

Sviluppo



Di Michele Costabile



Dallo sviluppo Object Oriented alla programmazione Protocol Oriented.

La rivoluzione copernicana dello sviluppo Apple inizia da Swift.

Il protocollo di Cupertino

Dave Abrahams è un brillante programmatore. Fa parte del comitato di standardizzazione del C++, ha scritto un libro sul *template metaprogramming* e ha fondato e animato una comunità che pubblica Boost, una raccolta di librerie applicative per il C++, distribuite con licenza open source e con un livello di qualità generale garantito dal passaggio di numerosi occhi qualificati sulle righe di codice. Boost è un'estensione della libreria standard del C++ e alcune librerie che ne fanno parte finiranno probabilmente nella Standard Template Library (Stl) in una prossima release del C++, anche perché molti dei collaboratori sono membri del comitato di standardizzazione, per questo ha avuto un notevole successo e la conferenza annuale, BoostCon raccoglie molti dei migliori

programmatore. Dopo essersi fatto notare con questo lavoro, dal 2013 Abrahams è stato assunto da Apple per lavorare al linguaggio Swift e attualmente è technical lead della libreria applicativa per il linguaggio Swift. In questa qualità, ha mantenuto fede al suo titolo autoimposto di *Professor of blowing your*

mind (professore di stupore) con una conferenza all'evento per sviluppatori Apple, il Wwdc, che ha scatenato l'immaginazione dei presenti e un grande entusiasmo nella comunità Swift. Durante la conferenza, dal titolo *Protocol oriented programming*, Abrahams ha presentato una piccola estensione al

ASCIWwdc

Search for WWDC Sessions...

Protocol-Oriented Programming in Swift



Session 408

WWDC 2015

At the heart of Swift's design are two incredibly powerful ideas: protocol-oriented programming and first class value semantics. Each of these concepts benefit predictability, performance, and productivity, but together they can change the way we think about programming. Find out how you can apply these ideas to improve the code you write.

[Music]

[Silence]

[Applause]

Dave Abrahams: Hi, everybody.

My name is Dave Abrahams, and I'm the technical lead for the Swift standard library, and it is truly my privilege to be with you here today.

It is great to see all of you in this room.

The next 40 minutes are about putting aside your usual way of thinking about programming.

È disponibile una trascrizione della conferenza, che può aiutare a seguirla. L'indirizzo è asciwwdc.com/2015/sessions/408

Dave Abrahams
durante la conferenza
alla WWDC 2015.

LISTATO 1

```
/// Instances of conforming
types can be compared for value
equality
/// using operators `==` and
`!=`.
///
/// When adopting `Equatable`,
only the `==` operator is
required to be
/// implemented. The
standard library provides an
implementation for `!=`.
protocol Equatable {

    /// Return true if `lhs` is
equal to `rhs`.
    ///
    /// **Equality implies
substitutability**. When `x ==
y`, `x` and
    /// `y` are interchangeable
in any code that only depends on
their
    /// values.
    ///
    /// Class instance identity
as distinguished by triple-
equals `===`
    /// is notably not part of
an instance's value. Exposing
other
    /// non-value aspects
of `Equatable` types is
discouraged, and any
    /// that are exposed
should be explicitly pointed out
in
    /// documentation.
    ///
    /// **Equality is an
equivalence relation**
    ///
    /// - `x == x` is `true`
    /// - `x == y` implies `y
== x`
    /// - `x == y` and `y == z`
implies `x == z`
    ///
    /// **Inequality is the
inverse of equality**, i.e. `!(x
== y)` iff
    /// `x != y`.
    func ==(lhs: Self, rhs:
Self) -> Bool
}
```

Protocol-Oriented Programming in Swift

At the heart of Swift's design are two incredibly powerful ideas: protocol-oriented programming and first class value semantics. Each of these concepts benefit predictability, performance, and productivity, but together they can change the way we think about programming. Find out how you can apply these ideas to improve the code you write.



linguaggio Swift 2.0, che apre le porte a un paradigma di programmazione del tutto nuovo, appunto chiamato *Protocol Oriented Programming*, che promette prestazioni migliori, minore rigidità e una flessibilità sorprendente nella costruzione di funzioni applicative plasmabili e personalizzabili in modo insolito.

PROTASI E PROTOCOLLO

I protocolli, sono un costrutto di programmazione che arriva da lontano: sono presenti in Objective-C fin dai tempi in cui Steve Jobs lavorava alla creazione di NeXT, dopo essere uscito da Apple. Gli ingegneri di NeXT li hanno aggiunti al linguaggio per sfuggire alle limitazioni dell'ereditarietà singola, che affliggono la maggior parte dei linguaggi, con qualche notevole eccezione, come il C++, Perl e Python. Ecco un semplice esempio di dichiarazione di un protocollo

```
@protocol NSLocking
- (void)lock;
- (void)unlock;
@end
```

Come si vede, un protocollo indica che un oggetto di tipo NSLocking deve

avere un metodo *lock* e un metodo *unlock*, in altri termini si tratta di quello che in altri linguaggi si chiama un'interfaccia. Quindi, anche Swift dedica più attenzione alle interfacce, un fenomeno che avevamo già visto nel più recente linguaggio Go creato da Griesemer, Pike, e Thompson, l'autore di Unix. Go non ha costrutti per creare classi e gestire ereditarietà, ma offre un completo supporto per la creazione di interfacce. Ecco un esempio di protocollo, preso dalla libreria standard di Swift, con tutti i commenti originali (**Listato 1**).

Il protocollo Equatable definisce un'operazione elementare: il confronto per uguaglianza (==) fra istanze dello stesso tipo. Come si dice nei commenti, i tipi che decidono di conformarsi a questo protocollo devono limitarsi a definire l'uguaglianza. La relazione contraria viene realizzata nella libreria standard.

Si noti la presenza di *Self* nelle dichiarazioni di tipo, che sta a indicare che se un tipo X si conforma al protocollo, la funzione di uguaglianza sarà:

```
func ==(lhs:X, rhs:X) -> Bool
```

Self, quindi, sta per "il tipo che dichiarerà di adeguarsi a questo protocollo".

LISTATO 2

```

/// A value type whose instances are
either `true` or `false`.
struct Bool {

    /// Default-initialize Boolean
    value to `false`.
    init()
}

extension Bool :
BooleanLiteralConvertible {
    init(_builtinBooleanLiteral
value: Builtin.Int1)

    /// Create an instance initialized
to `value`.
    init(booleanLiteral value: Bool)
}

extension Bool : BooleanType {

    /// Identical to `self`.
    var boolValue: Bool { get }

    /// Construct an instance
representing the same logical value
as
    /// `value`.
    init<T : BooleanType>(_ value: T)
}

extension Bool :
CustomStringConvertible {

    /// A textual representation of
`self`.
    var description: String { get }
}

extension Bool : Equatable, Hashable
{

    /// The hash value.
    ///
    /// **Axiom:** `x == y` implies
`x.hashValue == y.hashValue`.
    ///
    /// - Note: the hash value is not
guaranteed to be stable across
    /// different invocations of
the same program. Do not persist the
    /// hash value across program
runs.
    var hashValue: Int { get }
}

extension Bool : _Reflectable {
}

```

PROTOCOLLI IN AZIONE

Se guardiamo la libreria di Swift, scopriamo inaspettatamente molte meno classi di quante sembrerebbe logico. Troviamo, invece, dichiarazioni di tipo e estensioni di protocollo in abbondanza. Per esempio, il tipo Bool, un tipo piuttosto di base nel linguaggio, realizza ben cinque protocolli: BooleanLiteralConvertible, BooleanType, CustomStringConvertible, Equatable e Hashable.

La dichiarazione completa del tipo Bool, che nasce come struttura e si arricchisce di estensioni di protocollo è leggibile nel **Listato 2**.

Come si vede, l'estensione del protocollo *BooleanType* è quella che assicura a Bool di contenere una variabile di tipo boolValue.

DOVE STA IL BELLO?

Abbiamo visto nel paragrafo precedente come si possono estendere i tipi base del linguaggio, aggiungendo protocolli a strutture e tipi base. Questa è una caratteristica della versione 2.0 di Swift, che ha effetti profondi sul

linguaggio, soprattutto per chi crea librerie.

Ecco due righe di codice che probabilmente possono scatenare uno di quei "momenti Oh" negli sviluppatori:

```

extension IntegerArithmeticType
{
    func double() -> Self {
        return self + self
    }
}

2.double() // 4
-2.double() // -4

```

Questa è una cosa a cui siamo probabilmente abituati in JavaScript, ma è molto meno familiare in linguaggi strongly typed. Il trucco sta nel fatto che 2 è una costante intera, quindi si adegua al protocollo IntegerArithmeticType. Estendendo questo protocollo, qualunque intero guadagna un metodo double, che restituisce il doppio del suo valore.

Se copiamo questo codice in un playground di XCode 6.4 abbiamo l'errore "Protocol IntegerArithmeticType cannot be extended", mentre il codice è legittimo in XCode 7 e produce i risultati attesi.

Crustacean
Protocol-Oriented Programming with Value Types

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology. The license for this document is available [here](#).

```

import CoreGraphics
let twoPi = CGFloat(M_PI * 2)

```

A protocol for types that respond to primitive graphics commands. We start with the basics:

```

protocol Renderer {
    /// Moves the pen to `position` without drawing anything.
    func moveTo(position: CGPoint)

    /// Draws a line from the pen's current position to `position`, updating
    /// the pen position.
    func lineTo(position: CGPoint)

    /// Draws the fragment of the circle centered at `c` having the given
    /// `radius`, that lies between `startAngle` and `endAngle`, measured in
    /// radians.
    func arcAt(center: CGPoint, radius: CGFloat, startAngle: CGFloat, endAngle: CGFloat)
}

```

A Renderer that prints to the console.

Printing the drawing commands comes in handy for debugging; you can't always see everything by looking at graphics. For an example, see the "nested diagram" section below.

```

struct TestRenderer : Renderer {

```

Il codice presentato nella conferenza è disponibile in un playground e si può scaricare cliccando su un link nella pagina del video della conferenza

David Owens, autore del blog *owensd.io*, mostra un esempio in cui si può sfruttare questa possibilità del linguaggio per semplificare il codice applicativo.

Owens mostra un punto chiave in un frammento del codice nell'inizializzazione di Handmade Hero (*handmadehero.org*), un gioco didattico di cui non solo sono disponibili i sorgenti, ma anche i video che mostrano l'autore durante la codifica e le ragioni di ogni riga di codice.

Nell'inizializzazione c'è un frammento come quello riportato nel **Listato 3**.

Una lunga fila di righe di codice molto simili, che hanno lo scopo di inizializzare diversi elementi di menu e il titolo della finestra con il nome dell'applicazione. Naturalmente, queste righe potrebbero facilmente essere sostituite da un ciclo di inizializzazione come questo.

```
for (var item) in items {
    item.title = item.title.
    replace(placeholder, withString:
    title)
}
```

Il problema sta nel fatto che alcuni oggetti sono di tipo *NSMenuItem*, mentre la finestra è di tipo *NSWindow* entrambi i tipi hanno una proprietà *title*, ma non hanno un antenato comune con questa proprietà.

Una soluzione possibile sarebbe gestire il titolo della finestra separatamente dal titolo dei menu. Una scelta più purista è creare un'estensione di protocollo per trattare il titolo in modo uniforme e usare un loop pulito, come segue:

```
protocol HasTitle {
    var ht_title: String { get set }
}
```

Ora applichiamo un'estensione a *NSMenuItem* e *NSWindow* per gestire questo protocollo, con l'implementazione appropriata per ogni tipo

```
extension NSMenuItem: HasTitle {
    var ht_title: String {
        get { return self.title }
        set { self.title = newValue }
    }
}
```

LISTATO 3

```
self.aboutMenuItem.title = [self.aboutMenuItem.title
    stringByReplacingOccurrencesOfString:DEADZONE_TITLE
    withString:APP_NAME];
self.hideMenuItem.title = [self.hideMenuItem.title
    stringByReplacingOccurrencesOfString:DEADZONE_TITLE
    withString:APP_NAME];
self.quitMenuItem.title = [self.quitMenuItem.title
    stringByReplacingOccurrencesOfString:DEADZONE_TITLE
    withString:APP_NAME];
self.helpMenuItem.title = [self.helpMenuItem.title
    stringByReplacingOccurrencesOfString:DEADZONE_TITLE
    withString:APP_NAME];
self.window.title = [self.window.title
    stringByReplacingOccurrencesOfString:DEADZONE_TITLE
    withString:APP_NAME];
```

```
extension NSWindow : HasTitle {
    var ht_title: String {
        get { return self.title ??
        "" }
        set { self.title = newValue }
    }
}
```

Con queste premesse, il loop di inizializzazione diventa lineare e espressivo come qualunque programmatore si aspetta.

```
let items: [HasTitle] = [aboutItem,
    hideItem, quitItem, helpItem,
    window]
for (var item) in items {
    item.ht_title = item.ht_title.
    replace(placeholder, withString:
    title)
}
```

Si noti che *replace* è un'estensione di protocollo applicata alle stringhe per gestire la sostituzione in modo più semplice.

SI AVVICINA LA FINE DELL'OBJECT ORIENTED?

Il fatto che fino a questo punto non sia stata nominata una classe e non sia stata mostrata una catena di derivazione dovrebbe avere destato la curiosità, o la preoccupazione, del lettore.

Il punto è proprio questo: le classi stanno pian piano evaporando dalla libreria di Swift, per lasciare il posto a strutture e protocolli che, come abbiamo visto, possono contenere codice e aggiungere comportamento, oltre che dati al tipo di partenza.

Ma, naturalmente, la domanda è: in cosa questo paradigma di programmazione è migliore di quello object oriented, che abbiamo conosciuto per decenni.

TRE RISPOSTE DI ABRAHAMS A QUESTA DOMANDA

Il primo problema che si incontra in complesse gerarchie di oggetti è la condivisione di stato fra oggetti diversi. Può capitare che due oggetti abbiano un riferimento alla stessa istanza di un terzo oggetto e può succedere che uno dei due modifichi lo stato condiviso compromettendo l'altro. Risolvere il problema è complesso, si può ricorrere

“
Nella libreria di Swift
le classi potrebbero
gradualmente lasciare
il posto a strutture
e protocolli.

a copie private, ma questo aumenta il peso in termini di memoria e tempo di elaborazione. Si può anche incorrere in race conditions, quando thread diversi tentano di accedere allo stesso stato. Per evitarle si possono creare dei lock rallentando il codice, perché, naturalmente, ogni lock è la fine del parallelismo di esecuzione sulle moderne macchine multicore. Insomma, quando si presentano questi problemi è facile cacciarsi in un ginepraio, per di più finendo per combattere il sistema che si usa e facendosi del male in termini di performance.

Il secondo problema, sta nell'ereditarietà singola. Se ogni classe può ereditare da un singolo antenato, è fatale che le proprietà potenzialmente più utilizzate salgano il più possibile nella gerarchia di derivazione. Questo comporta un aggravamento dell'occupazione di memoria e l'appesantimento dell'inizializzazione. Infine, le classi derivate devono badare a non compromettere lo stato ereditato, in un modo che spesso è lasciato alla documentazione e non alla semantica del linguaggio.

Un esempio tipico del problema, lo abbiamo visto nel workaround all'inizializzazione di un titolo fra classi imparentate alla lontana, che abbiamo presentato qualche paragrafo più sopra. Spostare in un antenato comune a *NSWindow* e *NSMenuItem* la proprietà *title*, nell'esempio di prima, non avrebbe senso, dato che l'antenato in comune è troppo astratto e aggiungere una proprietà a una classe in alto nella gerarchia, comporta un aggravio di memoria e una responsabilità di inizializzazione per tutte le classi derivate.

Il terzo problema sta nel fare affidamento a corrette inizializzazioni e override da parte di classi derivate, come nel frammento di codice che segue

```
class Ordered {
    func precedes(other:
Ordered) -> Bool {
fatalError("implement me!")
}

class Label : Ordered { var
text: String = "" }

class Number : Ordered {
    var value: Double = 0
    override func
precedes(other: Ordered) -> Bool
{
        return value < (other
```

GitHub Gist: Search... All gists GitHub

rbobbins / protocols.md

Notes from "Protocol-Oriented Programming in Swift"

protocols.md

PS: If you liked this talk or like this concept, let's chat about iOS development at Stitch Fix! #shamelessplug

Protocol-Oriented Programming in Swift

Speaker: David Abrahams. (Tech lead for Swift standard library)

- "Crusty" is an old-school programmer who doesn't trust IDE's, debuggers, programming fads. He's cynical, grumpy.
- OOP has been around since the 1970's. It's not actually new.
- Classes are Awesome
 - Encapsulation

Un Gist dal titolo *Protocol Oriented Programming*, mostra un riassunto della conferenza e gli esempi di codice (gist.github.com/rbobbins). Per i test è necessaria l'ultima versione di XCode.

```
as! Number}.value
}
}
```

La classe *Ordered* specifica l'esistenza di un metodo *precedes*, ma non può fare altro che demandare la realizzazione a classi derivate.

Con queste premesse, è facile convincersi che il codice che segue genera un'eccezione, perché la classe *Label* dimentica di gestire l'override del metodo *precedes*.

```
let l = Label()
l.text = "ciao"
let n = Number()
n.value = 3

n.precedes(l)
```

CONCLUSIONI

La presentazione di Abrahams è davvero spettacolare e merita di essere vista (developer.apple.com/videos/wwdc/2015/?id=408), eventualmente con l'ausilio della trascrizione (asciwwdc.com/2015/sessions/408).

Certamente, si può obiettare che i problemi che vengono presentati hanno alcune soluzioni nei più comuni linguaggi object oriented, fra cui probabilmente sventa il C++, che vanta l'ereditarietà multipla.

Al netto di tutti i reality check e del

giusto scetticismo con cui vanno valutate le novità, il paradigma che viene presentato è semplice, funzionale, espressivo, facile da capire e da estendere e risolve il problema di creare un corpo di codice coerente esteso come la libreria standard di un linguaggio senza incorrere nei problemi citati. Se possiamo trovargli un difetto, il problema può essere che si può finire per avere a che fare con codice che aggiunge protocolli arbitrari a stringhe e interi, per perseguire qualche intento estetico, come avviene per Prototype, una libreria JavaScript che fa somigliare il linguaggio a Ruby.

Guardando il codice di sistema, comunque, si intravede chiaramente che questa è la strada su cui Apple sta andando. Questo non significa che bisogna smettere di pensare o usare classi e oggetti. Il consiglio che lo stesso Abrahams dà ai programmatori è "don't fight the system". Alcune librerie di sistema, come Cocoa, richiedono chiaramente la manipolazione di oggetti, il nostro codice non ha ragione di non adeguarsi per il tentativo di espungere dal proprio codice classi e istanze. Quando scriviamo codice ex novo, però, la potenza del sistema dei tipi del linguaggio e dei protocolli estendibili, ci mette in mano costrutti di grande potenza e duttilità, che il programmatore ha senz'altro vantaggio a utilizzare.