UNIVERSITÀ
DI PARMA

Dipartimento di Scienze Matematiche,
Fisiche e Informatiche

Corso di Laurea in Informatica

Tesi di Laurea

Applicazioni iOS: analisi dei linguaggi di programmazione utilizzati e sviluppi futuri della piattaforma

Candidato:
Federico Bertoli

Relatore:

Prof. Roberto Alfieri

Anno Accademico 2015/2016

Indice

TODO

Introduzione

Lo sviluppo di applicazioni in ambito mobile ha subito una crescita esponenziale da quando, nel 2008, Apple ha permesso di accedere agli strumenti di sviluppo interni (nello specifico il software Xcode e i relativi SDK) e ha inserito nella versione 2.0 del software iOS l'applicazione App Store, permettendo di fatto a chiunque di rendere la propria idea una realtà, supportata da un'infrastruttura di servizi ben collaudata, concentrando i propri sforzi solamente sullo sviluppo del software.

Nello stesso anno anche Google, tramite l'Android Market ha permesso la pubblicazione di applicazioni sul proprio store, creando così una "corsa all'oro" che solamente ora, 8 anni dopo, sta subendo un sensibile calo dopo anni di crescita significativa e costante.

Precedentemente a queste due piattaforme lo sviluppo in ambito mobile era frenato da fattori importanti quali la mancanza di software di sviluppo stabili e continuamente aggiornati, la mancanza di sistemi operativi sufficientemente avanzati e la scarsa potenza dei dispositivi dell'epoca.

Ciò che ha portato a questa tesi è stato un interesse molto forte verso questo mondo con nemmeno una decade sulle spalle (se si parla di smartphone) e ancora pieno di sviluppi, la maggior parte di questi solo annunciati e ancora in piena fase di progettazione (si pensi per esempio alla realtà aumentata, alle features appena annunciate sui nuovi dispositivi come le doppie fotocamere, i microfoni HAAC o i sensori per il touchscreen in 3 dimensioni).

In questa tesi abbiamo focalizzato l'attenzione sull'ecosistema creato da Apple per i propri dispositivi iOS ed in particolare sui linguaggi utilizzati per lo sviluppo, Objective-C e Swift.

Capitolo 1

Objective-C

1.1 Cenni storici

Nei primi anni ottanta, la pratica comune dell'ingegneria del software era basata sulla programmazione strutturata. Questa modalità era stata sviluppata per poter suddividere programmi di grandi dimensioni in parti più piccole, principalmente per facilitare il lavoro di sviluppo e di manutenzione del software. Ciononostante, col crescere della dimensione dei problemi da risolvere, la programmazione strutturata divenne sempre meno utile, dato che conduceva alla stesura di un numero sempre maggiore di procedure, ad uno spaghetti code e ad uno scarso riuso del codice sorgente.

Venne ipotizzato poi che la programmazione orientata agli oggetti potesse essere una potenziale soluzione al problema. In effetti Smalltalk aveva già affrontato molte di queste questioni ingegneristiche, pur con lo svantaggio di necessitare di una macchina virtuale che interpretava un oggetto in memoria chiamato immagine contenente tutti gli strumenti necessari. L'immagine Smalltalk era molto grossa, usava tendenzialmente un'enorme quantità di memoria per l'epoca e girava molto lentamente anche per la mancanza di un supporto specifico dell'hardware alle macchine virtuali.

Objective C fu creato principalmente da Brad Cox e Tom Love all'inizio degli anni ottanta alla Stepstone. Entrambi erano stati introdotti a Smalltalk durante la loro permanenza al Programming Technology Center della ITT Corporation nel 1981. Cox aveva iniziato ad interessarsi ai problemi legati alla riusabilità del software e si accorse che un linguaggio simile a Smalltalk sarebbe stato estremamente valido per costruire potenti ambiente di sviluppo per i progettisti di ITT.

Cox iniziò così a modificare il compilatore C per aggiungere alcune delle caratteristiche di Smalltalk. Ottenne così ben presto una implementazione funzionante di una estensione ad oggetti del linguaggio C che chiamò OOPC (Object-Oriented Programming in C). Nel frattempo Love fu assunto da Schlumberger Research nel 1982 ed ebbe l'opportunità di acquisire la prima copia commerciale di Smalltalk-80 che influenzò in seguito lo sviluppo della loro creatura.

Per dimostrare che il linguaggio costituiva un reale progresso, Cox mostrò che per realizzare componenti software intercambiabili erano necessari pochi adattamenti pratici agli strumenti già esistenti. Nello specifico, era necessario supportare gli oggetti in modo flessibile con un insieme di librerie software che fossero usabili e consentissero al codice sorgente (e ad ogni risorsa necessaria al codice) di essere raccolto in un solo formato multipiattaforma.
Cox e Love formarono infine una nuova impresa, la Productivity Products International (PPI), per commercializzare il loro prodotto che accoppiava un compilatore Objective C con una potente classe di librerie.

Nel 1986 Cox pubblicò la sua descrizione dell'Objective C nella sua forma originale nel libro Object-Oriented Programming, An Evolutionary Approach. Sebbene fosse attento a puntualizzare che la questione della riusabilità del software non poteva essere esaurita dal linguaggio di programmazione, l'Objective C si trovò spesso ad essere confrontato, caratteristica per caratteristica, con gli altri linguaggi.

Nel 1988, NeXT, la compagnia fondata da Steve Jobs dopo Apple, ottenne la licenza dell'Objective C da Stepstone (allora proprietaria del marchio) e realizzò il proprio compilatore Objective C e le librerie sulle quali basò l'interfaccia utente di NeXTSTEP. Sebbene le workstation NeXTSTEP non riuscissero ad avere un forte impatto sul mercato, i loro strumenti vennero ampiamente apprezzati dall'industria del settore. Ciò portò NeXT ad abbandonare la produzione di hardware ed a focalizzarsi sugli strumenti software, vendendo NeXTSTEP (e OpenStep) come piattaforma per la programmazione.

In seguito il progetto GNU iniziò a lavorare sul clone libero che chiamò GNUstep, basato sullo standard OpenStep. Dennis Glatting scrisse il primo run-time gnu-objc nel 1992 e Richard Stallman lo seguì subito dopo con un secondo. Il run-time GNU Objective C, che è usato dal 1993, è stato sviluppato da Kresten Krab Thorup mentre era studente universitario in Danimarca.

1.2 Caratteristiche del linguaggio

Questo capitolo descrive tutte le funzionalità che il linguaggio aggiunge allo standard C.

1.2.1 Sintassi

Implementazione di una classe, i file .h e .m

L'implementazione di una classe avviene attraverso due file distinti:

- Un file con estensione .h, che conterrà la dichiarazione dell'interfaccia di classe, i metodi e le properties pubbliche.
- Un file con estensione .m, che conterrà l'implementazione della classe, la definizione di metodi e properties private; in questo file inoltre saranno definiti i metodi e le variabili d'istanza private.

Esempio di file header (.h) di una classe:

```
#import <Foundation/Foundation.h>

@interface Persona: NSObject ()

//costruttore personalizzato con parametri

- (id)initConNome:(NSString *)unNome
      cognome:(NSString *)unCognome;

// dichiarazione delle properties private

@private
        NSString *_nome;
        NSString *_cognome;
        int _eta;
}

// dichiarazione dei metodi getter e setter

- (NSString *)nome;
- (void)setNome:(NSString *)nome;
- (NSString *)cognome;
- (void)setCognome:(NSString *)cognome;
- (int)eta;
- (void)setEta:(int)eta;
```

```
@end
```

Esempio di file di implementazione (.m) di una classe:

```
#import "Persona.h"

@implementation Persona

//implementazione della classe dichiarata in Persona.h

- (id)initConNome:(NSString *)unNome
      cognome:(NSString *)unCognome {

    if( self = [super init] )
    {
        _nome = unNome;
        _cognome = unCognome;
    }

    return self;
}

-(NSString *) nome {
        return _nome
}

-(NSString *) cognome {
        return _cognome
}

-(int) eta {
        return _eta
}

-(void)setNome:(NSString *)nome {
        _nome = nome;
}

-(void)setCognome:(NSString *)cognome {
        _cognome = cognome;
}

-(void)setEta:(int)eta {
        _eta = eta;
}
@end
```

Attraverso la parola chiave @property è anche possibile lasciare la definizione dei setter e dei getter al compilatore:

```
//come dichiarati in Persona.h
- (NSString *)nome;
- (void)setNome:(NSString *)nome;

//con property diventa
@property NSString * nome;
```

Inoltre, attraverso la parola chiave @synthesize, è possibile lasciare al compilatore anche il compito di implementare i setter e i getter di una property che sia stata definita precedentemente:

```
//come dichiarati in Persona.m

-(NSString *) nome {
        return _nome
}

-(void)setNome:(NSString *)nome {
        _nome = nome;
}
//con synthesize diventa:

@synthesize nome = _nome;
```

Le properties permettono di accedere ai getter e ai setter utilizzando la notazione puntata, per maggiore leggibilità del codice:

```
//senza properties:

int eta = [persona eta];
[persona setEta:22];

//con properties:

int eta = persona.eta;
persona.eta = 22;
```

Dichiarazione e definizione dei metodi

Come visto per la definizione dei setter, una dichiarazione di funzione (metodo) ha la seguente sintassi:

```
//Nell'ordine: (tipoDiRitorno) nomeMetodo:(tipoArg1)nomeArg1;

- (NSInteger)calcolaEta:(NSDate)dataDiNascita;
```

Definizione del metodo appena dichiarato:

```
- (NSInteger)calcolaEta:(NSDate)dataDiNascita {

        NSDate* oggi = [NSDate date];
        NSDateComponents* componentiCalendario = [[NSCalendar currentCalendar]
                                                  components:NSCalendarUnitYear
                                                  fromDate:dataDiNascita
                                                  toDate:oggi
                                                  options :0];

        NSInteger eta = [componentiCalendario year];

        return eta;
}
```

Invio dei messaggi

In Objective-C, a differenza della maggior parte degli altri linguaggi, non si chiama direttamente un metodo, ma si "passa un messaggio" all'oggetto stesso:

```
[persona calcolaEta:dataDiNascita];
```

Questo perchè il runtime del linguaggio mantiene traccia di tutti i metodi e delle funzioni che conosce; Ogni componente della lista ha due campi: il nome del metodo (conosciuto come il "selector" dello stesso) e la sua locazione in memoria.
Quando un oggetto cerca di chiamare un metodo, il comportamento di questo linguaggio è il seguente: compilando il codice, il compilatore ha tradotto il codice della chiamata (ad esempio:

```
[persona calcolaEta:dataDiNascita];
```

in

```
objc_msgSend(persona, @selector(calcolaEta:),dataDiNascita);
```

La funzione objc_msgSend() opera tramite un lookup dinamico: conoscendo il nome del metodo da ricercare scorre la lista per verificare la sua effettiva presenza e, se presente, lo esegue.
Questo comportamento ha caratteristiche particolari: potremmo far puntare un certo selettore A, che prima puntava al codice per A, ad un codice per un certo B.
Lo svantaggio è che il tempo di esecuzione è leggermente minore di una chiamata diretta alla parte di codice; si parla comunque di nano secondi di differenza.

Inoltre il runtime, prima di inviare il messaggio, richiede all'oggetto se il metodo è riconosciuto dallo stesso. Questo significa che l'oggetto può decidere se accettare il messaggio (è anche per questo che si possono inviare messaggi a nil), inoltrarlo ad un oggetto differente, decidere di eseguire codice differente per un metodo specifico.
Più in dettaglio, quando inviamo un messaggio ad un oggetto, non abbiamo garanzie sul fatto che:
-Il metodo che andiamo a chiamare non sia necessariamente quello che verrà effettivamente chiamato
-L'oggetto che riceverà il messaggio non sarà necessariamente quello che vogliamo che risponda.

1.2.2 Compilatore

L'IDE di Apple, Xcode, utilizza clang. Quest'ultimo è un compilatore front end per C, C++, Objective-C, Objective-C++, OpenMP, OpenCL e CUDA, basato sul compilatore LLVM (Low Level Virtual Machine).
LLVM in origine doveva utilizzare il front end di GCC, ma dopo aver riscontrato problemi di integrazione e di sviluppo l'azienda ha deciso di crearne uno proprio (clang) reso open-source nel Giugno 2007.

1.2.3 Utilizzo in iOS e frameworks Cocoa

Il linguaggio principale di tutti i frameworks (quali CoreOS, Foundation, CocoaTouch) di iOS e MacOS è ancora Objective-C come dal principio, in quanto Swift non ha ancora raggiunto un livello di maturità tale per essere impiegato per questi scopi (la versione 3 del linguaggio non ha ancora ABI stabili e la sintassi è ancora in corso di modifiche).L'intenzione di Apple è quella di mantenere e migliorare i due linguaggi parallelamente per ancora molto tempo.

Capitolo 2

Swift

2.1 Cenni storici

Lo sviluppo di Swift è iniziato nel 2010 da Chris Lattner, aiutato in seguito da molti altri programmatori. Swift ha preso idee "da Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, e molti altri".Il 2 giugno 2014 l'app per il WWDC è divenuta la prima app distribuita al pubblico scritta in Swift.
Il 3 dicembre 2015 viene lanciato il sito swift.org ed il codice sorgente del linguaggio è pubblicato con licenza Apache 2.0 sul repository GitHub dell'azienda.
Apple resta lo sviluppatore principale e ne rende disponibile anche una versione del compilatore per Linux (Creato appositamente per Ubuntu).
Il 13 settembre 2016, durante la WWDC 2016, Apple ha presentato la terza versione del suo linguaggio di programmazione insieme ad un'applicazione per iPad, Swift Playgrounds, che permette, tramite una grafica semplice e intuitiva, di imparare a programmare con Swift, soprattutto orientato ai più giovani.

2.2 Caratteristiche

2.2.1 Sintassi

Implementazione di una classe

In Swift, al contrario di molti altri linguaggi, non ci sono 2 file distinti per l'interfaccia e l'implementazione, ma uno solo con l'estensione .swift
Esempio di creazione di una nuova classe:

```
class Persona {

//dichiarazione delle properties

        var nome: String?
        var cognome: String?
        var eta: Int?

//costruttore personalizzato con parametri

        init(nome: String, cognome: String, eta: Int) {
                self.nome = nome
                self.cognome = cognome
                self.eta = eta
        }

//dichiarazione dei metodi setter e getter

        func getNome() -> String {

                return self.nome
        }

        func setNome(nome: String) {

                self.nome = nome
        }

        func getCognome() -> String {

                return self.cognome
        }

        func setCognome(cognome: String) {

                self.cognome = cognome
        }

        func getEta() -> Int {

                return eta
        }

        func setEta(eta: Int) {

                self.eta = eta
        }
}
```

Dichiarazione e definizione dei metodi

In Swift, una dichiarazione di funzione (metodo) ha la seguente sintassi:

```
//Nell'ordine: func nomeMetodo(nomeArg1:tipoArg1) -> tipoDiRitorno

func calcolaEta(dataDiNascita: NSDate) -> Int
```

Definizione del metodo appena dichiarato:

```
func calcolaEta(dataDiNascita: NSDate) -> Int

        let oggi = Date()

    let componentiCalendario = Calendar.current.dateComponents([.year], from:
            dataDiNascita, to: oggi)

        let eta = componentiCalendario.year!

        return eta;
}
```

Le funzioni in Swift sono trattate come oggetti, ciò significa che una funzione può ritornare un'altra funzione:

```
func creaIncrementatore() -> ((Int) -> Int) {

        func aggiungiUno(numero: Int) -> Int {

                return 1 + numero

        }

        return aggiungiUno
}

var incrementatore = creaIncrementatore()
incrementatore(7)
```

Closures

Le funzioni sono un caso speciale di "closure": il codice in una closure ha accesso a variabili e funzioni che sono disponibili nel suo scope, anche se viene eseguita in uno scope diverso. Una closure viene definita dalla sintassi { }, utilizzando il separatore ïnpër gli argomenti e il tipo di ritorno dal corpo:

```
var numeri = [2,25,21,89,90]

numeri.map({
        (numero: Int) -> Int in
        let risultato = 3 * numero
        return risultato
})
```

Ci sono vari modi per scrivere le closure: quando il tipo è già conosciuto, come per esempio in una callback per un delegate, si possono omettere i tipi dei parametri, il tipo di ritorno o entrambi nel caso in cui ci sia un singolo statement, in quanto la closure ritorna implicitamente il valore di ritorno:

```
let numeriInMap = numeri.map({ numero in 3 * numero })
```

Ci si può riferire ai parametri per numero invece che per nome, approccio utile specialmente in closure che richiedono poco codice; una closure passata come ultimo argomento di una funzione può essere scritta immediatamente dopo le parentesi, e se quest'ultima è l'unico argomento della funzione stessa si possono omettere le parentesi tonde:

```
//ordino i numeri in modo crescente

let numeriOrdinati = numeri.sorted { $0 > $1 }
```

Enumerazioni

La sintassi enum è utilizzata per dichiarare le enumerazioni in Swift. La particolarità rispetto ad Objective-C è che quest'ultime possono contenere metodi:

```
enum TipologieDiCase {

        case condominio, villa, indipendente, attico

        func descrizione() -> String {

                switch self {

                case .condominio:
                        return "Condominio"
                case . villa :
                        return "Villa"
                case .indipendente:
                        return "Casa indipendente"
                case . attico
```

```
                return "Attico"
            default:
                return String(self.rawValue)

            }
        }
}

let villa = TipologieDiCase.villa
let descrizioneVilla = villa.descrizione()
```

Protocolli ed estensioni

Un protocollo definisce un'interfaccia di metodi, variabili ed altri eventuali requisiti che definiscono una particolare funzionalità. Quest'ultimo può essere quindi adottato da una classe, struct o enum che ne forniranno l'implementazione:

```
protocol ProtocolloDiNavigazioni {

    func navigaAlleImpostazioni(sender: CollectionView)
    func navigaAlDettaglioEvento(sender: UICollectionView)
}

class MenuPrincipale: UICollectionViewController, UICollectionViewDelegateFlowLayout:
     ProtocolloDiNavigazioni {

    override func viewDidLoad() {
    super.viewDidLoad()
        collectionView?.delegate = self
        collectionView?.dataSource = self
    }

     ...

    //MARK: CollectionViewDelegate

    override func collectionView(collectionView: UICollectionView,
         didSelectItemAtIndexPath indexPath: NSIndexPath) {

        switch indexPath {

            case 0:
                navigaAlleImpostazioni(self.collectionView)
            case 1:
                navigaAlDettaglioEvento(self.collectionView)
        }
    }
```

```
        //MARK: Implementazione del protocollo

        func navigaAlleImpostazioni(sender: UICollectionView) {

        appDelegate.gotoSettingsVC()
    }

    func navigaAlDettaglioEvento(sender: UICollectionView) {

        appDelegate.gotoEventDetailVC()
    }
}
```

Le estensioni sono invece un modo per aggiungere funzionalità ad un tipo eisstente, come nuovi metodi e computed properties:

```
//Estensione che aggiunge un effetto di blur ad una imageView

extension UIImageView
{
    func aggiungiBlur()
    {
        let blurEffect = UIBlurEffect(style: UIBlurEffectStyle.Light)
        let blurEffectView = UIVisualEffectView(effect: blurEffect)
        blurEffectView.frame = self.bounds
        self.addSubview(blurEffectView)
    }
}

let containerImmagine = UIImageView()
containerImmagine.image = UIImage(named: "beer2beerlogo.jpg")
containerImmagine.aggiungiBlur()
```

Gestione degli errori

Gli errori vengono rappresentati utilizzando qualsiasi tipo che si conformi al protocollo Error; la parola chiave throws viene utilizzata per indicare che una funzione può ritornare un errore, utilizzando la parola chiave throw. Se si lancia un errore dall'interno di una funzione, quest'ultima ritorna immediatamente e l'errore viene gestito dalla funzione chiamante:

```
enum ErroriStampante: Error {

        case cartaEsaurita
        case inchiostroEsaurito
        case cassettoChiuso
```

```
}

func invia(lavoro: Int, allaStampante nomeStampante: Stringa) throws -> String {
        if nomeStampante = "Rusty old printer" {

                throw ErroriStampante.cassettoChiuso
        }

        return "Lavoro inviato alla stampante"
}

        do {

        let rispostaStampante = try invia(lavoro: 2303, allaStampante: "Sala meeting")

        print(rispostaStampante)

        } catch {

                print(error)
        }
```

Si possono inoltre utilizzare più blocchi catch per gestire errori specifici:

```
        do {

                let rispostaStampante = try invia(lavoro: 2303, allaStampante: "Sala
                        meeting")

                print(rispostaStampante)

                } catch ErroriStampante.cassettoChiuso {

                        print("Aprire il cassetto")
                }
```

Un altro modo per gestire gli errori è quello di utilizzare la parola chiave try? per convertire il risultato in un tipo optional: se la funzione lancia un errore, questo specifico errore è ignorato e la funzione ritorna nil; alternativamente il risultato è un optional contenente il valore ritornato dalla funzione:

```
let foglioStampato = try? invia(lavoro: 1984, allaStampante: "Sala meeting")
let erroreDiStampa = try? invia(lavoro: 1948, allaStampante: "Rusty old printer")
```

Generics

Una funzione generica, più comunemente chiamata template, è così dichiarata:

```
func creaArray<Elemento>(ripeti elemento: Elemento, numeroDiVolte: Int) -> [Elemento] {

        var risultato = [Elemento]()

        for _ in 0..< numeroDiVolte {

                risultato.append(elemento)
        }

        return risultato
}

//chiamata alla funzione

creaArray(ripeti: "Tick tock", numeroDiVolte: 3)
```

Si possono creare generics di funzioni o di classi, enumerazioni e struct:

```
//Reimplementazione del tipo optional della libreria standard di Swift

enum OptionalValue<Wrapped> {

        case none
        case some(Wrapped)
}

var possibileIntero: OptionalValue<Int> = .none
possibileIntero = .some(100)
```

Si utilizza la parola chiave where prima del corpo per indicare una lista di requisiti, per esempio per indicare che il tipo deve conformarsi ad un protocol, per richiedere che due tipi siano uguali o per indicare che una classe deve avere una particolare superclasse:

```
func elementiComuni<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> Bool {

        where T.Iterator.Element: Equatable,
                T.Iterator.Element == U.Iterator.Element {

                for lhsItem in lhs {

                        for rhsItem in rhs {

                                if lhsItem == rhsItem {
```

```
                                        return true
                                    }
                                }
                            }
                        }

                        return false
}

//chiamata alla funzione

elementiComuni([1,2,3], [3])
```

Tuples

Feature non presente in Objective-C, le tuple raggruppano più valori (di un tipo qualsiasi o tipi differenti) in un singolo valore composto. Per esempio, potremmo descrivere lo status code 404 dell'HTTP con una tupla in questo modo:

```
let errore404http = (404, "Not found")
```

La tupla (404, "Not found") raggruppa insieme un tipo Int e uno String; si può creare qualsiasi permutazione di tipi. Per ottenere i singoli valori da una tupla, quest'ultima deve essere scomposta:

```
let (statusCode, statusMessage) = errore404http
```

Questo particolare tipo è utile come valore di ritorno dalle funzioni, per esempio una funzione che ha il compito di caricare una pagina web potrebbe usare la tupla sopra descritta per indicare il successo o il fallimento del caricamento, fornendo più informazioni rispetto ad un valore di ritorno singolo di un singolo tipo.

Optionals

Questo tipo è un pilastro portante di Swift, in quanto viene utilizzato in tutti i casi in cui il valore di ritorno potrebbe essere nullo (nil). La logica di funzionamento è la seguente: o c'è un tipo di ritorno, e quindi si utilizza l'unwrapping per accedere al valore, o non c'è valore alcuno.
Il concetto di optional non esiste in C o Objective-C. Ciò che ci si avvicina di più è l'abilità di ritornare nil da un metodo che altrimenti ritornerebbe un oggetto, con nil a significare l'assenza di un oggetto valido.
Questo però funziona solamente per gli oggetti (non structs, tipi C o enumerativi); per questi i metodi Objective-C ritornano solamente un valore

speciale (come per esempio NSNotFound). Questo implica che il chiamante dei metodi sappia che c'è uno speciale valore da verificare, l'approccio di Swift permette invece di indicare l'assenza di qualsiasi valore in assoluto, senza la necessità per speciali costanti.
In questo esempio vediamo come gli optional possono essere utilizzati per gestire il caso di assenza di valore. Il tipo Int di Swift ha un costruttore che prova a convertire una String in un valore di tipo Int; questa conversione può fallire e quindi viene utilizzato un tipo optional:

```
let possibileNumero = "123"
let numeroConvertito = Int(possibileNumero)
//numeroConvertito e' di tipo Int?, che si legge come "optional Int"
```

Poichè il costruttore può fallire, questi ritorna un tipo optional Int, scritto Int?. Il punto di domanda indica che il valore contiene un tipo optional, il che significa che potrebbe contenere un valore Int, o nessun valore.
Utilizzando la speciale parola chiave nil, si indica che un tipo optional non ha valore alcuno. nil non può essere utilizzato con constanti e variabili non optional.

```
var codiceRispostaServer: Int? = nil
```

Se si crea una variabile optional, alla quale non si assegna alcun valore, a quest'ultima viene automaticamente assegnato nil.
Il nil di Swift non è però equivalente a quello di Objective-C: in quest ultimo, nil è un puntatore ad un oggetto che non esiste, in Swift non è invece un puntatore, è l'assenza di valore alcuno. Optionals di qualunque tipo possono essere settati a nil, non solamente oggetti.
Per verificare la presenza di valore in un tipo optional è possibile usare lo statement if in questo modo:

```
if numeroConvertito != nil {

        //numero convertito contiene un valore
}
```

Alternativamente è possibile utilizzare il simbolo !, letto come forced unwrapping del tipo optional; questo approccio è rischioso e porta spesso ad errori in runtime. Viene utilizzata molto più frequentemente la sintassi dell'optional binding, per verificare se una certa variabile optional contiene un valore e, se presente, assegnarlo ad una costante o variabile temporanea. Si utilizza con gli statement if e while:

```
if let numero = Int(possibileNumero) {
        //se entro nel blocco significa che possibileNumero
```

```
        //contiene un intero poiche' la conversione ad Int va a buon fine
} else {
        //possibileNumero non e' stato convertito ad Int
}
```

Questo codice può essere letto come: se l'optional Int ritornato da Int(possibileNumero) contiene un valore, assegna alla nuova costante chiamata numero il valore contenuto nell'optional. Questa costante è già stata inizializzata con il valore contenuto all'interno dell'optional, quindi non è necessario utilizzare la sintassi ! per forzare l'accesso al valore. In uno statement si possono concatenare più optional binding separati da virgola, se almeno uno di questi valori è nil o una condizione booleana valuta a false, l'intera condizione dello statement viene considerata falsa.
Le costanti e le variabili create tramite l'optional binding in uno statement if sono accessibili solamente all'interno dello statement stesso; per permettere l'accesso anche alle linee di codice che seguono lo statement, è necessario utilizzare lo statement guard.

2.2.2 Compilatore e libreria standard

Architettura del compilatore

Il compilatore di Swift è composto dai seguenti componenti principali:

- Parser: il parser è un semplice parser ricorsivo discendente con un lexer codificato a mano. Il parser è il responsabile della generazione dell'albero di sintassi astratta senza alcuna informazione semantica o di tipo, e genera errori in caso di problemi grammaticali nel sorgente.

- Analizzatore semantico: trasforma l'albero generato dal parser in un albero ben formato e con controllo sui tipi. Questo analisi include la type inference e in caso di successo, indica che è sicuro generare il codice dall'albero creato

- Clang importer: Importa i moduli di Clang e mappa le API C o Objective-C nelle rispettive API Swift. Gli alberi risultanti vengono utilizzati dall'analizzatore semantico.

- Generatore SIL: SIL è l'acronimo di Swift Intermediate Language, ovvero un linguaggio intermedio di alto livello, specifico per l'analisi e l'ottimizzazione del codice. Questa fase trasforma l'albero di sintassi astratta creato dall'analizzatore in un cosiddetto SIL "grezzo".

- Trasformazioni SIL: Questo strumento esegue ulteriori diagnostiche che influenzano la correttezza del programma (come ad esempio l'uso di variabili non inizializzate). Il risultato finale di queste trasformazioni è SIL "canonico".

- Ottimizzazioni SIL: Questa fase esegue ulteriori ottimizzazioni specifiche di alto livello, quali Automatic Reference Counting per la gestione della memoria, devirtualizzazione e specializzazione dei tipi generics.

- Generazione IR LLVM: IR significa Intermediate Representation. Questa fase passa dal SIL al LLVM IR, e a questo punto LLVM può continuare ad ottimizare il codice e generare il codice macchina.

Libreria standard Swift

La libreria standard Swift comprende un certo numero di tipi di dati, protocolli e funzioni, compresi i tipi fondamentali (ad esempio, int, double), collezioni (per esempio, Array, dizionario) insieme ai protocolli che li descrivono e gli algoritmi che operano su di essi, i caratteri e stringhe, e le primitive di basso livello (ad esempio, UnsafeMutablePointer).
Il repository della libreria standard viene ulteriormente suddiviso:

- Nucleo principale: include la definizione di tutti i tipi, protocolli e funzioni.

- Runtime: Il runtime di supporto è il componente che risiede nel mezzo tra il compilatore e il nucleo della libreria standard. E' il responsabile dell'implementazione delle features dinamiche del linguaggio, come il casting (ad esempio per gli operatori as! ed as?) o i metadata dei tipi (per supportare i generics e la reflection) e la gestione della memoria (allocazione degli oggetti, reference counting). Differentemente dalle altre librerie di alto livello, il runtime è scritto quasi esclusivamente in linguaggio C++ o Objective-C.

- Overlays per SDK: componenti specifici per le piattaforme Apple, forniscono modifiche ed aggiunte specifiche per Swift ai framework Objective-C, per migliorare la loro interoperabilità.

Whole module optimization

Con la versione 3.0 di Swift è stata introdotta una modalità di ottimizzazione del compilatore che, dipendentemente dal progetto, permette miglioramenti delle performance significativi.

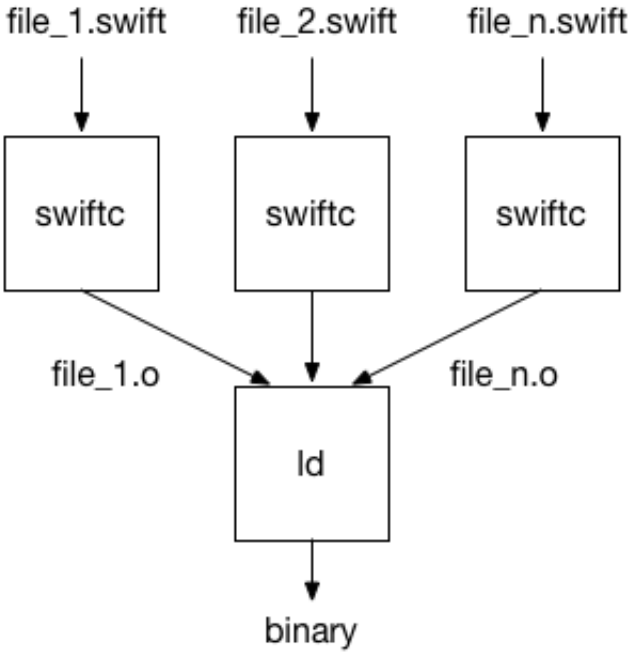

Figura 2.1: *Visualizzazione del lavoro del compilatore in modalità di compilazione per ogni singolo file*

Funzionamento del compilatore senza WMO: Un modulo è un set di files Swift, ed ogni modulo viene compilato in una singola unità di distribuzione (un framework o un eseguibile). Nella compilazione a file singolo il compilatore è invocato separatamente per ogni file nel modulo: Dopo la lettura ed il parsing del singolo file, il compilatore ottimizza il codice, genera il codice macchina e scrive un file oggetto. Successivamente il linker unisce tutti i file oggetto e genera la libreria condivisa o l'eseguibile.
Questa compilazione singola limita le ottimizzazione inter funzione, come l'inlining o la specializzazione dei generics alle le funzioni chiamate all'interno dello stesso file. Come esempio, assumiamo che un file di un modulo, chiamato utils.swift, contenga una struttura dati generica chiamata Container<T>, con un metodo chiamato getElement. Questo metodo è chiamato nel modulo, ad esempio nel file main.swift:

```
//main.swift:

func add (c1: Container<Int>, c2: Container<Int>) -> Int {
  return c1.getElement() + c2.getElement()
```

```
}
```

```
// utils.swift:

struct Container<T> {
  var element: T

  func getElement() -> T {
    return element
  }
}
```

Quando il compilatore ottimizza il file main.swift non conosce come la funzione getElement sia implementata, conosce solo il fatto che è presente; viene quindi generata una chiamata al suddetto metodo.
Analogamente, quando il compilatore ottimizza il file utils.swift non conosce per quale tipo concreto il metodo viene chiamato, quindi può generare solamente una versione generica della funzione, rendendo il codice più lento rispetto ad uno ottimizzato per un tipo concreto.
Anche solamente uno statement return in getElement necessita di un lookup di tipo per verificare come copiare l'elemento; può trattarsi di un tipo Int o di un tipo più dispendioso in termini di risorse che richiede operazioni di reference counting. Funzionamento del compilatore con WMO:
Con l'ottimizzazione per l'intero modulo il compilatore ottimizza tutti i file di un modulo nella loro interezza: Questo porta due notevoli vantaggi: il compilatore vede le implementazioni di tutte le funzioni del modulo, e può quindi procedere con le ottimizzazioni specifiche; specializzare una funzione significa che il compilatore crea una nuova versione del metodo specifica per il contesto della chiamata. Ad esempio, il compilatore può ottimizzare le funzioni di tipo generics per i tipi concreti.
Nell'esempio, il compilatore crea una versione della struct Container specifica per i tipi Int:

```
struct Container {
  var element: Int

  func getElement() -> Int {
    return element
  }
}
```

Può quindi procedere all'inlining del metodo specializzato getElement all'interno della funzione add:

```
func add (c1: Container<Int>, c2: Container<Int>) -> Int {
```

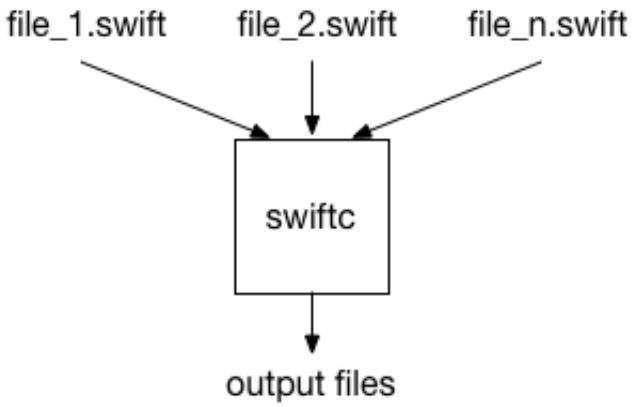

Figura 2.2: *Visualizzazione del lavoro del compilatore in modalità di compilazione whole module optimization*

```
  return c1.element + c2.element
}
```

Contrariamente alla compilazione a file singolo, il lavoro viene eseguito in poche istruzioni macchina

2.2.3 Utilizzo in iOS e frameworks Cocoa

TODO

Capitolo 3

Confronto tra i linguaggi (trovare un titolo adeguato)

3.1 Punti di forza e debolezze di Objective-C

TODO

3.2 Punti di forza e debolezze di Swift

TODO

3.3 Confronto delle performance e gestione della memoria

Objective-C supporta due meccanismi per la gestione della memoria:

-MMR (Manual retain-release), dove lo sviluppatore gestisce esplicitamente la memoria, tenendo traccia degli oggetti instanziati. E' implementato tramite un modello chiamato Reference Counting, fornito dalla classe NSObject in congiunzione all'ambiente di runtime; è il metodo più obsoleto e più dispendioso in termini di tempo di sviluppo in quanto è un approccio prettamente manuale.

-ARC (Automatic reference counting), che utilizza lo stesso sistema di tracciamento degli oggetti di MMR, ma aggiunge automaticamente chiamate ai metodi di gestione della memoria a tempo di compilazione. Questo sistema permette di assicurare che gli oggetti abbiano vita il tempo necessario per il

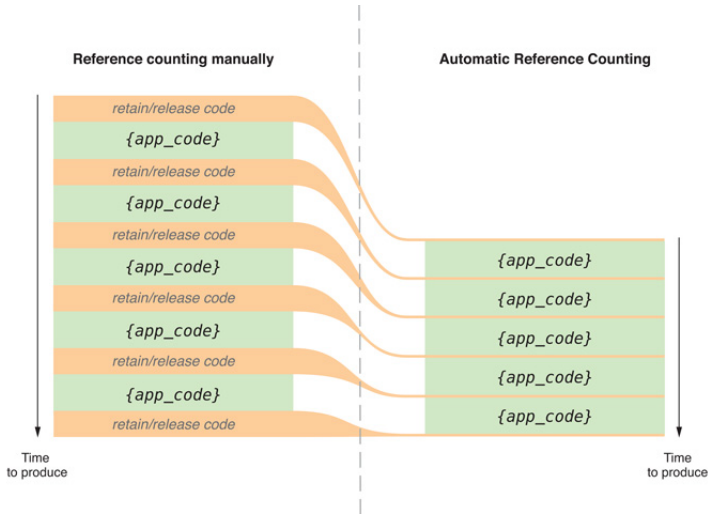

Figura 3.1: *Confronto tra MMC ed ARC relativo al tempo di creazione dei cicli di retain-release degli oggetti*

loro utilizzo e non oltre, poichè il compilatore genera in automatico anche i metodi di dealloc appropriati.
E' l'approccio moderno e più utilizzato della gestione della memoria in Objective-C.

Capitolo 4

Il futuro della piattaforma iOS

4.1 Quale futuro per i due linguaggi?

Swift poco utilizzato al momento, persino da Apple Da intervista Lattner i due linguaggi andranno di pari passo per molto tempo Swift non ha ABI stabili

Conclusioni

TODO

Ringraziamenti

TODO

Bibliografia

[1] Objective-C
Publications, Books, Articles, Interviews, etc.
`http://virtualschool.edu/objectivec/`

Appendice

Il codice del progetto è disponibile su GitHub all'indirizzo
`https://github.com/thebertozz/Tesi`