Classi, interfacce e trait

Prima della programmazione ad oggetti, i programmi venivano visti come semplici procedure, o funzioni, che alterano lo stato interno del programma (la memoria).

Il paradigma di programmazione ad oggetti nasce con l'idea di modularizzare lo sviluppo di programmi complessi. Viene quindi partizionato lo stato del programma all'interno di tanti oggetti. Ogni oggetto è responsabile della sua partizione, ed interagisce con altri oggetti scambiandosi messaggi (attraverso metodi).

I linguaggi di programmazione ad oggetti si dividono in due grandi famiglie:

- I linguaggi **object-based**: linguaggi privi della nozione di classe. Gli oggetti vengono creati quando necessario, definendo campi e metodi. E' possibile aggiungere successivamente campi e metodi, non è presente uno schema fisso.

  Il linguaggio object-based più diffuso è **Javascript**.

  Un possibile esempio in pseudo codice può essere:

  ```
  obj = object {
      x = 2
      set(n) = x <- n
      double() = set(2 * x)
  }

  obj.double()
  ```

  con successiva implementazione:

  ```
  obj = struct {
      x = 2
      set = set'              // i metodi diventano campi!
      double = double'
  }

  set'(self, n) = self.x <- n
  double'(self) = self.set(self, 2 * self.x)
  obj.double(obj)
  ```

  Siamo molto vicini alla nozione di *chiusura*. I metodi sono delle funzioni che hanno l'argomento implicito `self` (riferimento alla struttura che contiene le informazioni dell'oggetto) che permette di raggiungere l'oggetto che contiene campi e metodi.
  La differenza è la dinamicità della struttura, dove a run-time possiamo aggiungere o rimuovere elementi.

  I linguaggi object-based presentano però dei limiti:

  – I metodi viaggiano insieme all'oggetto. Comunicando l'oggetto a qualcuno, chi lo riceve può alterarlo come vuole, anche in modi non desiderati.

- Ragionare sulla correttezza del codice è difficile. Riprendendo l'esempio di codice precedente, il metodo `double` funziona correttamente a patto che il metodo `set` assegni il valore di `n` al campo `x`. Modificando (a run-time) il metodo `set` potrei compromettere il funzionamento di `double`.
- Dare un tipo agli oggetti è difficile. Se posso aggiungere/togliere campi/metodi il tipo di un oggetto è difficilmente tracciabile dal compilatore.

- I linguaggi **class-based**: linguaggi dove il comportamento è più disciplinato. Gli oggetti rappresentano un'entità in una famiglia ben definita. Queste famiglie sono classi.

  Alcuni linguaggi class-based più diffusi sono **C++, Java, C#**.

  Un possibile esempio in pseudo codice può essere:

```
class C {
    x = 2
    set(n) = x <- n
    double() = set(2 * x)
}

obj = new C()
obj.double()
```

  L'insieme dei metodi è fissato una volte per tutte. In questo caso è possibile ragionare sulla correttezza di una classe. E' inoltre possibile assegnare un tipo **statico** agli oggetti.

  Una possibile implementazione:

```
struct C {
    vtab : C_virtual_table
    x : int
}

struct C_virtual_table {
    set : int -> void = set'
    double : void -> void = double'
}

set'(self, n) = self.x <- n
double'(self) = self.vtab.set(self, 2 * self.x)
```

  I metodi sono campi di una *virtual table*. La virtual table è unica per tutti gli oggetti istanza di una certa classe e può essere precalcolata.

Ereditarietà e polimorfismo

L'idea alla base dell'**ereditarietà** è la possibilità di **riusare il codice** di una classe per una sottoclasse più specifica. E' possibile aggiungere campi e metodi, e ridefinire (`override`) metodi esistenti.

Riprendendo l'esempio precedente, si vuole estendere la classe `C`:

```
class D extends C {
    y = 3
    get() = y
    double() = super.double(); y <- 2 * y
}
```

con successiva implementazione:

```
struct D {
    vtab : D_virtual_table
    x : int
    y : int
}

struct D_virtual_table {
    set : int -> void = set'
    double : void -> void = double''
    get : void -> int = get'
}

double''(self) = double'(self); self.y <- 2 * self.y
```

La virtual table della classe derivata contiene al suo interno un puntatore alla virtual table della classe base per ereditare tutti i metodi della classe di partenza.

Il meccanismo dell'ereditarietà viene seguito dal concetto di **polimorfismo**. Vediamo un possibile esempio in Java:

```
abstract class Figure {
    private float x, y;
    public abstract float area();
    public abstract float perimeter();
}

class Square extends Figure {
    private float side;
    public float area() { return side * side; }
    public float perimeter() { return 4 * side; }
}

class Circle extends Figure {
    private float radius;
    public float area() { return pi * radius * radius; }
    public float perimeter() { return 2 * pi * radius; }
}

float sum(Figure f, Figure g) { return f.area() + g.area(); }
```

`Square` e `Circle` sono sottoclassi di `Figure`, andando a specializzare la classe base.
Il metodo `sum` permette quindi di sommare l'area di due figure qualsiasi.

Questo approccio però porta una rigidità. Idealmente, `sum` funzionerebbe per tutti gli oggetti che hanno un metodo `area`, ma nel nostro caso `sum` accetta solo figure.

Il problema principale è il definire una gerarchia sul **cosa un oggetto è**, ma non su **cosa sa realmente fare**.

Un ulteriore problema è ...