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 che il meccanismo dell'ereditarietà può diventare troppo "fragile".

Se la classe base viene modificata in un secondo momento, tutte le sottoclassi basate su quella classe padre devono adattarsi alle modifiche.

Si osservi una possibile implementazione di una pila (che si assume sia corretta) in Java:

```
class Stack<T> {
    ...
    void push(T x) { ... }
    T pop() { ... }
    void pushAll(T[] a) { for (T x : a) push(x); }
}
```

Si presenta ora l'esigenza di definire una nuova classe `SizedStack` basata sulla classe `Stack` ma con dimensione esplicita. Una sua possibile implementazione può essere:

```
class SizedStack<T> extends Stack<T> {
    int size = 0;
    void push(T x) { super.push(x); size++; }
    T pop() { size--; return super.pop(); }
}
```

In questo caso non c'è bisogno di ridefinire il metodo `pushAll`.

Si ipotizzi ora la seguente modifica della classe base `Stack`:

```
class Stack<T> {
    ...
    void push(T x) { ... }
    T pop() { ... }
    void pushAll(T[] a) { ... }     // versione che non usa push
}
```

In questo caso occorre ridefinire anche il metodo `pushAll`.
`pushAll`, non usando più il metodo `push`, non permette di non ridefinire il metodo `pushAll` all'interno della classe `SizedStack`.

A causa di queste problematiche (rigidità e conseguente fragilità) l'ereditarietà non risolve sempre i problemi che cerca di risolvere (come il copia-incolla di codice).

Dall'ereditarietà alla composizione

Per cercare di risolvere i problemi dell'ereditarietà si è passati alla composizione di oggetti. Si parla di sviluppare oggetti che al loro interno utilizzano altri oggetti.

Riprendendo l'esempio precedente di `SizedStack`, con composizione al posto di ereditarietà:

```
class SizedStack<T> {
    Stack<T> s;         // composizione
    int size = 0;
    void push(T x) { s.push(x); size++; }
    T pop() { size--; return s.pop(); }
    void pushAll(T[] a) { s.pushAll(a); size += a.length; }
}
```

In questo caso, `SizedStack` contiene uno `Stack`, rendendo le due implementazioni separate e indipendenti.

Ovviamente, anche il meccanismo della composizione non è perfetto. Un possibile problema può essere che i tipi `SizedStack<T>` e `Stack<T>` **non sono più in relazione** nonostante descrivano oggetti che **sanno fare le stesse cose**.

Interfacce

In parallelo all'idea di classe è molto utile avere anche l'idea di **interfaccia**. Una interfaccia definisce **cosa sa fare un determinato oggetto**.

Riprendendo l'esempio precedente di `SizedStack`, come interfaccia:

```
interface StackInterface<T> {
    void push(T x);
    T pop();
    void pushAll(T[] a);
}

class Stack<T> implements StackInterface<T> { ... }

class SizedStack<T> implements StackInterface<T> { ... }
```

Le due classi `Stack<T>` e `SizedStack<T>` implementano l'interfaccia `StackInterface<T>`, definendo i metodi che le due classi devono implementare. Le due nuove classi andranno ad implementare in base alle proprie esigenze i metodi definiti nell'interfaccia.

Trait

Si introduce ora il linguaggio **Go**, che punta ad essere semplice da imparare ed intuitivo. In Go non è presente il concetto di oggetto, ma viene utilizzata l'idea dei *trait*.

Un esempio di codice Go può essere:

```
package main
import (
    "fmt"
    "math"
)

type Vector struct {
    X, Y float64
}

func Mod(v Vector) float64 {
    return math.Sqrt(v.X * v.X + v.Y * v.Y)
}

func main() {
    v := Vector{3, 4}
    fmt.Println(Mod(v))
}
```

In Go il programma viene definito all'interno di un `package`.

Tramite `import` si possono importare librerie (moduli) esterne.
Viene successivamente definita una struttura `Vector`, con due campi di tipo `float`.
La funzione `Mod(v Vector)` calcola il modulo del vettore dato in input, ritornando un `float`.
Infine è presente una funzione `main()`, punto di esecuzione del programma.
Tramite il costrutto `:=` viene creata una variabile locale ed assegnata (nel nostro caso con un `Vector`).

Volendo utilizzare dei **metodi** al posto delle funzioni, avremmo potuto modificare il codice nel seguente modo:

```
package main
import (
    "fmt"
    "math"
)

type Vector struct {
    X, Y float64
}

func (v Vector) Mod() float64 {     // questa definizione è differente
    return math.Sqrt(v.X * v.X + v.Y * v.Y)
}
func main() {
    v := Vector{3, 4}
    fmt.Println(v.Mod())        // viene richiamato in maniera diversa
}
```

Un metodo è una funzione in cui un argomento specifico - detto **receiver** - viene distinto dagli altri.

I metodi vengono invocati con una sintassi `o.m(a1, ..., an)` invece della tradizionale `m(o, a1, ..., an)`. Questa sintassi è simile a quella di Java, anche se non si tratta di programmazione ad oggetti.

Questo viene fatto perché non è presente *late binding*, la sintassi con il receiver a sinistra è quindi scelta per **(f)utili motivi**: aiuta l'autocompletamento perché il tipo è definito localmente.

A run-time non cambia nulla tra l'esecuzione di funzioni e metodi.

Nota Bene: il receiver *deve avere un tipo definito nel file corrente*. Per ovviare a questo limite è possibile utilizzare degli **alias**:

```
package main
import ("fmt")

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

func main() {
    f := MyFloat(-math.Sqrt2)
    fmt.Println(f.Abs())
}
```

In questo caso viene definito un tipo all'interno del file che fa da alias per il `float64`.

In Go esiste anche il concetto di puntatore (riferimento). Un possibile esempio può essere:

```
func (v *Vector) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vector{3, 4}
    v.Scale(10)
    fmt.Println(v.Abs())
}
```

Il receiver può avere un tipo puntatore per consentire la modifica. La sintassi per accedere/modificare campi e per l'invocazione è la stessa. Non c'è bisogno di deferenziare il puntatore, sarà il compilatore a fare tutto il lavoro.

La differenza rispetto ad invocare il metodo senza `*` è che in questo caso la modifica viene effettuata globalmente, mentre senza `*` verrebbe creata una copia locale al quale verrebbe applicata la modifica.

A differenza di Java, dove tutti gli oggetti sono allocati nell'Heap e passando un oggetto ad un metodo viene sempre passato un riferimento di quell'oggetto, in Go se non viene specificato il passaggio come riferimento del dato si va a creare una copia locale del dato stesso.

Si introduce ora il concetto di **trait**, che in Go viene chiamato *interfaccia*.
Un trait è un **insieme di tipi di metodi**. Tutte le entità che implementano l'interfaccia supportano determinate operazioni.

Inoltre, non è necessario utilizzare alcuna cerimonia sintattica (come `implements` in Java). Un tipo `T` implementa **automaticamente** un'interfaccia `I` se definisce tutti i metodi elencati in `I` (con lo stesso tipo). Questo viene chiamato **Duck Typing**.

Si osservi ora un esempio di utilizzo delle interfacce in Go.

Vengono dichiarate due strutture di dato: `Square` e `Circle`. Entrambe le strutture implementano i metodi `Area()` e `Parimeter()`.

```
type Square struct {
    side float64
}

func (o Square) Area() float64 {
    return o.side * o.side
}

func (o Square) Perimeter() float64 {
    return 4 * o.side
}
```

```
type Circle struct {
    radius float64
}

func (o Circle) Area() float64 {
    return pi * o.radius * o.radius
}

func (o Circle) Perimeter() float64 {
    return 2 * pi * o.radius
}
```

Si vuole quindi dichiarare una interfaccia `Measurable`, la quale contiene al suo interno i due metodi precedentemente citati.

```
type Measurable interface {
    Area() float64
    Perimeter() float64
}

func PrintMeasure(o Measurable) {
    fmt.Println(o.Area())
    fmt.Println(o.Perimeter())
}

func main() {
    s := Square{1}
    c := Circle{2}
    PrintMeasure(s)     // Square -> Measurable
    PrintMeasure(c)     // Circle -> Measurable
}
```

`PrintMeasure` è un esempio di funzione polimorfa, al quale possiamo passare un oggetto `Measurable`. Possiamo quindi passare qualsiasi oggetto che implementi quella interfaccia.

Il compilatore Go in maniera automatica capisce se l'interfaccia viene implementata da un tipo, come nel caso di `Square` e `Circle` che implementano l'interfaccia `Measurable`.

Osserviamo ora l'implementazione delle interfacce in Go. Viene ripreso il seguente metodo:

```
func PrintMeasure(o Measurable) {
    fmt.Println(o.Area())
    fmt.Println(o.Perimeter())
}
```

Il compilatore, per tradurre il seguente metodo, deve sapere quali metodi `Area` e `Perimeter` invocare, ma anche l'oggetto su cui agire.

Viene quindi introdotto il concetto di **valore di tipo interfaccia**, che equivale alla **coppia (dati, codice)**.

Un valore di tipo interfaccia `I` è una coppia `(v, p)`, dove `v` è l'oggetto e `p` è la tabella dei metodi in `I` per l'oggetto `v`.

Il punto chiave è che viene fatta una chiamata a funzione il cui indirizzo in memoria è noto solo a tempo di esecuzione (una forma di *late binding*). Solo nel momento d'esecuzione viene letta la tabella presente nella coppia, e si scopre il metodo giusto da invocare.

Si vuole invece osservare cosa accade nel `main` dell'esempio precedente:

```
func main() {
    s := Square{1}
    c := Circle{2}
    PrintMeasure(s)
    PrintMeasure(c)
}
```

Passare `s` di tipo `Square` a una funzione che si aspetta `Measurable` significa creare e passare la coppia `(s, p)` dove `p` è una tabella (simile a una virtual table) che contiene puntatori ai metodi `Area` e `Perimeter` per i receiver di tipo `Square`. Il discorso è analogo per `c`.

In Go è possibile anche definire interfacce che estendono altre interfacce.

```
type MeasurableColorable interface {
    Measurable
    GetColor() color
    SetColor(c color)
}
```

In questo modo si afferma che `MeasurableColorable` possiede tutti i metodi presenti in `Measurable`, oltre ai nuovi metodi.

Questo tipo di ereditarietà è solo a livello di specifica dell'interfaccia, non di codice.

E' possibile in maniera analoga realizzare composizione di strutture in Go.

```
type ColoredSquare struct {
    Square
    c Color
}
```

Una struttura può avere come campo (anonimo) un'altra struttura. Tutti i campi della struttura inclusa sono accessibili come campi della struttura che la include.
I metodi già definiti per la struttura inclusa (es. `Area` e `Perimeter`) sono generati anche per la struttura che la include.

Volendo quindi mettere a confronto le chiusure ed i trait:

- le *chiusure* definiscono **una funzione** che può accedere a **diversi dati**,
- i *trait* definiscono **un dato** per il quale disponiamo di **diverse funzioni**.

Rust

Rust è un linguaggio di sistema (come *C o C++*) con un supporto a run-time minimo (ad esempio, non è presente alcun garbage collector) e con un sistema di tipi con gestione lineare delle risorse (per gestione lineare si intende che il sistema di tipi è in grado di capire se una certa entità del programma viene duplicata).

In Rust, a differenza di Go, i *trait* richiedono una sintassi esplicita per la loro implementazione. E' inoltre richiesta una sintassi esplicita anche per il passaggio di oggetti dove è atteso un trait.

In generale, un *metodo* equivale ad una *funzione con invocazione su receiver*.

Si osservi ora un esempio di funzioni e metodi in Rust:

```
struct Vector {
    x : f64,
    y : f64,
}

impl Vector {
    // non-instance method
    fn origin() -> Vector {
        Vector { x : 0.0, y : 0.0 }
    }

    // instance method
    fn modulus(&self) -> f64 {
        (self.x * self.x + self.y * self.y).sqrt()
    }
}

fn mod_origin() -> f64 {
    let o = Vector::origin()        // non-instance method invocation
    o.modulus()                     // instance method invocation
}
```

Viene inizialmente dichiarata una struttura `Vector`.
La struttura implementa, tramite il costrutto `impl`, i metodi `origin` e `modulus`. Il primo è una semplice funzione, senza receiver. Il secondo è un metodo d'istanza, ha un receiver.
`origin` può essere visto come un costruttore (viene chiamato origin arbitrariamente, senza una logica precisa).
`modulus` è un metodo d'istanza. Il `&self` indica che il metodo prende temporaneamente in prestito il possesso dell'oggetto su cui agisce.
Nella funzione `mod_origin` vengono richiamati i due metodi precedentemente citati. Inoltre, non è necessario effettuare un `return` esplicito alla fine della funzione, ma verrà restituito l'ultimo valore presente.

Per definire un trait in Rust viene utilizzata la seguente sintassi:

```
trait Measurable {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
```

A differenza di Go, viene indicato esplicitamente il receiver (`&self`).
L'uso di `self` come *tipo segnaposto* per l'argomento dell'oggetto receiver è fondamentale perché qui **non sappiamo** che tipo avrà.

Si osservi ora un esempio di implementazione del trait precedente in Rust:

```
struct Circle {
    radius : f64,
}

impl Measurable for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * self.radius * self.radius
    }

    fn perimeter(&self) -> f64 {
        2 * std::f64::consts::PI * self.radius
    }
}
```

Rispetto a Go, il sistema di tipi di Rust è più sofisticato.
Sono, ad esempio, presenti le funzioni polimorfe con vincoli:

```
fn print_measure<T : Measurable>(shape : T) {
    println!("Area = {}", shape.area());
    println!("Perimeter = {}", shape.perimeter());
}
```

La funzione `print_measure` è polimorfa nel tipo `T` dell'argomento.
Il trait `Measurable` è usato come vincolo per `T`. Analogo ad `extends` di Java.

Anche i tipi possono essere polimorfi con vincoli:

```
struct Rectangle<T> {
    width : T,
    height : T,
}

impl Rectangle<T : PartialEq + Display> Rectangle<T> {
    fn is_square(&self) -> bool {
        println!("Width = {}", self.width);
        self.width == self.height
    }
}
```

Il tipo `Rectangle` è polimorfo nel tipo delle lunghezze. I trait `PartialEq` e `Display` sono vincoli per `T`.
`self.width` in `println!` richiede che `T` implementi `Display`, per stampare a schermo.
`==` richiede che `T` implementi `PartialEq`, per effettuare il confronto.

Per implementare in Rust il trait `T` per il tipo `U` almeno uno tra `T` e `U` deve essere definito nel file corrente, a differenza di Go dove si possono implementare metodi per `U` solo nel file dove è stato definito (questo a volte porta all'utilizzo di tipi alias).

Questo permette di coprire due casi comuni:

- Ho definito un nuovo tipo `U` -> implemento tutti i trait `T` con cui `U` è compatibile.
- Ho definito un nuovo trait `T` -> implemento `T` per tutti i tipi `U` compatibili con `T`.

In Rust i trait possono fornire implementazioni di default. Queste implementazioni possono essere ridefinite quando si implementa il trait. Un possibile esempio può essere:

```
trait Foo {
    fn is_valid(&self) -> bool { !self.is_invalid() }
    fn is_invalid(&self) -> bool { !self.is_valid() }
}
```

In questo caso si può implementare uno dei due metodi, ottenendo l'opposto tramite l'implementazione di default.

Questo approccio ha alcuni potenziali problemi:

- *Cut & paste* di codice, analogie con la fragilità dell'ereditarietà.
- Se la stessa implementazione di default è fornita per lo stesso metodo in trait diversi, non è chiaro quale debba essere usata (*diamond problem*).

Anche in Rust, come in Go, è presente l'ereditarietà di trait:

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foo_bar(&self);
}
```

`FooBar` estende il trait `Foo`, includendo tutti i metodi definiti anche in `Foo`.

Binding statico in Rust
Un metodo invocato su un oggetto il cui tipo è *concreto* (non un trait) viene risolto **staticamente**:

```
fn smaller(a : Circle, b : Circle) -> bool {
    a.area() < b.area()
}
```

Un metodo invocato su un oggetto il cui tipo è *generico* (ma concreto) viene risolto staticamente grazie alla **monomorfizzazione** (definita una funzione polimorfa, il compilatore genera una versione per ogni applicazione della funzione):

```
fn smaller<T>(a : Rectangle<T>, b : Rectangle<T>) -> bool {
    a.width * a.height < b.width * b.height
}
```

Vengono create (a tempo di compilazione) tante versioni di `smaller`, una per ogni istanziazione del parametro di tipo `T`.

Binding dinamico in Rust
Un metodo invocato su un oggetto il cui tipo è un (riferimento a) trait viene risolto **dinamicamente**:

```
fn print_measure(shape : &Measurable) {
    println!("Area = {}", shape.area());
    println!("Perimeter = {}", shape.perimeter());
}
```

Il passaggio di un oggetto di tipo (riferimento a) trait deve essere indicato esplicitamente dal programmatore.

```
fn main() {
    let c = Circle{radius : 2};
    print_measure(&c as &Measurable);
}
```

L'implementazione è analoga a quella di Go, con coppie (valore + puntatore a tabella dei metodi).