

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Fondamenti di Informatica T-2

**Applicazioni multi-paradigma e multi-linguaggio:
il caso Prolog/Swift/Java**

CANDIDATO:

ALBERTO SITA

RELATORE:

Chiar.mo Prof. ENRICO DENTI

CORRELATORE:

Ing. ROBERTA CALEGARI

Anno Accademico 2014/15

Sessione II

Indice

Introduzione	1
1 Linguaggi iOS: Objective-C e Swift	3
1.1 Objective-C	3
1.1.1 Sintassi	3
1.1.2 Analisi del linguaggio	4
1.2 Swift	5
1.2.1 Sintassi	5
1.2.2 Analisi del linguaggio e confronto con Objective-C	6
2 Interfacciamento fra Java e iOS	8
2.0 Architetture	8
2.1 Bridge software	10
2.2 Mix and Match: Objective-C e Swift	11
2.2.1 Importare Objective-C in Swift	12
2.2.2 Importare Swift in Objective-C	13
2.2.3 Usare Swift da Objective-C	13
2.3 Java ed Objective-C	15
2.3.1 Java Native Interface	15
2.3.2 Java Native Access	18
2.4 Approcci a confronto	20
3 Strumenti per la traduzione automatica di codice	21
3.1 RoboVM	21
3.1.1 Architettura generale	22
3.1.2 Metodologia di sviluppo	23
3.2 Fire	27
3.2.1 Elements	27
3.2.2 Silver e Swift	28
3.2.3 L'IDE	29
4 tuProlog: un caso di studio multiplatforma	31
4.1 tuProlog: integrazione in un applicativo iOS	32

4.1.1 Progetto Xcode	34
4.1.2 Progetto Eclipse	35
4.1.3 Simulazione e collaudo	36
4.2 tuProlog e Swift tramite Fire	38
5 Conclusioni	42
Appendici	44
Bibliografia	60

Introduzione

L'interesse per applicativi multi-paradigma e multi-linguaggio è dovuto principalmente alla recente necessità di coniugare tecnologie eterogenee per realizzare programmi flessibili e non rigidamente vincolati ad un unico runtime. Piattaforme come iOS, Android e Windows Phone hanno raggiunto la maturità cambiando l'utilizzo comune dei cellulari, riducendo il gap tra i computer e dispositivi post-pc. Ciò ha permesso di sviluppare applicativi per device mobili paragonabili, per quanto riguarda la complessità e funzionalità, a programmi eseguiti su computer desktop. La possibilità di realizzare software di qualità che sfrutti appieno le risorse rese disponibili dall'hardware di ultima generazione ha sollevato problematiche legate alle politiche di sviluppo del software stesso ed ai relativi costi perché una applicazione scritta per una determinata piattaforma non è compatibile con un'altra a causa di differenze nei runtime coinvolti e nelle rispettive architetture hardware. Inoltre i vari progetti per lo stesso applicativo devono rimanere allineati ad ogni aggiornamento per non creare derive tra le diverse distribuzioni del prodotto che potrebbero penalizzare una fascia d'utenza legata ad una specifica piattaforma.

In questa tesi l'attenzione sarà posta sulle tecnologie, gli strumenti ed i linguaggi coinvolti nello sviluppo di applicazioni multipiattaforma, multi-paradigma e multi-linguaggio, con particolare interesse nei confronti del mondo Apple, nello specifico iOS. Obiettivo principale è individuare una metodologia di sviluppo del software che permetta la realizzazione di applicativi flessibili, utilizzabili su piattaforme differenti, sia desktop che mobile, che consenta di utilizzare in un unico progetto più linguaggi di programmazione e che non costringa lo sviluppatore a riprogettare componenti software precedentemente realizzati e testati, il tutto senza giungere a compromessi in termini di ottimizzazione ed efficienza.

Come primo caso di studio si considererà il seguente problema: si vuole integrare in una *app* iOS l'ultima versione Java del motore Prolog *tuProlog* sviluppato presso il DISI dell'Università di Bologna[1]. L'obiettivo è poter usare l'engine Prolog su iPhone ed iPad senza doverlo riprogettare, superando così le limitazioni imposte da Apple che

impediscono di usare Java come linguaggio di programmazione. Per evitare refactoring del codice occorre approfondire le possibilità di interazione tra Java ed i linguaggi Apple Swift[2] ed Objective-C[3], analizzando le loro differenze e somiglianze. Inoltre si testeranno le potenzialità di RoboVM, un tool che permette la traduzione automatica di codice Java verso Objective-C.

Il secondo caso di studio verterà sull'utilizzo di *tuProlog* in una applicazione scritta in Swift. Lo scopo è valutare la possibilità di richiamare l'engine da codice Swift ed eseguire l'applicativo in un ambiente diverso da quello nativo Apple, ad esempio Java. Si esploreranno quindi le potenzialità di Fire, strumento che consente l'utilizzo di C#, Oxygene[4] e Swift per lo sviluppo di software Java.

1 Linguaggi iOS: Objective-C e Swift

Apple permette agli sviluppatori iOS di utilizzare due linguaggi: Objective-C ed il più recente Swift. Entrambi sono linguaggi imperativi ad oggetti ma il secondo risulta più moderno sia per quanto riguarda la sintassi che per quanto riguarda le funzionalità disponibili[5]. Entrambi i linguaggi possono essere utilizzati nello sviluppo di un'unica applicazione e godono di una forte integrazione.

1.1 Objective-C

1.1.1 Sintassi

Objective-C, sviluppato all'inizio degli anni ottanta, è considerato un linguaggio di programmazione relativamente ostico[3]. Gran parte della sintassi come le clausole del preprocessore, le espressioni, le dichiarazioni e le chiamate di funzioni è stata mutuata da quella del C, mentre la sintassi relativa alle caratteristiche object-oriented è stata realizzata per ottenere una comunicazione basata su scambio di messaggi. Questa fa sì che in Objective-C non ci sia una chiamata vera e propria ad un metodo ma che venga inviato un messaggio. Si dice quindi che un *oggetto* di una certa classe che implementa un metodo X risponda ad un messaggio X. La sintassi è la seguente:

[oggetto metodo: argomento];

In questo modo viene inviato all'oggetto un messaggio con l'argomento desiderato. Al contrario in Java e C# la notazione utilizzata per le chiamate a metodi è quella con il punto tra l'oggetto e il metodo la quale fornisce più immediatezza, soprattutto se vi sono più chiamate innestate. Ad esempio per creare un'oggetto *Studente* in Objective-C, questa potrebbe essere una soluzione:

*Studente *s = [[Studente alloc] initWithMatricola: @"0000653874"];*

Mentre in Java lo stesso codice sarebbe:

```
Studente s = new Studente("0000653874");
```

Come si può notare, in Objective-C un oggetto deve essere prima allocato e poi inizializzato. Per riservare lo spazio necessario in memoria bisogna chiamare il metodo *alloc* e successivamente uno dei metodi *factory* forniti dalla classe utilizzata che inizializzano l'istanza corrente. Osservando il codice Java, si nota come questo sia più intuitivo: per creare l'oggetto basta chiamare la *new* dell'oggetto desiderato, passando gli opportuni argomenti. Java nasconde completamente al programmatore l'idea di puntatore mentre Objective-C ne consente l'utilizzo permettendo la scrittura di codice non sicuro.

1.1.2 Analisi del linguaggio

L'implementazione di Objective-C usa un runtime scritto in C che aumenta di poco la dimensione delle applicazioni. Al contrario di quanto avviene per la maggior parte dei linguaggi object-oriented esistenti che utilizzano una macchina virtuale, i programmi Objective-C tendono ad essere poco più ampi delle dimensioni del loro codice oggetto e delle librerie usate.

Objective-C utilizza una tipizzazione dinamica debole che aumenta la flessibilità ma lascia più spazio a errori che durante la compilazione o non vengono segnalati o compaiono solamente come warning. Il meccanismo di scambio di messaggi è realizzato tramite un sistema dinamico di risoluzione dei metodi, il quale permette anche di cambiare l'implementazione di un metodo a runtime (*method swizzling*). Altri aspetti caratteristici, alcuni dei quali hanno suscitato diverse critiche, sono il mancato supporto alla garbage collection (fino al 2006), ai namespaces e all'overloading dei metodi.

Apple ha introdotto ARC, automatic reference counting, per sopperire all'assenza del garbage collector e al posto degli spazi di nomi gli sviluppatori sono forzati ad aggiungere un prefisso ai nomi delle proprie classi per evitare collisioni. Per questi e altri motivi Objective-C risulta un linguaggio di programmazione piuttosto lontano dalla pulizia e dalla semplicità di Java. Il recente interesse per Objective-C è da motivarsi con il grande

successo ottenuto da iPhone ed iOS di Apple.

1.2 Swift

1.2.1 Sintassi

Swift è un linguaggio di programmazione presentato da Apple nel 2014, concepito per semplificare la scrittura del codice[5]. Può coesistere con Objective-C ma non mantiene la compatibilità verso C. È stato progettato per essere più resistente agli errori ed è fortemente tipizzato. Una innovazione importante introdotta da Swift risiede nella sua sintassi: il linguaggio ne utilizza una moderna, semplice ed intuitiva, più pratica rispetto a quella di Objective-C, simile a quella di Java o C#.

Swift rende superfluo il metodo statico `main` che fungeva da start point ai precedenti programmi Objective-C. Una riga di codice come:

```
println("Hello, world!")
```

costituisce un programma completo che non necessita di alcuna direttiva di import di librerie che trattano l'input/output o la gestione di stringhe. Inoltre, se dopo una istruzione si va a capo, non è richiesta la presenza del punto e virgola come terminatore.

Swift è dotato di *type inference*, cioè permette al programmatore di non specificare alla dichiarazione il tipo di una variabile. Ad esempio:

```
var mele  
...  
mele=3
```

la variabile *mele* è un intero ma il tipo non è stabilito staticamente ma dedotto a runtime quando si ha l'assegnamento del valore 3. Swift consente anche la possibilità di esplicitare fin da subito il tipo di una variabile se necessario.

In Swift le chiamate a metodi utilizzano la classica forma dei più diffusi linguaggi ad oggetti ed i metodi o le funzioni sono considerati dei tipi di *prima classe*, ciò significa che una funzione può a sua volta ritornare una funzione.

Ad esempio:

```
1  func makeIncrementer() -> (Int -> Int) {  
2      func addOne(number: Int) -> Int {  
3          return 1 + number  
4      }  
5      return addOne  
6  }  
7  var increment = makeIncrementer()  
8  increment(7)
```

Figura 1 - Esempio di restituzione di una funzione

Come si vede in *Figura 1* il metodo *makeIncrementer* restituisce una funzione da intero a intero. La *riga 7* dichiara una variabile di tipo funzione la quale immagazzina un riferimento al metodo *addOne* di *makeIncrementer*. La *riga 8* costituisce una invocazione anonima di tale metodo[5][6].

1.2.2 Analisi del linguaggio e confronto con Objective-C

Attualmente non si ha accesso al runtime di Swift 1.2 in quanto non è open source come accade invece per Objective-C. Quando Apple rilascerà la versione 2.0 di Swift come codice open source si potranno avere maggiori dettagli riguardo al compilatore del linguaggio e ad altre nuove caratteristiche implementative. Le informazioni attualmente disponibili sono quelle divulgate dalla stessa Apple e quelle dovute ad azioni di reverse-engineering su codice Swift compilato.

Per la gestione dei metodi, Swift utilizza una *virtual method table*, chiamata *vtable*[7], la quale elenca tutti i metodi disponibili in una classe. Questa tabella viene creata a compile time e contiene puntatori a funzione che vengono utilizzati a runtime per eliminare il

dispatch dinamico caratteristico di Objective-C. La VMT dovrebbe garantire un accesso più rapido ai metodi rispetto al meccanismo di scambio di messaggi di Objective-C. È possibile però forzare Swift ad utilizzare il dispatch dinamico tipico di Objective-C in più modi: ereditando da una classe Objective-C, marcando una classe Swift con la clausola *dynamic* o contrassegnandola con l'attributo *@objc*. Se si utilizza la keyword *dynamic*, la classe in questione è automaticamente marcata *@objc*. Mentre però *dynamic* assicura che la risoluzione delle chiamate a metodo avvenga come per Objective-C, ovvero che sia anche possibile il cambio a runtime di una implementazione di una funzione, *@objc* espone le API Swift ad Objective-C consentendo l'interazione tra i due linguaggi, ma non garantisce il dispatch dinamico di proprietà, metodi o di initializer in quanto il compilatore Swift potrebbe bypassare la procedura per ottimizzare il codice compilato.

In conclusione Swift unisce semplicità a produttività in un linguaggio ad oggetti moderno, intuitivo e soprattutto compatibile con la grande quantità di codice Objective-C disponibile.

2 Interfacciamento fra Java e iOS

2.0 Architetture

Il mondo Java è un esempio di come un linguaggio ad oggetti possa essere portabile su diverse architetture[8][9]. Ciò è reso possibile grazie alla presenza di una macchina virtuale (JVM - *Java Virtual Machine*) la quale ha il compito di astrarre gli applicativi dalla piattaforma host scelta.

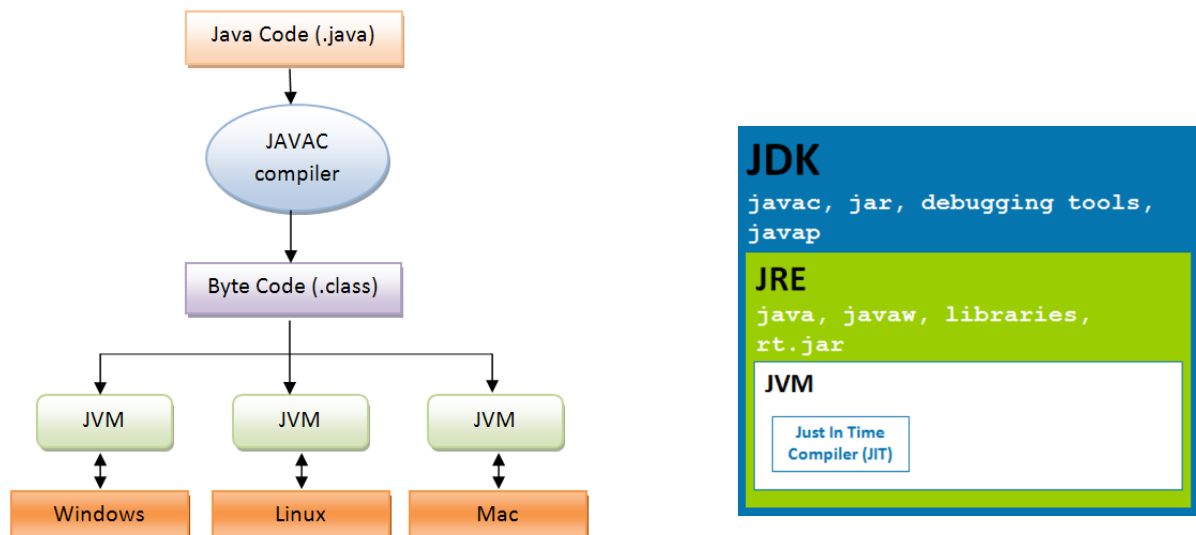


Figura 2 - Struttura di Java

L'infrastruttura Java si suddivide in *Java Development Kit* (JDK) che contiene tutti gli strumenti necessari per lo sviluppo di programmi Java e *Java Runtime Environment* (JRE) che racchiude il necessario per l'esecuzione degli applicativi. Dopo la compilazione, l'application launcher chiama il JRE, carica le classi necessarie ed invoca il metodo main. Il *Just In Time Compiler* (JIT) è un modulo interno alla JVM che ha il compito di compilare alcune parti del *Byte Code* in codice macchina per ottenere migliori performance a tempo di esecuzione. Ciò che non viene tradotto da JIT viene interpretato dalla virtual machine e poi eseguito.

Seguendo l'ordine delle frecce in *Figura 2*, i passi da svolgere per eseguire un programma scritto in Java sono i seguenti:

- Scrivere una o più classi Java tramite un editor di testo.
- Il compilatore trasforma il programma in *Byte Code* che è una astrazione del codice macchina conforme al principio *Write Once, Run Everywhere* in quanto è portabile su più architetture.
- Il *Byte Code* viene fornito ad una macchina virtuale Java che lo esegue.
- La JVM si interfaccia con il sistema operativo sottostante.

Si supponga di voler sviluppare in Java delle *app* sia per Android che per iOS[10], i sistemi operativi più diffusi per smartphone e tablet. Mentre Android si basa su Java, iOS invece non offre alcun supporto a questo runtime, ovvero non dispone di tutta l'infrastruttura che permettere l'esecuzione di applicativi basati su JVM. Si rende quindi necessaria una soluzione che sfrutti quanto già progettato in Java e che lo integri con ciò che è necessario sviluppare di nuovo, ad esempio l'interfaccia grafica, per permettere alle applicazioni di funzionare correttamente in iOS.

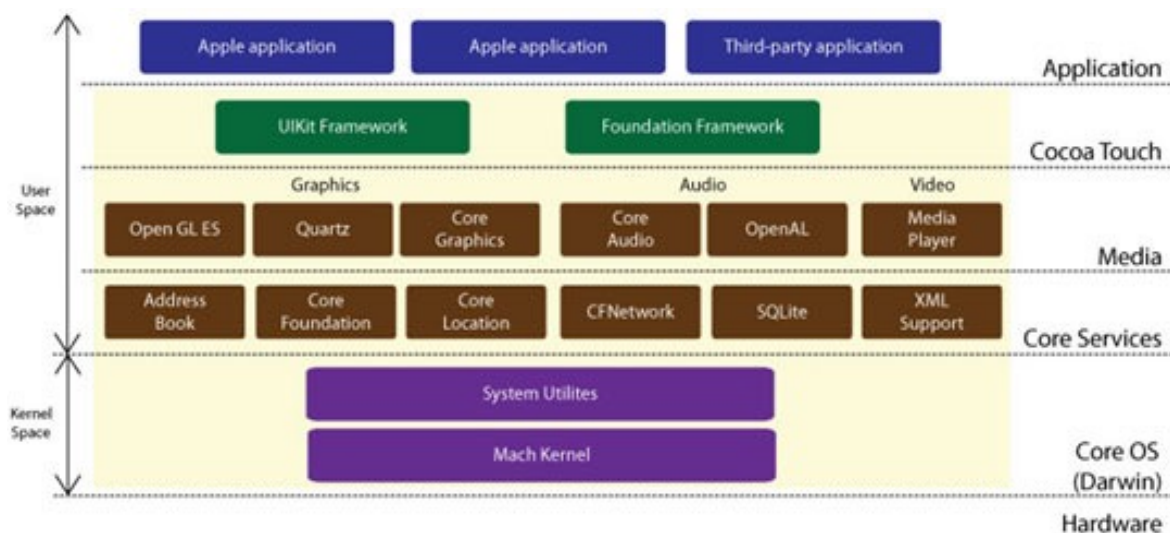


Figura 3 - Architettura di iOS

Considerando l'architettura di iOS mostrata in *Figura 3*, si nota come le applicazioni risiedano sopra il framework Cocoa Touch il quale fornisce, tramite interfacce (per esempio *UIKit Framework* per la grafica e *Foundation Framework* per le librerie di sistema), gli strumenti di base per la creazione di *app*. Il framework è basato sull'insieme di tool fornito dalle Mac OS X API, come queste è principalmente scritto in Objective-C e consente sia l'utilizzo di funzionalità software che di componenti hardware specifici per i dispositivi iOS. Per quanto riguarda la grafica segue il pattern architetturale MVC, *Model-View-Controller*.

2.1 Bridge software

Se si ha necessità di far interagire due oggetti differenti che vivono in ambienti diversi si potrebbe utilizzare un componente software che funga da tramite tra i due runtime coinvolti e che permetta la comunicazione tra essi. Questo componente prende il nome di bridge, ovvero ponte; esso ha il compito di rendere trasparenti all'applicativo le differenze presenti tra le varie piattaforme.

Un bridge si comporta come un adattatore: permette di ottenere flessibilità e software riutilizzabile ma, se implementato a livello applicativo, può appesantire l'esecuzione di un programma in quanto si pone come ulteriore strato di logica che introduce *overhead*. Al contrario, se il meccanismo di bridge è implementato a basso livello, come avviene per Objective-C e Swift, si ottengono prestazioni migliori. In ogni caso però, in base a come è realizzato il ponte, vanno verificate le compatibilità tra i linguaggi coinvolti in quanto non tutte le caratteristiche di ciascuno di questi potrebbero essere portabili.

2.2 Mix and Match: Objective-C e Swift

Essendo Swift molto recente, ed essendo disponibile molto software scritto in Objective-C grazie al grande successo avuto da iOS fin dal 2007, anno di lancio di iPhone, Apple permette in un progetto Xcode di utilizzare entrambi i linguaggi. Questa funzionalità prende il nome di *Mix and Match*[11] e, realizzando un bridge tra i due linguaggi, consente allo sviluppatore di creare applicazioni che utilizzano le più recenti funzionalità di Swift in simbiosi con codice Objective-C già esistente. La compresenza è possibile anche se Swift è diverso da Objective-C in quanto Swift è stato studiato fin dall'inizio per interfacciarsi con il precedente linguaggio.

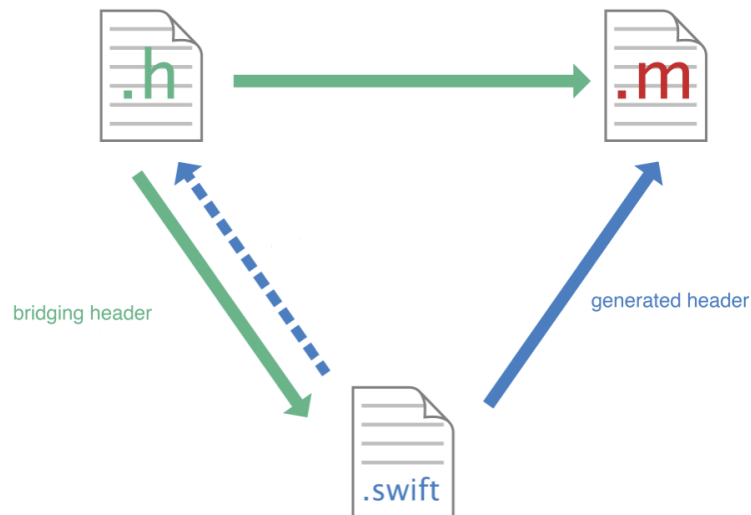


Figura 4 - Logica alla base di Mix and Match

Come visibile in *Figura 4*, *Mix and Match* si basa sull'utilizzo di header file contenenti le dichiarazioni del codice che si vuole esporre ad Objective-C o Swift. Le frecce verdi indicano i file header da creare esplicitamente ed importare nei sorgenti mentre quella blu indica la stessa azione ma compiuta in automatico da Xcode. Il tratteggio blu indica che una o più classi Swift derivano da una classe Objective-C.

Segue ora una trattazione approfondita sull'utilizzo in un unico progetto di entrambi i linguaggi. Il sottoparagrafo 2.2.1 fornisce una descrizione *low level* di come importare Objective-C in Swift. Il sottoparagrafo 2.2.2 analizza il caso contrario. Infine al sottoparagrafo 2.2.3 si discute di come utilizzare classi Swift in Objective-C.

2.2.1 Importare Objective-C in Swift

Per importare una serie di sorgenti scritti in Objective-C ed utilizzarli da codice Swift^[12] oppure per aggiungere un file Swift in un progetto Objective-C, si utilizza un file di header che ha il compito di esporre le API Objective-C a Swift. Il file header di bridge ha come nome il *product module*, ovvero il nome del prodotto impostato alla creazione del progetto, seguito da un suffisso che ne denota la funzione. È possibile creare questo file anche selezionando in Xcode: *File > New > File > (iOS or OS X) > Source > Header File*. Poi, per poter chiamare in Swift codice Objective-C, è necessario importare in esso tutti gli header dei sorgenti che si vogliono esporre.

Dopo aver modificato l'header inserendo gli *#import* necessari per l'applicazione che si sta sviluppando, è bene controllare sotto la voce *Build Settings* di Xcode che il compilatore Swift abbia un percorso relativo al progetto che punti a questo file. Nella maggior parte dei casi non si ha necessità di modificare le configurazioni di default.

Qualsiasi header Objective-C pubblico indicato nel bridging header sarà esposto a Swift e le implementazioni di tali interfacce potranno essere usate nei sorgenti Swift con la stessa sintassi impiegata per chiamare classi di sistema.

2.2.2 Importare Swift in Objective-C

Quando si vuole esporre codice Swift ad Objective-C[12] si utilizza un file header generato in automatico da Xcode. Questo file è un header Objective-C che dichiara le interfacce Swift che si vogliono condividere ed ha come nome il product module seguito da *-Swift.h*. Di default questo file contiene le interfacce delle classi Swift marcate *public* o al più *internal* se nel progetto è presente anche il bridging header per usare Objective-C da Swift. Le dichiarazioni marcate *private* non compaiono nel file generato in automatico e non sono esposte ad Objective-C a meno che non siano marcate con i seguenti attributi: *@IBAction*, *@IBOutlet* o *@objc*.

Non è necessario compiere alcuna operazione sull'header: basta importarlo per usare quanto scritto in Swift da Objective-C. Le interfacce Swift aggiunte nell'header includono a loro volta i riferimenti a tutti i tipi Objective-C in esse utilizzati. Se si usano propri oggetti Objective-C nel codice Swift da esportare bisogna controllare di aver importato gli opportuni header prima di inserire quello generato da Xcode all'interno del file .m in cui si vuole accedere in Objective-C a codice Swift.

2.2.3 Usare Swift da Objective-C

Una volta importato il codice Swift in Objective-C si può usare la sintassi di Objective-C per istanziare oggetti Swift come mostrato in *Figura 5*.

```
1  MySwiftClass *swiftObject = [[MySwiftClass alloc] init];  
2  [swiftObject swiftMethod];
```

Figura 5 - Creazione di un oggetto Swift con sintassi Objective-C

Una classe Swift per essere accessibile ed utilizzabile da Objective-C deve essere marcata `@objc` o derivare direttamente da *NSObject*. L'attributo `@objc` notifica al compilatore che la seguente porzione di codice Swift può essere acceduta in Objective-C. Invece se la classe Swift deriva da una classe Objective-C o direttamente da *NSObject*, il compilatore aggiunge automaticamente l'attributo `@objc`.

`@objc` permette di chiamare in Objective-C codice Swift ma non rende disponibile ad Objective-C funzionalità tipiche di Swift come i tipi generici, le tuple, le strutture definite in Swift, gli enumerativi definiti senza il tipo *raw int*, le variabili globali, i tipi annidati, le funzioni *top-level* ed alias di tipi Swift. Ad esempio un metodo avente nella propria firma un tipo generico come argomento ed una tupla come valore di ritorno non può essere utilizzato da Objective-C. Inoltre non è possibile derivare una classe Swift da una Objective-C, mentre è possibile il contrario.

2.3 Java ed Objective-C

Se si dispone di codice eseguibile Java e lo si vuole riutilizzare in una applicazione basata sul runtime Objective-C, è possibile utilizzare la *Java Native Interface*[13] per la comunicazione tra questo ambiente e Java (sottoparagrafo 2.3.1). Al contrario, se si lavora in Java e si desidera utilizzare oggetti Objective-C, è possibile usare un bridge che sfrutta *Java Native Access*[14] per inviare messaggi al runtime Objective-C (sottoparagrafo 2.3.2). In entrambi i casi l'approccio utilizzato prevede il coinvolgimento di codice nativo che non rende portabili gli applicativi sviluppati.

2.3.1 Java Native Interface

JNI o *Java Native Interface* è una API standard che realizza un bridge software e permette a questo di richiamare o di essere richiamato da codice nativo, ovvero specifico di un determinato sistema operativo o, più in generale, scritto in altri linguaggi di programmazione quali C, C++ o assembly.

In JNI sono rilevanti i puntatori *JavaVM* e *JNI interface pointer*: il primo rappresenta la Java Virtual Machine con cui si vuole comunicare, il secondo rappresenta l'accesso alle implementazioni delle interfacce JNI disponibili. La *Figura 6* mostra come sono gestiti i puntatori alle interfacce JNI.

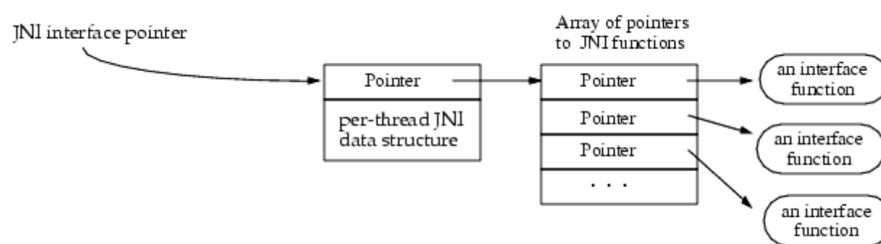


Figura 6 - Gestione dei puntatori alle interfacce JNI.

Se si utilizza JNI da Objective-C per interpellare il runtime Java, è comodo dichiarare la porzione di codice in *Figura 7* la quale definisce una struttura contenente i puntatori all'istanza corrente della JVM che eseguirà il *Byte Code* di interesse e all'environment JNI che consente l'accesso alle funzioni della *Native Interface*.

```
#import <JavaVM/jni.h>

struct JavaKeys
{
    JavaVM* jvm;
    JNIEnv* env;
};
```

Figura 7 - Dichiarazione di una struttura contenete puntatori alla JVM ed al JNI environment.

Le principali API JNI sono le seguenti[15]:

- **jint JNI_CreateJavaVM(JavaVM **p_vm, JNIEnv **p_env, void *vm_args)** : carica e inizializza la Java Virtual Machine. Il thread che chiama questo metodo diviene quello principale. Richiede come parametri un puntatore alla locazione dove verrà salvata la struttura che rappresenta la JVM, un puntatore alla locazione dove memorizzare l'environment JNI per il thread principale ed un puntatore agli argomenti di inizializzazione della JVM creata. Ritorna 0 se si ha successo, un numero negativo in caso di fallimento.
- **jint AttachCurrentThread(JavaVM *vm, JNIEnv **p_env, void *thr_args)** : collega il thread corrente alla Java Virtual Machine. Richiede come parametri un puntatore ad un oggetto *JavaVM* rappresentante la JVM, un puntatore alla locazione dove verrà salvato l'environment JNI in caso di successo ed un puntatore agli argomenti del thread da collegare. Ritorna 0 se si ha successo, un numero negativo in caso di fallimento.
- **jint DetachCurrentThread(JavaVM *vm)** : scollega il thread corrente dalla Java Virtual Machine. Richiede come parametri un puntatore ad un oggetto *JavaVM*. Ritorna 0 se si ha successo, un numero negativo in caso di fallimento.
- **jclass FindClass(JNIEnv *env, const char *name)** : carica una classe definita localmente. Richiede come parametri un puntatore all'environment JNI ed il nome della classe da caricare. Ritorna un oggetto *jclass* in caso di successo, altrimenti *nil*. Il *Byte Code* della classe richiesta viene ricercato nelle directory e nei file .zip alla locazione specificata nella variabile di ambiente *CLASSPATH*.

- **void DeleteLocalRef(JNIEnv *env, jobject localRef)** : cancella il riferimento all'oggetto locale mantenuto in *localRef*. Come parametri prevede un puntatore all'environment JNI ed un riferimento ad un oggetto locale.
- **jmethodID GetMethodID(JNIEnv *env, jclass class, const char *name, const char *sig)** : ritorna l'identificativo di un metodo per l'istanza di una classe o interfaccia. Richiede come parametri un puntatore all'environment JNI, un oggetto *jclass* rappresentante la classe di interesse, il nome del metodo richiesto e la sua firma. Ritorna un oggetto *jmethodID* in caso di successo, altrimenti *nil* in caso di errore.
- **jobject NewObject(JNIEnv *env, jclass class, jmethodID methodID, ...)** : costruisce un nuovo oggetto Java. Questa funzione richiede come parametri un puntatore all'environment JNI, un oggetto *jclass* rappresentante la classe di interesse, un oggetto *jmethodID* che rappresenta l'identificativo di un metodo della classe *jclass* (quando si chiama questa funzione spesso è l'id del costruttore) ed un numero variabile di parametri che dipendono dall'oggetto che si vuole istanziare.
- **void CallVoidMethod(JNIEnv *env, jobject obj, jmethodID methodID, ...)** : chiama il metodo dell'oggetto *obj* identificato da *methodID* avente come tipo di ritorno void. Come parametri richiede un puntatore all'environment JNI, un oggetto *jobject* ed un oggetto *jmethodID*, rappresentanti l'oggetto ed il relativo metodo da chiamare, ed un numero variabile di parametri che dipendono dal metodo chiamato.

In *Figura 8* è mostrato un esempio di programma Objective-C che fa uso di JNI per chiamare una classe Java che si occupa di sommare due numeri interi.

```

if(keys.jvm != nil && keys.env != nil)
{
    if ((*keys.jvm)-->AttachCurrentThread(keys.jvm, (void**) &keys.env, NULL) == JNI_OK)
    {
        jclass javaTestCls = ((*keys.env)-->FindClass(keys.env, "Magic"));

        if (javaTestCls != nil)
        {
            NSLog(@"Accessed JVM");

            jmethodID constructID = ((*keys.env)-->GetMethodID(keys.env, javaTestCls, "<init>", "()V"));
            jobject javaTestObj = ((*keys.env)-->NewObject(keys.env, javaTestCls, constructID));
            jmethodID methID = ((*keys.env)-->GetMethodID(keys.env, javaTestCls, "sum2Integers", "(II)V"));

            if (javaTestObj == nil)
            {
                NSLog(@"Nil value");
            }
            else
            {
                ((*keys.env)-->CallVoidMethod(keys.env, javaTestObj, methID, num1, num2));
            }

            ((*keys.env)-->DeleteLocalRef(keys.env, javaTestCls));
        }
        else
        {
            NSLog(@"Not Accessed");
        }
    }
    (keys.jvm[0])-->DetachCurrentThread(keys.jvm);
}
else
{
    NSLog(@"Cannot contact \"Magic\" Java class");
}

```

Figura 8 - Esempio di programma Objective-C che fa uso di JNI

Nella *Figura 8* si osserva che dopo l'invio del messaggio ad *AttachCurrentThread*, si ha la ricerca della classe Java *Magic* tramite il metodo *FindClass*. Se questa operazione ha successo, si procede identificando il costruttore da chiamare, si istanzia un oggetto Java di classe *Magic* e si identifica il metodo di interesse tramite la chiamata a *GetMethodID*. Se anche questa parte ha successo, si procede all'invocazione vera e propria del metodo per la somma degli interi e alla rimozione del riferimento all'oggetto istanziato (*CallVoidMethod*, e *DeleteLocalRef*). Infine si rimuove anche il riferimento del thread corrente alla JVM tramite il metodo *DetachCurrentThread*.

2.3.2 Java Native Access

JNA o *Java Native Access* è una libreria che permette ad un applicativo Java di accedere a librerie native condivise, senza la necessità di usare la *Java Native Interface*. JNA è stata progettata per garantire l'accesso a codice nativo nel modo più naturale possibile, minimizzando gli sforzi necessari da parte del programmatore. JNA usa una libreria nativa chiamata *foreign function interfece*, meglio nota come *libffi*, per invocare dinamicamente codice nativo[14]. JNA è disponibile per Mac OS X, Windows e Linux.

Utilizzando un bridge verso Objective-C sviluppato da *Weblite*[16] che sfrutta JNA è possibile scrivere un programma Java basato sul paradigma di programmazione Client-Proxy che utilizzi oggetti Objective-C[17]. Lo scopo è quello di delegare al proxy e a JNA l'interazione con il runtime di Objective-C in modo che la comunicazione tra questo e Java sia il più trasparente possibile.

L'annotazione custom *@Msg* permette al bridge di specificare il metodo a cui inviare il messaggio (tramite l'attributo *selector*) e la tipologia della firma del metodo (tramite l'attributo *signature*).

Ad esempio, il metodo Java mostrato in *Figura 9* è utilizzato per inviare al corrispondente metodo Objective-C “*allocateAnObject:*” un messaggio per l’allocazione di un oggetto di classe *toAllocate*.

```
@Msg(selector="allocateAnObject:", signature="v:@")
public Proxy allocateAnObject(String toAllocate)
{
    Proxy obj = this.c.sendProxy(toAllocate, "alloc");
    obj = obj.sendProxy("init");
    return obj;
}
```

Figura 9 - Invio di un messaggio per l’allocazione di un oggetto Objective-C

2.4 Approcci a confronto

JNA è più flessibile di JNI ma ha dei limiti: se con un bridge che usa Java Native Access è possibile utilizzare oggetti Objective-C in un programma Java, a causa delle differenze nel runtime tra Swift ed Objective-C non è possibile sfruttare lo stesso bridge per comunicare con oggetti Swift. Attualmente non è disponibile uno strumento che lo permetta principalmente per due motivi: Swift ha solo un anno di vita ed è proprietario Apple.

JNI si rivela utile per stabilire una comunicazione da Objective-C a Java ma si dimostra inutilizzabile per integrare un programma Java in un applicativo iOS poiché sui device Apple non è presente una JVM.

I due approcci JNI e JNA non sono adatti per gli scopi di questa tesi in quanto si basano sull'utilizzo diretto di codice nativo, mantenendo però separati gli ambienti d'esecuzione che si desidera far comunicare.

Una soluzione alternativa è offerta dalla possibilità di tradurre il *Byte Code* Java in codice nativo per il runtime iOS così da bypassare la necessità di una *virtual machine*. Questo approccio prevede che l'applicazione sia scritta completamente in Java e che il codice sia opportunamente marcato con annotazioni che forniscono i *bindings* necessari a tempo di compilazione per l'integrazione con l'ambiente nativo, l'unico presente a runtime.

Per quanto riguarda l'utilizzo di Swift per la progettazione di software eseguibile al di fuori di Cocoa/Cocoa Touch, è possibile applicare una metodologia analoga che prevede l'utilizzo di una opportuna libreria per la traduzione di codice Swift verso il runtime desiderato.

In questa logica, nella tesi si sono approfondite le potenzialità di due strumenti: *RoboVM* e *Fire*, per la traduzione del codice e per la realizzazione di applicativi nativi.

3 Strumenti per la traduzione automatica di codice

3.1 RoboVM

La finalità alla base dello strumento è quella di rendere disponibile Java anche per l'ambiente iOS e Mac OS X favorendo lo sviluppo di applicazioni commerciali in un linguaggio diverso da Objective-C e Swift. RoboVM rende possibile l'utilizzo di Java in quanto traduce il linguaggio in codice nativo eseguibile per il runtime Objective-C[18].

RoboVM è un software rilasciato sotto licenza Apache 2.0 ad eccezione del compilatore e del plug-in Eclipse i quali sono soggetti alla licenza GNU GPLv2 poiché dipendono da altri strumenti. Da giugno 2015 è disponibile anche *RoboVM Studio*, un IDE proprietario che include il tool di traduzione.

Sia dal plug-in Eclipse che da RoboVM Studio è possibile implementare la parte grafica dell'applicazione rimanendo all'interno dell'ambiente Java utilizzando le librerie del tool che fungono da bridge verso le API di Cocoa/Cocoa Touch. Per quanto riguarda la struttura del codice, cioè quali classi sono presenti a runtime e con quali funzioni, lo strumento rispecchia appieno la struttura presente in un progetto Xcode (sottoparagrafo 3.1.2). La traduzione che viene effettuata prendendo in ingresso i file *.class* derivati da una prima compilazione Java genera come output direttamente il file eseguibile nativo per architetture ARM o x86 (sottoparagrafo 3.1.1).

Per l'installazione e l'utilizzo del plug-in Eclipse è necessario disporre di:

- Java SE JDK 1.7 o superiore
- Mac OS X 10.10 o superiore
- Xcode 6 o superiore

se si è optato per il download di RoboVM Studio, sarà l'IDE stesso al primo avvio a guidare lo sviluppatore nella configurazione del proprio ambiente di lavoro.

3.1.1 Architettura generale

Come schematizzato in *Figura 10*, RoboVM mette a disposizione due livelli di astrazione logica: il primo si occupa di rendere disponibili in Java le API native di iOS effettuando il *wrapping* delle librerie, il secondo fornisce invece i *bindings* necessari per le chiamate al codice nativo, in particolare facilitando l'interazione con le classi Objective-C direttamente dall'ambiente Java. Per questo scopo vengono rese disponibili delle annotazioni ad-hoc che permettono di segnalare al compilatore quali metodi Java fungono da *event handler* oppure quali metodi sono *setter* di attributi privati (annotazioni `@IBAction`, `@IBOutlet`).

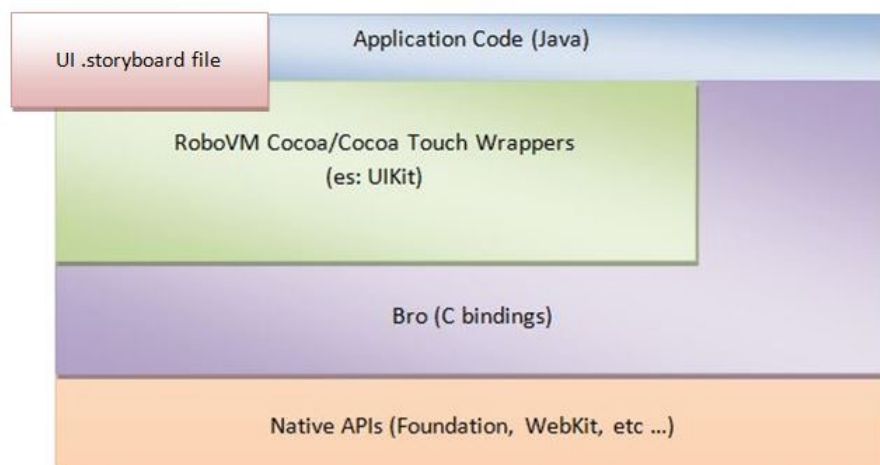


Figura 10 - Architettura RoboVM

RoboVM fornisce un compilatore AOT, *Ahead-Of-Time*, basato su una LLVM, *Low Level Virtual Machine*, che contiene i bindings per C e un bridge per Objective-C (*Bro*). La compilazione AOT viene utilizzata in quanto non è possibile portare una macchina virtuale Java in ambiente iOS per via dei vincoli imposti da Apple che obbligano l'utilizzo dell'ambiente Cocoa/Cocoa Touch. Con l'approccio AOT, il *Byte Code* Java viene compilato in codice nativo prima dell'esecuzione, in particolare durante l'installazione dell'app. In questo modo è possibile eseguire direttamente il compilato che non necessita di interpretazioni intermedie come invece accade per il *Byte Code*. Così facendo non è necessario interpretare parti del programma durante la sua esecuzione e di conseguenza si possono migliorare le prestazioni e la reattività dell'app.

Per quanto riguarda la compatibilità verso Swift, RoboVM si integra con le classi marcate *@objc* o derivanti direttamente da *NSObject*, ma lo strumento non offre una traduzione diretta verso il runtime Swift. Di conseguenza gli applicativi prodotti da RoboVM risultano compilati solamente per l'ambiente Objective-C.

3.1.2 Metodologia di sviluppo

Per lo sviluppo di una *app* iOS basata su Java è possibile utilizzare Eclipse assieme al plug-in RoboVM in quanto i due strumenti sono ben integrati ed è possibile usare contemporaneamente sia la documentazione fornita dall'IDE che il supporto di RoboVM al debug ed al deploy dell'applicativo.

Se non si vuole utilizzare Eclipse come programma di sviluppo è possibile utilizzare RoboVM Studio: questo strumento permette la realizzazione contemporanea sia di applicativi iOS che Android. Degna di nota è la possibilità di creare un progetto ibrido diviso in tre sezioni: una per la GUI Android, una per la GUI iOS e l'ultima per il codice applicativo condiviso. In questo modo se si dispone dei simulatori per entrambe le piattaforme è possibile compilare lo stesso applicativo per una o per l'altra e testarlo.

Siccome si è scelto iOS come piattaforma target, si è deciso di approfondire l'utilizzo del plug-in RoboVM per Eclipse in quanto non si ha necessità del supporto ad Android.

La metodologia di sviluppo tramite il plug-in Eclipse RoboVM prevede la presenza sia dei file Java contenenti la logica applicativa sia dei file *.jar*, specifici di RoboVM, che rendono possibile la traduzione verso Objective-C e realizzano gli opportuni bindings. In questo modo è possibile riutilizzare codice Java già sviluppato sia che si disponga dei sorgenti sia che si disponga di file *.jar* e/o *.class*. La *Figura 11* schematizza quanto detto per i componenti di un progetto RoboVM in Eclipse.

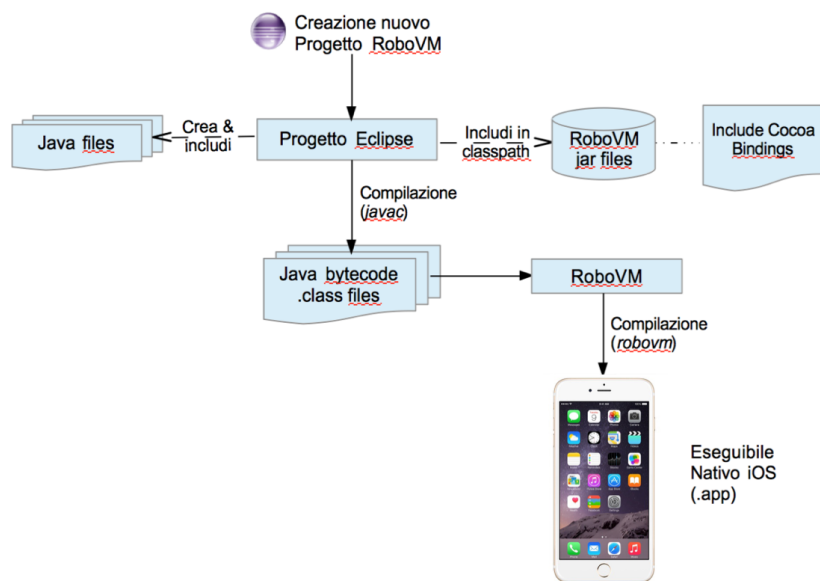


Figura 11 - Metodologia di sviluppo tramite plug-in Eclipse RoboVM

Come template di progetto è possibile scegliere tra un applicativo iOS o console; scegliendo iOS è possibile decidere se si vuole realizzare un programma che utilizzi i file *.storyboard* per la grafica o se si preferisce procedere con un progetto vuoto. Per la realizzazione della GUI si può usare in maniera completamente indipendente Xcode e poi copiare nel progetto i file generati oppure, se si dispone della relativa licenza, è possibile utilizzare l'interface builder messo a disposizione dallo stesso plug-in. In questo caso Xcode verrà lanciato da Eclipse ed i file saranno già inclusi nel progetto.

La struttura di un progetto Eclipse RoboVM è simile a quella mostrata in Figura 12. La cartella *src/main/java* racchiude i file sorgenti Java ed obbligatoriamente prevede la presenza di due file, uno contenente il metodo statico *main* che funge da entry-point dell'applicativo ed il metodo *didFinishLaunching*, eseguito al termine del caricamento dell'app (in Figura 12 è il file *Test.java*), l'altro contenente tutti i metodi di istanza del ViewController (*MyViewController.java*). Tutti gli applicativi iOS usano come pattern architetturale MVC, il che significa che la parte centrale dell'applicativo è basata sul view

controller il quale gestisce gli input ricevuti tramite la GUI ed altera in base a questo i dati di model. Il controller provvede anche ad aggiornare la view mostrando i risultati dell'elaborazione degli input impartiti dall'utente.

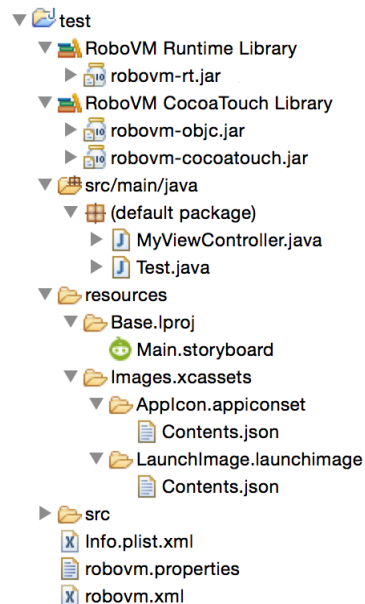


Figura 12 - Struttura di un progetto Eclipse basato su RoboVM con storyboard

Oltre alla inclusione automatica da parte di RoboVM dei file .jar necessari alla traduzione verso Objective-C, in *Figura 12* si nota la presenza della cartella *resources* la quale contiene due sottodirettori, uno per il file *Main.storyboard* e l'altro per il set delle icone usate in tutto il ciclo di vita dell'applicativo. Lo storyboard in *Base.lproj* non è altro che un complesso file XML contenente tutte le informazioni relative alla GUI: in esso non sono solamente indicati i componenti grafici di cui eseguire il rendering, ma anche tutte le loro proprietà ed eventuali vincoli ad event handler, ovvero compare esplicitamente il nome del metodo che avrà il compito di gestire gli eventi scatenati dal componente grafico in questione. In generale questo file viene scritto in automatico da Xcode ed è consigliabile evitare di modificarlo manualmente, eccetto che per semplicissimi hack, per evitare comportamenti inaspettati o malfunzionamenti della GUI. In *Images.xcassets*, *AppIcon.appiconset* contiene un file in formato JSON il quale dichiara un array di oggetti JSON rappresentante tutte le icone utilizzate nell'applicativo. La cartella contiene anche i

file delle immagini da mostrare. In *LaunchImage.launchimage* si ha un file JSON simile al precedente ma per le immagini da visualizzare al boot dell'app. Anche questi file sono generati in automatico e valgono le stesse considerazioni fatte per *Main.storyboard*.

Infine i file *Info.plist.xml*, *robovm.properties* e *robovm.xml* contengono delle informazioni di servizio, come il nome dell'applicativo, la sua versione, il nome della classe contenente il metodo main, eventuali librerie da includere esplicitamente nel processo di compilazione e traduzione di RoboVM, il codice identificativo dell'app e della firma digitale dello sviluppatore ed altro ancora.

Per quanto riguarda il collaudo, è possibile utilizzare gratuitamente il simulatore di dispositivi iOS oppure, se si desidera testare l'applicazione su un device fisico, da Xcode è possibile generare un certificato provvisorio per firmare digitalmente l'app.

Una volta ultimato l'applicativo, è possibile procedere alla creazione dell'archivio *.ipa* per la pubblicazione su *App Store*, bisogna però possedere un *Apple ID* registrato al programma developer di Apple ed aver sottoscritto il piano di distribuzione a pagamento.

3.2 Fire

Se da un lato RoboVM apre il mondo di Java all'universo Apple, *Elements*, *Fire* e *Silver*[19] permettono di fare l'esatto contrario (sottoparagrafi 3.2.1 e 3.2.2), infatti se con RoboVM è possibile scrivere codice Java e compilarlo per il framework Cocoa/Cocoa Touch, attraverso l'IDE *Fire* (sottoparagrafo 3.2.3) è possibile scrivere codice Swift e compilarlo per ottenere un file *.jar* eseguibile da una JVM.

Fire è prodotto dall'azienda RemObjects Software che sviluppa tool che permettano di scrivere programmi eseguibili in più ambienti senza che questi siano *cross-platform*. Lo scopo è fornire la soluzione migliore per ogni ambiente specifico, piuttosto che realizzare una soluzione mista che, con compromessi, permetta di sviluppare software multiplatforma. Secondo la filosofia di RemObjects, una applicazione è vincente quando è in grado di dialogare tramite API native con il runtime che la ospita e non attraverso librerie intermedie. Con questo approccio lo sviluppatore può scrivere codice nel linguaggio che preferisce tra quelli disponibili in Fire, ma l'applicativo risultante è compilato ed eseguibile solamente per l'ambiente scelto. Per rendere l'applicazione portabile bisognerà quindi creare un nuovo progetto basato sul runtime target e ricompilarlo con Fire.

3.2.1 Elements

Elements è il compilatore creato ad hoc da RemObjects che permette di compilare applicativi per le maggiori piattaforme come Java/Android, iOS/Mac e .NET/Windows. E' possibile scegliere tra tre linguaggi: Oxygene, C#, Silver, tutti compatibili con gli ambienti suddetti.

Oxygene, proprietario di RemObjects, si basa sulle fondamenta di Object Pascal presentandosi come un linguaggio ad oggetti moderno[4].

C# è al 100% C# ovvero RemObjects assicura il completo supporto del linguaggio e il suo utilizzo è consigliato ai programmatori abituati a lavorare con il framework .NET.

Silver è una estensione di Swift usando la quale è possibile scrivere direttamente codice Swift e renderlo eseguibile per .NET, Java/Android e Cocoa (sottoparagrafo 3.2.2).

Elements offre non solo la possibilità di sviluppare codice eseguibile per ambienti differenti, ma anche il supporto al debug e al deploy direttamente dall'IDE scelto. Se si lavora con Windows si può sfruttare l'integrazione di Elements con Visual Studio. Nella versione 2013 di quest'ultimo è infatti possibile scaricare un componente tramite il quale usare Elements, oppure dal sito di RemObjects è disponibile al download una versione di Visual Studio già configurata per utilizzare il compilatore. Se invece si usa Mac OS X è possibile scaricare Fire.

3.2.2 Silver e Swift

Con Silver è possibile usare Swift per scrivere codice utilizzabile in ambienti che sfruttano le API di .NET, Java e Cocoa. Offre pieno supporto agli ambienti menzionati ma resta non *cross-platform* in quanto consente al programmatore di sviluppare programmi specifici per ciascuna piattaforma considerata. In pratica permette di utilizzare codice Swift in progetti che usano componenti non basati su Swift.

Il compilatore Silver, incluso in Elements, offre anche supporto al debug e al deploy delle applicazioni direttamente attraverso l'IDE Fire, o Visual Studio se si lavora con Windows, facilitando il processo di sviluppo del software.

Attualmente la release 8.1 di Elements supporta appieno tutte le funzionalità di Swift 1.2. Swift 2.0 sarà completamente supportato con la release Elements 8.2 prevista per la fine del 2015.

Essendo una estensione, Silver offre alcune funzionalità non presenti nella attuale versione di Swift come la gestione delle eccezioni, l'uso di iteratori, classi parziali, eventi, funzioni inline, *aspect-oriented programming* (AOP). Laddove sono necessarie parole chiave

addizionali, come nel caso dell'*exception handling* mostrato in *Figura 13*, RemObjects ha scelto di essere conforme alla convenzione dei linguaggi C-style che consiste nell'anteporre due underscore (“__”) all'identificatore scelto per evitare conflitti di nome.

```
__try
{
    if (!solveGoal(goal))
    {
        return true
    }
    else
    {
        return false
    }
}
__catch E: Exception
{
    println("Error -> \"(E.Message) occurred (E2)\"")
}
```

Figura 13 - Esempio di gestione delle eccezioni

3.2.3 L'IDE

Fire è un IDE per lo sviluppo di applicativi tramite i linguaggi supportati da Elements. Attualmente è in stato di avanzata beta (ultima versione disponibile è la 8.1.83.1761 del 13/05/2015) abbastanza stabile ma ancora rigida dal punto di vista dell'interfaccia grafica. Fire non è basato su framework IDE preesistenti ma è stato interamente sviluppato *from scratch* da RemObjects e scritto principalmente in C#. L'IDE è compatibile con OS X Mavericks e Yosemite e per eseguirlo basta effettuare il download dal sito di RemObjects e lanciare il file .app scaricato. Gli unici requisiti sono legati alla piattaforma verso la quale si vuole sviluppare: se si sceglie come target Java e Android è necessario avere installato Java 7 o 8 SDK o Android SDK, per quanto riguarda iOS e applicazioni Mac è richiesta l'installazione di Xcode 6 o successivi, per .NET è richiesta l'installazione di Mono 3.0 o successivi.

Una caratteristica di Fire è Oxidizer, una utility di traduzione che permette di importare nel proprio progetto file sorgenti scritti in C#, Java, Objective-C o *Delphi* e di tradurli automaticamente in Oxygene, C# o Swift[20]. La *Figura 14* chiarisce le conversioni possibili tra i linguaggi citati.

From	To
C#	Oxygene
C#	Swift
Java	Oxygene
Java	C#
Java	Swift
Objective-C	Oxygene
Objective-C	C#
Objective-C	Swift
Delphi	Oxygene

Figura 14 - Conversioni possibili usando Oxidizer

4 tuProlog: un caso di studio multiplatforma

tuProlog[1] è un sistema Prolog leggero per infrastrutture e applicazioni distribuite, sviluppato presso il DISI dell'Università di Bologna, intenzionalmente basato su di un core minimale per essere poi configurato in un secondo momento sia staticamente che dinamicamente tramite il caricamento di librerie di predicati. tuProlog supporta nativamente la programmazione multi-paradigma realizzando una integrazione tra Prolog ed i più comuni linguaggi di programmazione object-oriented come Java, per la relativa versione dell'engine, e tutti i linguaggi del framework .NET come C# per l'analoga versione del sistema.

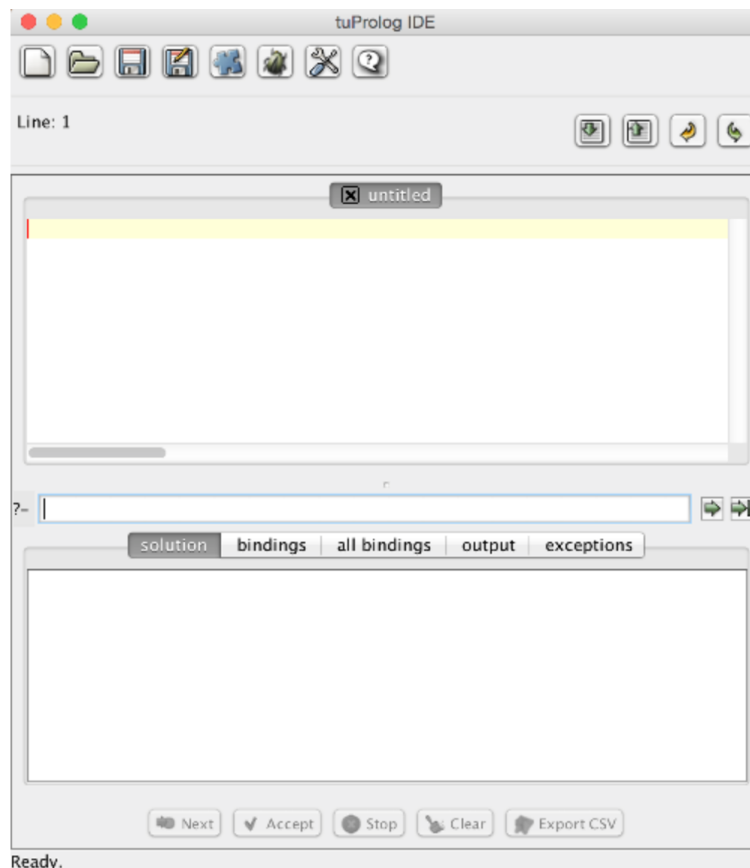


Figura 15 - tuProlog IDE

tuProlog è stato progettato per essere uno dei componenti base delle infrastrutture e applicazioni internet; questo scopo ha determinato le sue principali caratteristiche come la semplicità di deploy, la leggerezza, la configurabilità dinamica, l'alta integrazione con Java e la relativa facilità di interoperabilità. Per installare tuProlog è necessaria la presenza della Java VM standard ed il download del file .jar eseguibile, disponibile al seguente link:

<https://bitbucket.org/tuprologteam/tuprolog/downloads>

Lanciato il file .jar è possibile interagire con tuProlog attraverso l'IDE mostrato in *Figura 15*.

4.1 tuProlog: integrazione in un applicativo iOS

Si desiderano realizzare due app iOS, una per iPhone ed una per iPad, che permettano di interrogare tuProlog e che soddisfino i seguenti requisiti:

- Scrittura dell'applicazione in linguaggio Java;
- Supporto completo in ambiente Objective-C alle librerie di interfacciamento Java/Prolog utilizzate da tuProlog;
- Esecuzione non vincolata alla presenza di una Java Virtual Machine;
- Interfaccia grafica personalizzata ad-hoc per i target device.

Per soddisfare i requisiti descritti si può utilizzare il plug-in Eclipse RoboVM (sottoparagrafo 3.1.2) assieme ad Xcode in quanto questi strumenti permettono di riutilizzare codice già sviluppato (tuProlog in questo caso) insieme all'interfaccia grafica progettata *ex novo*. In questo modo è possibile continuare ad utilizzare Java come linguaggio di programmazione e si possono tradurre in Objective-C le librerie di interfacciamento Java/Prolog, contenute nel file *tuprolog.jar*, ottenendo una soluzione blended Java/Objective-C/Prolog. Operando in questo modo si svincola l'esecuzione degli applicativi dalla presenza di una JVM. Il processo di sviluppo seguito per la progettazione e realizzazione dell'applicazione multiplatforma (relativamente ad iOS) è quello schematizzato in *Figura 16*.

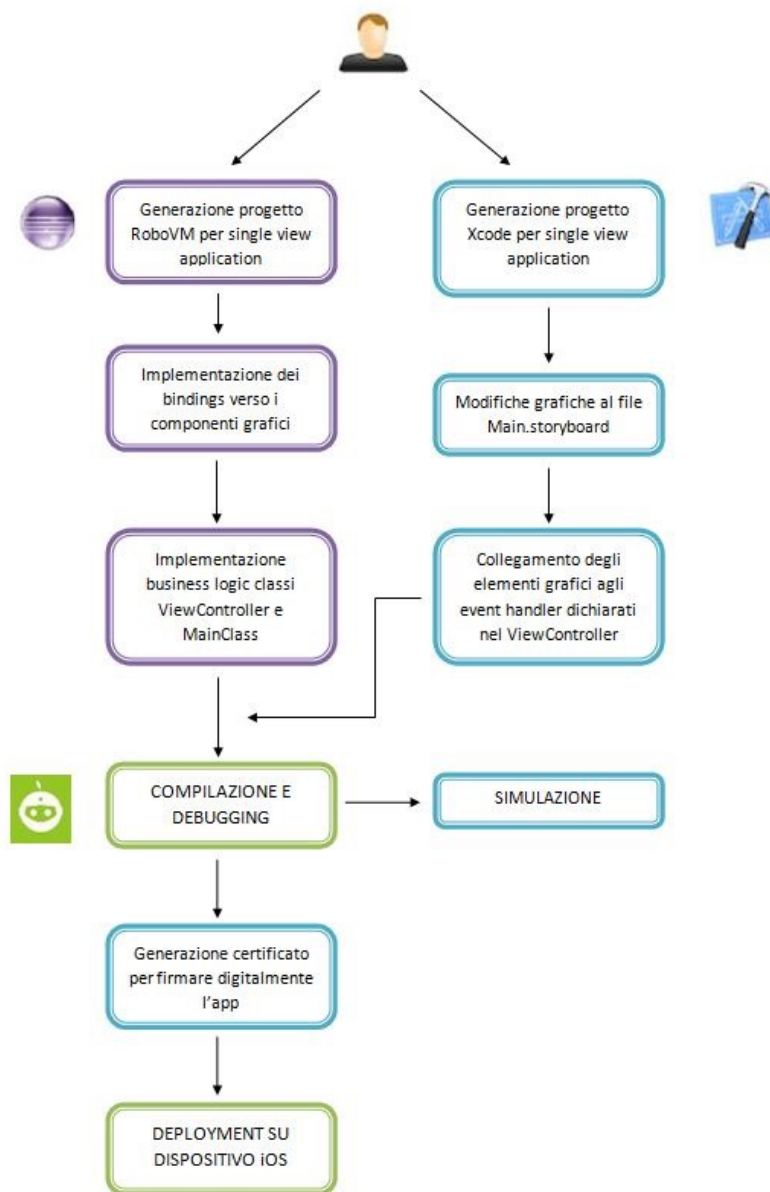


Figura 16 - Sviluppo app iOS tuProlog tramite plug-in Eclipse RoboVM

La necessità di progettare una GUI specifica per i dispositivi iOS è stata risolta creando due progetti: uno per il disegno della grafica tramite Xcode (in blu in *Figura 16*), l'altro per l'implementazione in Java degli event handler associati ai componenti grafici (in viola la parte Eclipse). RoboVM (in verde) ha il compito di eseguire la compilazione e traduzione verso il runtime Objective-C della applicazione scritta in Java e di farne il deployment sul device fisico di interesse. Segue una descrizione dettagliata dei progetti Xcode (sottoparagrafo 4.1.1) ed Eclipse (sottoparagrafo 4.1.2).

4.1.1 Progetto Xcode

Per lo sviluppo della grafica si utilizza Xcode. Si è creato un nuovo progetto selezionando come template iOS la *Single View Application* e si è selezionato Swift come linguaggio, in quanto risulta più semplice ed intuitivo rispetto a Objective-C. Questa decisione non ha conseguenze sul progetto RoboVM in quanto soltanto il file XML *Main.storyboard* prodotto da Xcode sarà da importare in Eclipse, permettendo al traduttore di generare successivamente una applicazione grafica basata su Objective-C.

Tramite l'interface builder di Xcode è possibile disegnare la GUI selezionando i componenti necessari (ad esempio: *UIButton*, *UILabel*, *UITextView*...). Una volta ultimata, è possibile esplicitare nella relativa classe *ViewController.swift* le *IBOutlet* ed *IBAction* richiamabili in Java per collegare i corrispondenti setter di campi privati ed i metodi che fungono da event handler. Il procedimento è lo stesso sia che si consideri la grafica di iPhone che di iPad. Terminato lo sviluppo si è di fronte alla situazione mostrata in *Figura 17*.

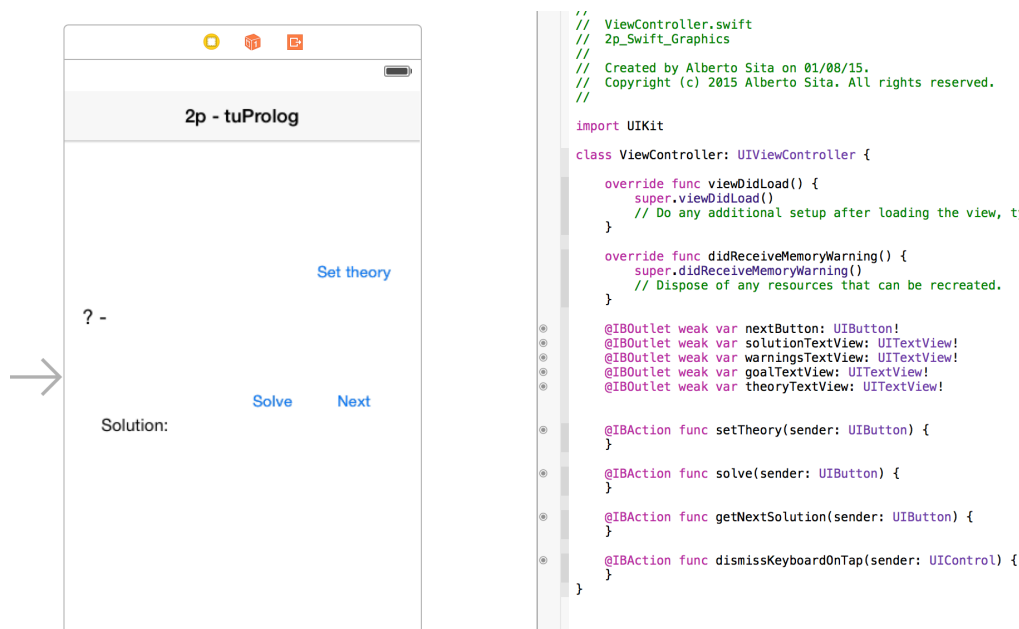


Figura 17 - Sviluppo grafica app *tuProlog* per iPhone

4.1.2 Progetto Eclipse

Il seguente procedimento è valido sia che si consideri la versione dell'app per iPhone che per iPad.

In Eclipse si crea un nuovo progetto RoboVM per una applicazione iOS *Single View* basata su storyboard e si importa il file *Main.storyboard* creato precedentemente con Xcode. Aprendo questo file con l'editor di testo, bisogna eliminare gli attributi del tag `<viewController> customModule = "nome-modulo-progetto"` e `customModuleProvider = "target"` altrimenti Eclipse non riesce a creare correttamente il compilato Objective-C. Tutti i nomi degli event handler contenuti in *Main.storyboard* vanno privati dei `:` finali perché devono corrispondere con i nomi dei metodi usati in Java. Successivamente si importa il file *tuprolog.jar* e, dopo aver controllato i file di *properties* di RoboVM contenenti le icone utilizzate, l'identificativo dell'app e le librerie da includere tramite tag `<forceLinkClass>`, è possibile concentrarsi sulla classe *ViewController.java* che implementa la logica del controller.

La classe *ViewController* eredita da *UIViewController* ed è suddivisa in tre sezioni: i setter pubblici di componenti grafici privati, marcati dall'annotazione custom `@IBOutlet`, i gestori degli eventi scatenati dalla GUI, marcati da `@IBAction` ed i metodi di istanza che costituiscono la logica applicativa del controller o fanno override di metodi della superclasse. I nomi dei metodi setter devono rispecchiare quelli dei componenti grafici presenti in *ViewController.swift* (vedi progetto Xcode sottoparagrafo 4.1.1) con anteposto il termine *set*, mentre i gestori degli eventi devono avere lo stesso nome esplicitato nel file XML *Main.storyboard*. I metodi *viewDidLoad* e *shouldAutorotate* permettono rispettivamente di compiere azioni subito dopo che la grafica è stata caricata e di bloccare o attivare la sensibilità alla rotazione del dispositivo.

Gli altri metodi implementati sono specifici dell'applicativo e servono per interagire con l'engine Prolog ovvero per impostare una teoria o risolvere un goal. Degno di nota è il metodo *init_prolog* il quale ha il compito di istanziare, se non presente, un oggetto di tipo *Prolog* che raffigura il motore a cui rivolgere le proprie query.

4.1.3 Simulazione e collaudo



Figura 18a - Versione di tuProlog per iPad

In *Figura 18a* è mostrata la versione di tuProlog per iPad. In particolare si osserva come interrogare il motore prolog per ottenere una lista dopo aver impostato una teoria. Analogamente in *Figura 18b* è mostrata la versione di tuProlog per iPhone.

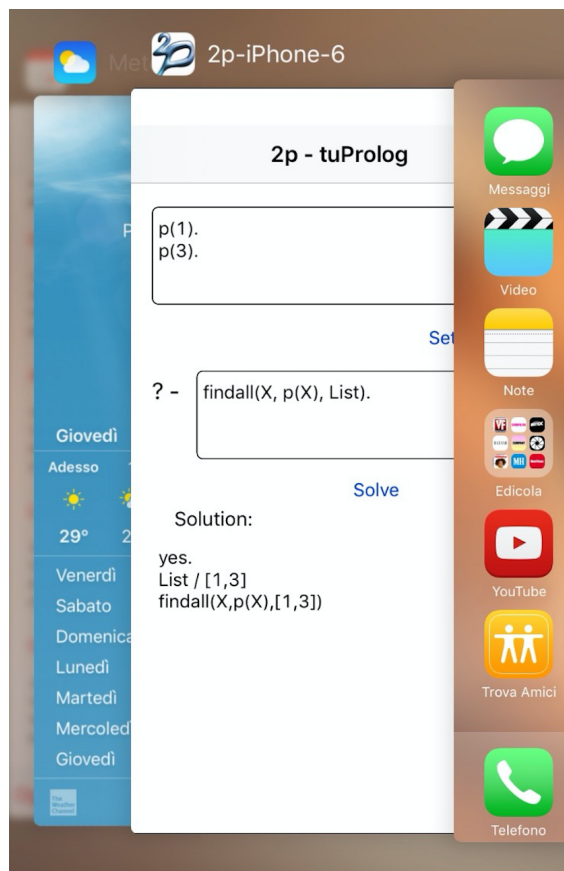


Figura 18b - Versione di tuProlog per iPhone

Per quanto riguarda il collaudo, si è generato da Xcode un certificato digitale temporaneo per firmare le app ed effettuare il deploy sui dispositivi fisici tramite il plug-in Eclipse RoboVM. I test sono stati effettuati su *iPhone 5/5S/6* ed *iPad Air/Air2/Mini* con *iOS 8.1/8.4/9.0*. Su tutti i dispositivi coinvolti le rispettive versioni dell'app si sono dimostrate stabili e prive di rallentamenti.

4.2 tuProlog e Swift tramite Fire

Si vuole ora discutere di come realizzare una applicazione a riga di comando che soddisfi i seguenti requisiti:

- Scrittura del codice in linguaggio Swift;
- Integrazione con tuProlog;
- Esecuzione dell'applicativo in un ambiente diverso da Cocoa, ad esempio Java.

Usando Fire si ottiene il *blending* di linguaggi tra Java, Swift e Prolog. In particolare si ha la possibilità di utilizzare tuProlog da codice Swift. Seguono i passi necessari alla realizzazione dell'applicazione.

Step 1

Dopo aver scaricato e installato la preview gratuita di Fire ed Elements al seguente link:

<http://www.elementscompiler.com/elements/download.aspx>

si può procedere alla scelta del template di progetto. Aprendo Fire, si seleziona *Swift* come linguaggio di programmazione, *Java* come piattaforma target e *Console Application* come tipo di template. Infine *OK* per confermare e procedere.

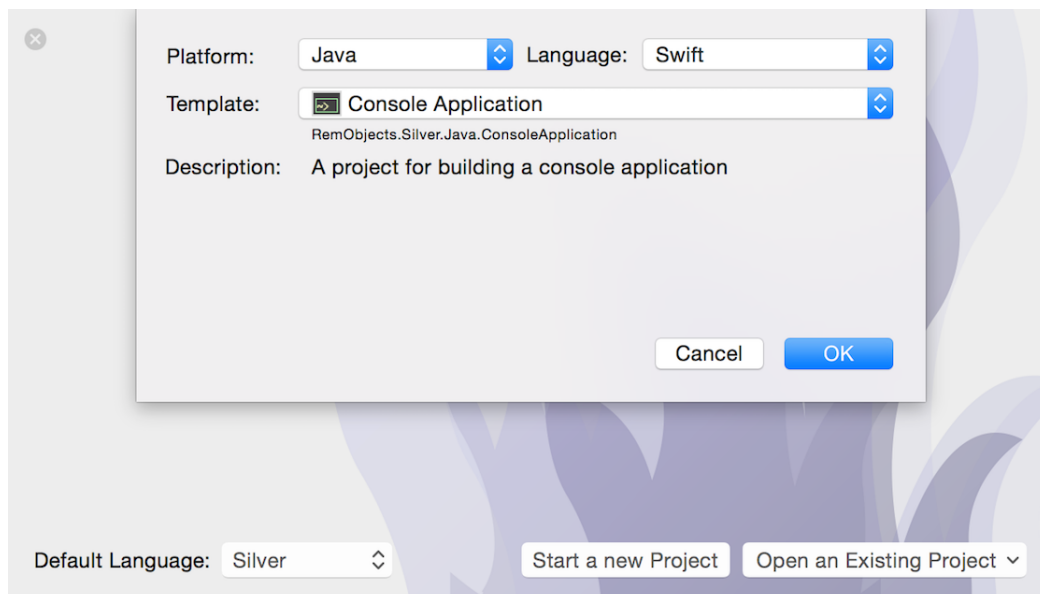


Figura 19 - Configurazione iniziale di un progetto Fire

Anche se semplice, questa prima operazione (*Figura 19*) pone dei forti vincoli di progetto: scegliendo come piattaforma Java si richiede a Fire di utilizzare le API native di Java e di compilare il codice sorgente per questo specifico ambiente di esecuzione.

Step 2

A questo punto, il progetto Fire risulta correttamente configurato. Per poter interagire con tuProlog occorre importare ed includere nelle risorse del progetto il file `tuprolog.jar`. Per fare ciò si seleziona la voce *References* dell'explorer di progetto e si aggiunge il file `.jar` selezionando la sua locazione su disco. Dopo questa operazione è possibile iniziare a scrivere codice Swift nel file *Program.swift* creato di default da Fire, aggiungendo eventualmente altri file sorgente al progetto. Ora, per usare tuProlog tramite console, è necessario importare i seguenti package tramite clausola `import`: *alice.tuprolog* (contenente tutte le risorse Prolog di tuProlog), *java.util* e *java.io*.

Osservando il codice in *Figura 20*, disponibile integralmente in appendice, è evidente la potenza espressiva di Fire: la funzione *init_Prolog* è scritta in Swift puro ma permette di chiamare la funzione *Prolog* (contenuta in *alice.tuprolog*) che restituisce una istanza di tipo *Engine*, ovvero il vero motore Prolog. Si ha quindi visibilità delle classi Java contenute nel package importato e la possibilità di chiamarle scrivendo codice Swift.

Le API di tuProlog *setTheory*, *solve*, *solveGoal*, *solveInfoToString* e *getNextSolution* sono implementate in Swift e permettono di interagire con l'engine Prolog consentendo di impostare una teoria, di risolvere un goal e di ottenere le soluzioni proposte dal motore. Per esempio in *Figura 20* si vede l'implementazione della funzione *setTheory*.

```

func init_Prolog()
{
    if (Engine == nil)
    {
        Engine = Prolog()
        println("\ntuProlog system - release " + Prolog.getVersion() + "\n")
    }
}

func setTheory(theory: String)
{
    if (theory != nil && theory != "")
    {
        //IL BLOCCO TRY CATCH È UNA ESTENSIONE DEL LINGUAGGIO DI REMOJECTS
        {
            Engine.setTheory(Theory(theory))
            println("Theory set!")
        }
        __catch E: InvalidTheoryException
        {
            println("Error -> \"(E.Message) Occurred for an invalid theory! (E1)\"")
        }
        __catch E: Exception
        {
            println("Error -> \"(E.Message) (E2)\"")
        }
    }
    else
    {
        println("WARNING: Theory is empty")
    }
}

```

Figura 20 - Funzioni *init_Prolog()* e *setTheory()*

Infine la funzione *Program* svolge il ruolo di costruire l'interazione a console tra l'utente finale e l'engine Prolog. Rilevanti sono i seguenti oggetti per l'input da console:

```

var reader: InputStreamReader = InputStreamReader(System.`in`)

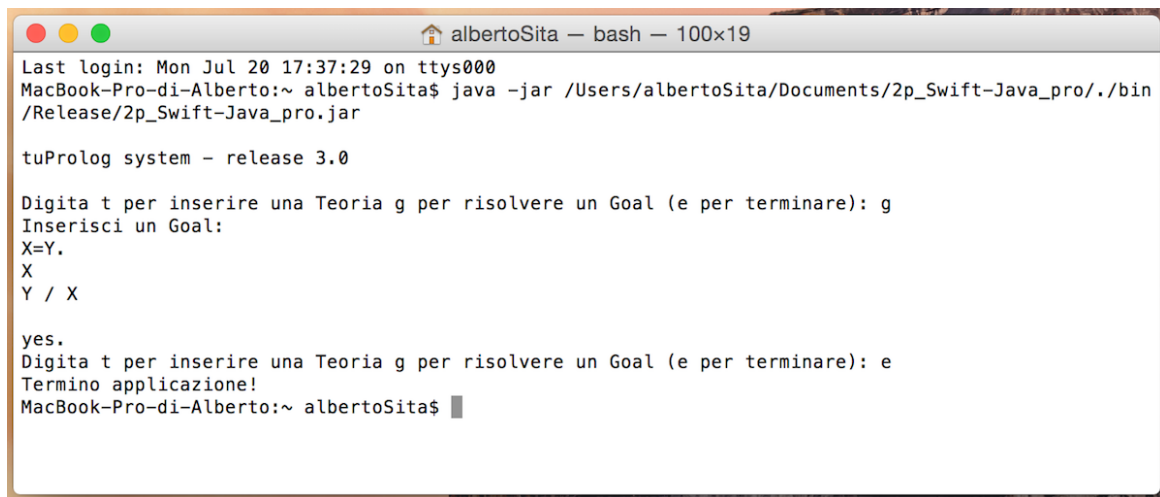
var bReader: BufferedReader = BufferedReader(reader)

```

in quanto Fire permette di istanziarli con la sintassi Swift mantenendo omogeneo l'aspetto del programma e nascondendo la dipendenza da Java. Questo esempio di codice non è riutilizzabile in un altro progetto Fire basato su Swift e rivolto a una piattaforma diversa da Java perché usa oggetti tipici di questa.

Step 3

Per collaudare, direttamente dall'IDE sotto il menù *Project*, è possibile selezionare la voce *Run w/o Debugging* che permette di lanciare l'applicazione in un terminale dedicato. Se invece si sceglie l'opzione *Run* non si ha a disposizione un terminale.



```
albertoSita - bash - 100x19
Last login: Mon Jul 20 17:37:29 on ttys000
MacBook-Pro-di-Alberto:~ albertoSita$ java -jar /Users/albertoSita/Documents/2p_Swift-Java_pro/./bin/Release/2p_Swift-Java_pro.jar

tuProlog system - release 3.0

Digita t per inserire una Teoria g per risolvere un Goal (e per terminare): g
Inserisci un Goal:
X=Y.
X
Y / X

yes.
Digita t per inserire una Teoria g per risolvere un Goal (e per terminare): e
Termino applicazione!
MacBook-Pro-di-Alberto:~ albertoSita$
```

Figura 21 - Esecuzione applicativo

L'immagine in *Figura 21* mostra come Fire chiami Java e lanci il file .jar eseguibile prodotto dalla compilazione. Se si dispongono di tutti i file .jar presenti nel progetto, ovvero le risorse incluse di default da Fire e le risorse aggiunte dallo sviluppatore, si può eseguire l'applicativo realizzato invocandolo direttamente da riga di comando. Ciò garantisce flessibilità in quanto gli unici requisiti per l'esecuzione sono la presenza di Java 7 o 8 e che i file .jar eseguibili siano tutti allo stesso livello nell'albero del filesystem.

5 Conclusioni

La filosofia alla base delle compagnie RoboVM e RemObjects è simile: entrambe hanno come scopo quello di realizzare applicativi nativi ottimizzati per la piattaforma su cui devono eseguire e di permettere lo sviluppo di software in linguaggi diversi da quelli tipici del runtime considerato, dando luogo ad applicativi multi-linguaggio e multi-paradigma.

Per quanto riguarda la strategia di sviluppo del software, si può dire che è conveniente l'utilizzo di più linguaggi all'interno dello stesso applicativo. Questo consente di unire diverse tecnologie in un unico programma, sia per poter riutilizzare codice già sviluppato sia per necessità di progetto, in quanto potrebbe essere utile scrivere una parte di codice in un linguaggio avente maggiore potenza espressiva.

Per quel che riguarda RoboVM, il tool di traduzione permette di scrivere in Java software iOS senza dover ricorrere a codice Objective-C. Ciò permette lo sviluppo e il riutilizzo di codice per dispositivi mobili che altrimenti sarebbe impossibile poiché Apple impedisce di programmare direttamente in Java sui device iOS. Aspetti positivi del tool sono la forte integrazione con Eclipse e la semplicità con cui è possibile fare il debugging ed il deploy degli applicativi realizzati.

Fire di RemObjects ha una filosofia simile a quella di RoboVM ma si prefigge lo scopo opposto: vuole rendere disponibile il linguaggio Swift per ambienti al di fuori del runtime Cocoa/Cocoa Touch come ad esempio Java o .NET. L'IDE e il compilatore Elements sono gli unici strumenti software a concedere l'utilizzo di Swift in progetti non creati in Xcode. Un aspetto negativo può essere riscontrato in ambito tecnologico: basare il proprio lavoro su Fire ed Elements significa sottostare al *vendor-lock-in*, cioè legarsi sia alle politiche di RemObjects che a quelle di Apple.

I due strumenti hanno permesso di realizzare applicativi che coniugano le caratteristiche multi-paradigma e multi-linguaggio di tuProlog con nuovi scenari, quali la possibilità di tradurre la libreria di interfacciamento Java/Prolog dell'engine in Objective-C e di poter usare Swift per interagire con le API di tuProlog.

Con RoboVM si è ottenuta una app iOS nativa che utilizza la traduzione in Objective-C di tuProlog e che comprende la versione *Objective-C blended* della libreria di interfacciamento tra Java e Prolog. Durante il processo di compilazione e traduzione non sono state riscontrate incompatibilità con l'ambiente Objective-C.

L'utilizzo di Fire ha permesso di sviluppare una applicazione a riga di comando basata su Java ma scritta in Swift. In particolare si sono usate le API di tuProlog da codice Swift per interagire con il motore Prolog. Anche in questo caso non sono state riscontrate incompatibilità tra l'utilizzo di Swift, Java e tuProlog.

Usare Fire e RoboVM comporta però che lo sviluppatore li mantenga aggiornati in quanto questi strumenti si basano su tecnologia all'avanguardia in continua evoluzione.

Tutte le applicazioni realizzate risultano stabili e permettono di utilizzare tuProlog senza rallentamenti. Anche i test effettuati sulla recente release 9.0 di iOS non hanno evidenziato incompatibilità o rallentamenti.

Appendici

In questa sezione si riporta il codice completo degli applicativi realizzati.

Utilizzo di JNA e bridge Weblite verso Objective-C

File ObjectiveC.java

```
import ca.weblite.objc.Client;
import ca.weblite.objc.NSObject;
import ca.weblite.objc.Proxy;
import ca.weblite.objc.annotations.Msg;

public class ObjectiveC extends NSObject
{
    private Client c = null;

    public ObjectiveC(Client c) throws Exception
    {
        super();
        init("NSObject");

        if(c == null) throw new Exception("Client is null!");
        else this.c = c;
    }

    @Msg(selector="allocateAnObject:", signature="v@:@")
    public Proxy allocateAnObject(String toAllocate)
    {
        Proxy obj = this.c.sendProxy(toAllocate, "alloc");
        obj = obj.sendProxy("init");
        return obj;
    }
}
```

File Main.java

```
import ca.weblite.objc.Client;
import ca.weblite.objc.Proxy;

public class Main
{
    public static void main(String[] args) throws Exception
    {
        Client client = Client.getInstance();

        Proxy string = client.sendProxy("NSString", "alloc");
        String str = string.sendString("initWithString:", "Ciao, questa è una stringa
objectiveC :-)");
        System.out.println(str);
        System.out.println();

        ObjectiveC key = new ObjectiveC(client);

        Proxy array = key.allocateAnObject("NSMutableArray");
        array.send("addObject:", "Hello");
        array.send("addObject:", "World");
        String lastString = array.sendString("lastObject");
        System.out.println(lastString);
        System.out.println();

        Proxy fileManager = key.allocateAnObject("NSFileManager");
        String currentDirectory = (String)fileManager.send("currentDirectoryPath");
        System.out.println(currentDirectory);
        System.out.println();
    }
}
```

Applicazione iOS RoboVM: tuProlog

Segue il codice condiviso dall'applicativo iPhone ed iPad. Al termine seguirà il codice specifico di ciascuna versione.

Progetto Xcode

File ViewController.swift

```
import UIKit

class ViewController: UIViewController{

    override func viewDidLoad(){
        super.viewDidLoad()
    }

    override func didReceiveMemoryWarning(){
        super.didReceiveMemoryWarning()
    }

    @IBOutlet weak var nextButton: UIButton!
    @IBOutlet weak var solutionTextView: UITextView!
    @IBOutlet weak var warningsTextView: UITextView!
    @IBOutlet weak var goalTextView: UITextView!
    @IBOutlet weak var theoryTextView: UITextView!

    @IBAction func setTheory(sender: UIButton) {
    }

    @IBAction func solve(sender: UIButton) {
    }

    @IBAction func getNextSolution(sender: UIButton) {
    }

    @IBAction func dismissKeyboardOnTap(sender: UIControl) {
    }
```

Progetto Eclipse

File ViewController.java

```
import org.robovm.apple.foundation.NSRange;
import org.robovm.apple.uikit.*;
import org.robovm.objc.annotation.CustomClass;
import org.robovm.objc.annotation.IBAction;
import org.robovm.objc.annotation.IBOutlet;

import alice.tuprolog.*;

@CustomClass("ViewController")
public class ViewController extends UIViewController
{
    private UITextView solutionTextView;
    private UITextView warningsTextView;
    private UITextView goalTextView;
    private UIButton nextButton;
    private UITextView theoryTextView;

    private Prolog engine = null;
    private SolveInfo info = null;
    private String result = "";

    @Override
    public void viewDidLoad()
    {
        super.viewDidLoad();

        init_prolog();

        String incipit = "tuProlog system - release " + Prolog.getVersion() + "\n\nBeta version";
        solutionTextView.setText(incipit);
    }

    // event handlers
    @IBAction
    private void setTheory()
    {
        hideApplicationKeyboard();
        setTheory(theoryTextView.getText());
    }
}
```

```

@IBAction
private void solve()
{
    hideApplicationKeyboard();
    solve(goalTextView.getText());
}

@IBAction
private void getNextSolution()
{
    hideApplicationKeyboard();
    getNextPrologSolution();
}

//per nascondere la tastiera con un tap
@IBAction
private void dismissKeyboardOnTap()
{
    this.getView().endEditing(true);
}

// funzioni d'istanza
private void init_prolog()
{
    if (engine == null)
    {
        engine = new Prolog();
    }
}

private void clearWarnings()
{
    warningsTextView.setText("");
}

private void setTheory(String theory)
{
    if (theory != null && theory != "")
    {
        try
        {
            engine.setTheory(new Theory(theory));
            solutionTextView.setText("Theory set!");
            clearWarnings();
        }
        catch (InvalidTheoryException e)
        {
        }
    }
}

```

```

        warningsTextView.setText("Error setting theory: Syntax Error
                                at/before line " + e.line);
    }
    catch (Exception e)
    {
        warningsTextView.setText("Error: "+theory+" is an invalid
                                theory!");
    }
}
else
{
    warningsTextView.setText("WARNING: Theory is empty");
}
}

public void solve(String goal)
{
    result = "";
    clearWarnings();
    if (!goal.equals(""))
    {
        try
        {
            solveGoal(goal);
        }
        catch (Exception e)
        {
            warningsTextView.setText("Error: " + e);
            if(nextButton.isEnabled())
            {
                nextButton.setEnabled(false);
            }
        }
    }
    else if (goal.equals(""))
    {
        result += "Ready.";
    }

    solutionTextView.setText(result);
    solutionTextView.scrollToVisible(new
        NSRange(solutionTextView.getText().length(), 0));
}

private void solveGoal(String goal)
{
    result = "";

```

```

clearWarnings();
try
{
    info = engine.solve(goal);

    if (engine.isHalted())
    {
        result += "halt.";
    }
    if (!engine.isHalted() && !info.isSuccess())
    {
        if(info.isHalted())
        {
            result += "halt.";
        }
        else
        {
            result += "no.";
        }
    }
    else
    {
        if (!engine.hasOpenAlternatives())
        {
            String binds = info.toString();
            if (binds.equals(""))
            {
                result += "yes.";
            }
            else
            {
                result += solveInfoToString(info);
            }
            nextButton.setEnabled(false);
        }
        else
        {
            result += solveInfoToString(info);
            nextButton.setEnabled(true);
        }
    }
}
catch (MalformedGoalException ex)
{
    warningsTextView.setText("Syntax Error: malformed goal.");
    if(nextButton.isEnabled())
    {

```

```

        nextButton.setEnabled(false);
    }
}
catch (NoSolutionException ex)
{
    warningsTextView.setText("No solution is available!");
}
}

private String solveInfoToString(SolveInfo result) throws NoSolutionException
{
    return result.toString() + "\n" + result.getSolution().toString() + "\n\n";
}

public void getNextPrologSolution()
{
    if (info != null && info.hasOpenAlternatives())
    {
        try
        {
            info = engine.solveNext();

            if (!info.isSuccess())
            {
                result += "no.\n";
            }
            else
            {
                result += solveInfoToString(info);
            }
        }
        catch (NoMoreSolutionException ex)
        {
            result += "no.";
        }
        catch (NoSolutionException ex)
        {
            warningsTextView.setText("No solution is available!");
        }
        finally
        {
            if(!info.hasOpenAlternatives())
            {
                nextButton.setEnabled(false);
            }
        }
    }
}

```

```

        solutionTextView.setText(result);
        solutionTextView.scrollToVisible(new
            NSRange(solutionTextView.getText().length(), 0));
    }

    private void hideApplicationKeyboard()
    {
        if (theoryTextView.isFirstResponder())
        {
            theoryTextView.resignFirstResponder();
        }
        else if (goalTextView.isFirstResponder())
        {
            goalTextView.resignFirstResponder();
        }
    }

    // View elements setters
    @IBOutlet
    public void setGoalTextView(UITextView textView)
    {
        goalTextView = textView;
        goalTextView.setTextColor(UIColor.black());
        goalTextView.getLayer().setCornerRadius(5.0);
        goalTextView.getLayer().setBorderWidth(1.0);
    }

    @IBOutlet
    public void setSolutionTextView(UITextView textView)
    {
        solutionTextView = textView;
        solutionTextView.setEditable(false);
    }

    @IBOutlet
    public void setWarningsTextView(UITextView textView)
    {
        warningsTextView = textView;
        warningsTextView.setEditable(false);
        warningsTextView.setTextColor(UIColor.red());
    }

    @IBOutlet
    public void setNextButton(UIButton button)
    {
        nextButton = button;
    }

```

```

        nextButton.setEnabled(false);
    }

    @IBOutlet
    public void setTheoryTextView(UITextView textView)
    {
        theoryTextView = textView;
        theoryTextView.setTextColor(UIColor.black());
        theoryTextView.getLayer().setCornerRadius(5.0);
        theoryTextView.getLayer().setBorderWidth(1.0);
    }
}

```

File MainClass.java

```

import org.robovm.apple.foundation.NSAutoreleasePool;
import org.robovm.apple.uikit.UIApplication;
import org.robovm.apple.uikit.UIApplicationDelegateAdapter;
import org.robovm.apple.uikit.UIApplicationLaunchOptions;

public class MainClass extends UIApplicationDelegateAdapter
{
    @Override
    public boolean didFinishLaunching(UIApplication application,
    UIApplicationLaunchOptions launchOptions)
    {
        return true;
    }

    public static void main(String[] args)
    {
        try (NSAutoreleasePool pool = new NSAutoreleasePool())
        {
            UIApplication.main(args, null, MainClass.class);
        }
    }
}

```

Codice specifico della versione iPhone del file ViewController.java

```
//impedisce la rotazione del dispositivo evitando effetti grafici indesiderati
@Override
public boolean shouldAutorotate()
{
    return false;
}

//aggiusta la rotazione del dispositivo evitando effetti grafici indesiderati
@Override
public UIInterfaceOrientationMask getSupportedInterfaceOrientations()
{
    return UIInterfaceOrientationMask.Portrait;
}
```

Codice specifico della versione iPad del file ViewController.java

```
//attiva la rotazione del dispositivo
@Override
public boolean shouldAutorotate()
{
    return true;
}

//aggiusta la rotazione del dispositivo evitando effetti grafici indesiderati
@Override
public UIInterfaceOrientationMask getSupportedInterfaceOrientations()
{
    if(UIDevice.getCurrentDevice().getOrientation().compareTo(UIDeviceOrientation.PortraitUpsideDown) == 0)
    {
        return UIInterfaceOrientationMask.PortraitUpsideDown;
    }
    else
    {
        return UIInterfaceOrientationMask.Portrait;
    }
}
```

Applicazione Fire: richiamare tuProlog da Swift

File Program.swift

```
import java.util
import alice.tuprolog
import java.io

var Engine: Prolog
var Result: String
var Info: SolveInfo

var reader: InputStreamReader = InputStreamReader(System.`in`)
var bReader: BufferedReader = BufferedReader(reader)

func init_Prolog()
{
    if (Engine == nil)
    {
        Engine = Prolog()
        println("\ntuProlog system - release " + Prolog.getVersion() + "\n")
    }
}

func setTheory(theory: String)
{
    if (theory != nil && theory != "")
    {
        __try //IL BLOCCO TRY CATCH è UNA ESTENSIONE DEL
LINGUAGGIO DI REMOBJECTS
        {
            Engine.setTheory(Theory(theory))
            println("Theory set!")
        }
        __catch E: InvalidTheoryException
        {
            println("Error -> \(E.Message) Occurred for an invalid theory!
(E1)\n")
        }
        __catch E: Exception
        {
            println("Error -> \(E.Message) (E2)\n")
        }
    }
    else
    {

```

```

        println("WARNING: Theory is empty")
    }
}

func solve(goal: String) -> Bool
{
    Result = ""
    if (!goal.equals(""))
    {
        __try
        {
            if (!solveGoal(goal))
            {
                return true
            }
            else
            {
                return false
            }
        }
        __catch E: Exception
        {
            println("Error -> \"(E.Message) occurred (E2)\"")
        }
    }
    else if (goal.equals(""))
    {
        Result += "Ready.\n"
    }
    return true
}

func solveGoal(goal: String) -> Bool
{
    Result = ""
    __try
    {
        Info = Engine.solve(goal)
        if (Engine.isHalted())
        {
            Result += "\nhalt.\n"
            return false
        }
        if (!Engine.isHalted() && !Info.isSuccess())
        {
            if (Info.isHalted())
            {

```

```

        Result += "\nhalt.\n"
    }
    else
    {
        Result += "\nno.\n"
    }
}
else
{
    if (!Engine.hasOpenAlternatives())
    {
        var binds: String = Info.toString()
        if (binds.equals(""))
        {
            Result += "\nyes.\n"
        }
        else
        {
            Result += solveInfoToString(Info)
        }
    }
    else
    {
        println(solveInfoToString(Info))
        var stopRenderAlternatives = false
        while(!stopRenderAlternatives &&
Engine.hasOpenAlternatives())
        {
            print("Digita 'y' per visualizzare le alternative,
qualsiasi tasto per annullare: ")
            var userInput = bReader.readLine()
            if(userInput.equalsIgnoreCase("y"))
            {
                print("\nNext Solution:")
                println(getNextSolution())
            }
            else
            {
                stopRenderAlternatives = true
                println()
            }
        }
        return true
    }
}
}
}

```

```

        __catch E: MalformedGoalException
        {
            println("Error -> \"(E.Message) occurred for a malformed goal!
(E3)\\n\"")
        }
        return false
    }

func solveInfoToString(result: SolveInfo) -> String
{
    return "\\n"+result.getdescription() + "\\nSolution: "+result.getSolution() + "\\n"
}

func getNextSolution() -> String
{
    if (Info.hasOpenAlternatives())
    {
        __try
        {
            Info = Engine.solveNext()
            if (!Info.isSuccess())
            {
                return "\\nno.\\n"
            }
            else
            {
                return solveInfoToString(Info)
            }
        }
        __catch E: NoMoreSolutionException
        {
            return "\\nno.\\n"
        }
    }
    __throw Exception("Illegal operation!")
}

func Program()
{
    init_Prolog()

    var terminateApplication = false

    while(!terminateApplication)
    {
        print("Digita 't' per inserire una Teoria, 'g' per risolvere un Goal, 'e'
per terminare: ")
    }
}

```

```

        var userInput = bReader.readLine()
        if(userInput.equalsIgnoreCase("t") ||
userInput.equalsIgnoreCase("g"))
        {
            if(userInput.equalsIgnoreCase("t"))
            {
                println("Inserisci la Teoria: ")
                var th = bReader.readLine()
                setTheory(th)
            }
            else
            {
                println("Inserisci un Goal: ")
                var go = bReader.readLine()
                if(solve(go))
                {
                    println(Result)
                }
            }
        }
        else if (userInput.equalsIgnoreCase("e"))
        {
            println("Termino applicazione!\n")
            terminateApplication = true
            bReader.close()
            reader.close()
        }
    }
}

```

Program()

Bibliografia

- [1] APICe, *tuProlog*, <http://apice.unibo.it/xwiki/bin/view/Tuprolog/> (2015)
- [2] Wikipedia, *Swift*, [https://en.wikipedia.org/wiki/Swift_\(programming_language\)](https://en.wikipedia.org/wiki/Swift_(programming_language)) (08/2015)
- [3] Wikipedia, *Objective-C*, <https://it.wikipedia.org/wiki/Objective-C> (08/2015)
- [4] RemObjects, *Oxygene*, <http://www.elementscompiler.com/elements/oxygene/> (2015)
- [5] The Swift Programming Language, Apple, disponibile su *iBooks Store* (2014)
- [6] Apple Developer, *Swift*, <https://developer.apple.com/swift/> (2015)
- [7] Our Blog, *Objective-C vs Swift messages dispatch*, <http://blog.untitledkingdom.co.uk/obj-c-vs-swift/> (2015)
- [8] Wikipedia, *Java*, [https://it.wikipedia.org/wiki/Java_\(linguaggio_di_programmazione\)](https://it.wikipedia.org/wiki/Java_(linguaggio_di_programmazione)) (05/2015)
- [9] Oracle, *Java Software*, <https://www.oracle.com/java/index.html> (2015)
- [10] Wikipedia, *iOS*, [https://it.wikipedia.org/wiki/iOS_\(Apple\)](https://it.wikipedia.org/wiki/iOS_(Apple)) (09/2015)
- [11] Apple Developer, *Swift and Objective-C in the Same Project*, <https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/BuildingCocoaApps/MixandMatch.html> (08/2015)
- [12] Using Swift with Cocoa and Objective-C, Apple, disponibile su *iBooks Store* (2014)
- [13] Wikipedia, *JNI*, https://en.wikipedia.org/wiki/Java_Native_Interface (08/2015)
- [14] Wikipedia, *JNA*, https://en.wikipedia.org/wiki/Java_Native_Access (07/2015)
- [15] Oracle documentation, *JNI Functions*, <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html> (2014)
- [16] Weblite, *Home*, <http://solutions.weblite.ca> (2010)
- [17] GitHub, *Java-Objective-C bridge*, <https://github.com/shannah/Java-Objective-C-Bridge> (08/2014)
- [18] RoboVM, *Home*, <http://robovm.com> (08/2015)
- [19] RemObjects, *Home*, <http://www.elementscompiler.com/elements/> (2015)

- [20] RemObjects, *Oxidizer*, <http://docs.elementscompiler.com/Tools/Oxidizer/#q=oxidizer> (2015)