchia di classi, dimenticandoci in quale delle superclassi sono state definite caratteristiche ereditate.

**POLIMORFISMO**

*Classificazione di Cardelli e Wegner*

POLIMORFISMO UNIVERSALE

Può operare su un numero illimitato di tipi, i morfismi sono generati automaticamente e c’è una base unificante comune a tutti i diversi morfismi che può assumere l’eredità polimorfa.

* Parametrico 🡪 Riguarda funzioni polimorfe, cioè funzioni con un parametro di tipo, dette funzioni generiche, che può lavorare su argomenti di molti tipi esibendo lo stesso comportamento indipendentemente dal tipo dell’argomento. In Ada opera a livello sintattico, perché in fase di compilazione si genera il codice apposta per il tipo dell’argomento specificato. In genere invece, nei linguaggi come Java, questa operazione viene eseguita a runtime.
* Di inclusione -> Si ha polimorfismo di inclusione se un oggetto appartiene ad una classe e a tutte le sue superclassi. Si manifesta in almeno due modi: assegnando un istanza di una sottoclasse ad una variabile del tipo della superclasse, oppure invocando un metodo di una superclasse su una istanza di sottoclasse. È interessante osservare come le invocazioni di metodi di oggetti di classi diverse producono comportamento differente anche se la dichiarazione del metodo è unica. Ciò dipende dal **tipo di legame statico/dinamico** fra identificatore di funzione e realizzazione di funzione. Si parla di legame statico se i binding vengono definiti in fase di compilazione; si parla di legame dinamico se i binding vengono definiti in fase di esecuzione.

POLIMORFISMO AD HOC

Può operare su un numero finito di tipi, i morfismi sono generati manualmente, non c’è una base comune a tutti i morfismi.

* Coercizione 🡪 Conversione implicita operata da un compilatore, per esempio la somma fra un int e un float, o l’autoboxing o l’unboxing di un Integer in un int. La coercizione opera ad un livello semantico, cioè modifica la rappresentazione del dato
* Overloading 🡪 Si usa lo stesso identificatore per metodi differenti e si ricorre ad informazioni di contesto per decidere a quale metodo si sta facendo riferimento. La disambiguazione si basa o sui parametri del metodo o sulla classe dell’oggetto a cui si richiede il servizio. Opera a livello sintattico.