Alma Mater Studiorum – Università di Bologna
Corso di Laurea Magistrale in Ingegneria Informatica

Processo di sviluppo per applicazioni multi-piattaforma e multi-paradigma: il caso tuProlog

Tesi di laurea in Linguaggi e Modelli Computazionali M

Relatore
Prof. Enrico Denti

Presentata da
Andrea Bucaletti

Correlatore
Dott.Ing. Roberta Calegari

Sessione II – Anno Accademico 2014/2015

Indice generale

1. Introduzione e obiettivi

Negli ultimi anni, il termine multi-piattaforma è stato soprattutto associato ad applicazioni di tipo mobile che necessitano di essere sviluppate per dispositivi molto diversi tra loro, sia come hardware che come sistema operativo. In questo ambito sono ad oggi disponibili diversi framework che permettono di progettare l'applicazione in un solo linguaggio di programmazione e successivamente si fanno carico del deployment per un determinato insieme di piattaforme.

In questa tesi vengono considerate invece applicazioni multi-piattaforma e multi-linguaggio, ossia sviluppate utilizzando più linguaggi di programmazione, i quali hanno le proprie IDE e i propri tool per la compilazione e il deployment. In questo contesto è interessante considerare il *processo di sviluppo* dell'applicazione, ossia quel processo che si fa carico di tutti i passi necessari per realizzare il deployment dell'applicazione per tutte le piattaforme considerate.

In particolare, si considera il caso di tuProlog[1], un interprete prolog[2] disponibile per le piattaforme Java, Android, .NET e iOS. Data l'eterogeneità delle piattaforme utilizzate, tuProlog è implementato utilizzando più linguaggi di programmazione e dispone di un processo di sviluppo progettato ad-hoc che permette il deployment dell'applicazione per le piattaforme supportate.

Il processo è svolto in buona parte in maniera manuale, con l'ausilio di alcuni script Apache ANT[3], ed è progettato per integrarsi con Subversion[4], con tutte le rigidità che questo comporta. Obiettivo di questa tesi è di effettuare un refactoring di tale processo in modo da renderlo:

- *locale*, e quindi non dipendente da repository esterni
- *estendibile*, in modo da supportare le piattaforme più innovative
- *facilmente riconfigurabile* per esigenze future

Il processo attuale dispone di una documentazione solo parziale, il che rende necessaria una fase di *studio* dell'implementazione per poterne definire dettagliatamente l'architettura e tutte le possibili modalità di utilizzo.

Si procede quindi con l'*analisi* architetturale e implementativa del processo al fine di individuare un insieme di *requisiti* che il nuovo processo deve soddisfare per garantire il raggiungimento degli obiettivi fissati.

In fase *progettazione* viene illustrato il nuovo processo di sviluppo, mostrando come la soluzione adottata soddisfa i requisiti fissati in precedenza con scelte a livello architetturale e implementativo.

1 Interprete prolog sviluppato da APICe [1]
2 Linguaggio per la programmazione logica
3 Strumento per rendere automatici i processi di sviluppo software [3]
4 Sistema di version control open-source [2]

2. Caso di studio: tuProlog

tuProlog è un'interprete Prolog scritto in Java appositamente pensato per essere multi-piattaforma, multi-paradigma e multi-linguaggio. Il sistema è sviluppato attorno ad un core di dimensioni estremamente contenute che consente poi di caricare librerie di predicati Prolog in modo statico o dinamico.

Attualmente, tuProlog è disponibile per le piattaforme Java, .NET e iOS. Quest'ultima non viene inserita nel processo in quanto era ancora in fase di sviluppo al momento della progettazione, ma è comunque necessario tenere conto della sua presenza e quindi prevedere dei meccanismi per integrarla nel nuovo processo in modo semplice e veloce.

E' presente anche una versione Plugin per Eclipse[1], al momento non compatibile con le versioni più recenti della IDE.

In questo capitolo verranno analizzate la struttura del progetto e il processo di sviluppo che consente il deployment per tutte le piattaforme disponibili.

2.1. Processi di sviluppo multi-piattaforma

Il processo di sviluppo software più semplice che si può immaginare è quello che utilizza un solo linguaggio per l'implementazione e considera una sola piattaforma. In questi casi si utilizzano tool di sviluppo specifici per il linguaggio e la piattaforma considerata. Generalmente il processo è gestito in maniera automatica dalla IDE utilizzata per lo sviluppo, che si fa carico di tutte le fasi, dalla gestione dei sorgenti alla generazione dell'applicazione pronta per la distribuzione.

Nel caso invece l'applicazione ha come target una sola piattaforma ma è implementata utilizzando più linguaggi di programmazione lo scenario si complica leggermente, in quanto il processo di sviluppo richiede l'utilizzo di tool specifici per i linguaggi utilizzati. Tuttavia lo scenario è ancora abbastanza semplice, in quanto la scelta dei linguaggi è dettata anche dalla capacità di questi di integrasi tra loro in maniera naturale: ad esempio, considerando Windows come piattaforma target, si può decidere di implementare le funzioni di libreria dell'applicazione in C/C++ per ottenere delle buone performance, e utilizzare Java/C# per quanto riguarda l'interfaccia utente.

Aggiungere il supporto per più piattaforme significa aggiungere una dimensione al problema: anche nel caso mono-linguaggio si devono utilizzare strumenti specifici per ogni piattaforma per effettuare il deployment dell'applicazione e considerare anche la possibilità che il linguaggio scelto non sia compilabile in determinate piattaforme. Inoltre, se si utilizzano librerie di terze parti, è necessario verificare che anch'esse abbiano le relative implementazioni per le piattaforme considerate.

Il caso di un'applicazione multi-piattaforma e multi-linguaggio è il più complesso: è necessario considerare la possibilità che alcune funzionalità non siano disponibili per una o più piattaforme considerate, e quindi richiedano di essere reimplementate per la specifica piattaforma. Il caso limite è che, per ogni piattaforma considerata, sia necessaria una completa implementazione dell'applicazione: questo è uno scenario poco verosimile, in quanto implica la gestione di *N* implementazioni diverse per la stessa applicazione, rendendo quindi molto difficile l'apporto di modifiche e/o la correzione di bug. Inoltre, essendo le implementazioni totalmente indipendenti, si possono riscontrare comportamenti diversi a seconda della piattaforma considerata.

1 IDE open-source basata su Java [4]

2.2. Processo di sviluppo di tuProlog

Nel caso specifico di tuProlog, lo scenario è quello di un applicazione multi-piattaforma e multi-linguaggio, tuttavia non così complesso come quello descritto in precedenza: le funzionalità di base di tuProlog sono implementate in Java, e vengono integrate nelle altre piattaforme attraverso delle operazioni di conversione del bytecode e tramite del codice platform-dependant.

Il seguente schema[1] mostra una visione semplificata del processo di sviluppo di tuProlog per le piattaforme considerate:

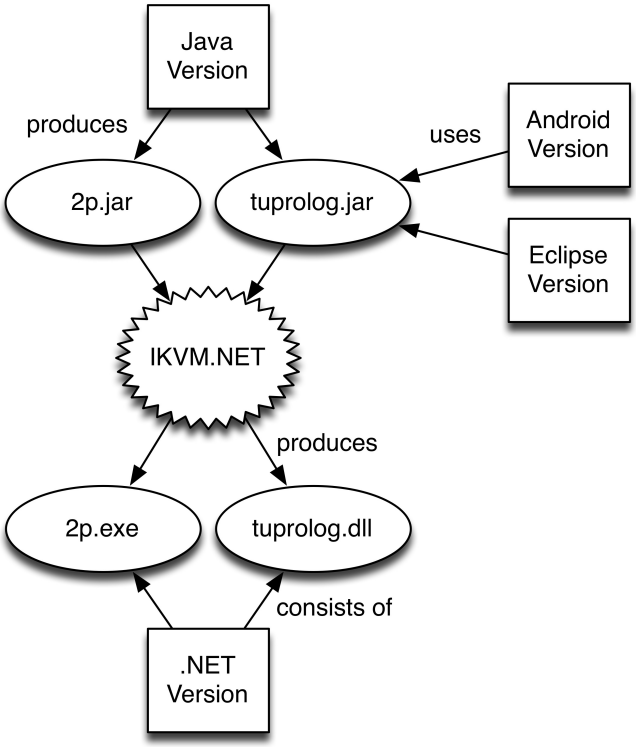


Illustrazione 2.1: Processo di sviluppo

La versione Java è l'implementazione del core di tuProlog (*tuprolog.jar*), che include tutte le funzionalità per interpretare ed eseguire programmi prolog, e di un interfaccia grafica per l'editing e l'esecuzione di programmi Prolog (*2p.jar*).

Le versioni Android e Eclipse sono costituite dall'implementazione delle rispettive interfacce grafiche per le due piattaforme, e utilizzano il core come libreria. In questi due casi è possibile utilizzare direttamente *tuprolog.jar* in quanto il linguaggio utilizzato per lo sviluppo di applicazioni Android e plugin per Eclipse è comunque Java.

La versione .NET è invece ottenuta tramite traduzione del bytecode Java in file eseguibili e DLL (*2p.exe* e *tuprolog.dll*) utilizzando il tool *IKVM*[2] (per approfondimenti si rimanda al capitolo 5).

Il processo di sviluppo è costituito da una serie di script e alcune azioni che lo sviluppatore deve eseguire manualmente per generare l'applicazione pronta per la

1 Fonte: tuProlog Project Wiki [5]
2 Tool per la conversione di bytecode Java in bytecode .NET [6]

distribuzione.

Apache ANT è lo strumento utilizzato per eseguire il processo in maniera automatica: si tratta di un tool che tramite degli script XML permette di definire i vari passi che compongono il processo, detti *target*[1], e la sequenza con cui devono essere eseguiti. Per una descrizione dettagliata di questo strumento si rimanda al capitolo 5.

2.2.1. Dipendenze da Subversion

Il codice sorgente viene mantenuto tramite GoogleCode[2], che utilizza un sistema di versioning basato su *Subversion* (o *SVN).* Tutti gli script del processo di sviluppo sono pensati per integrarsi con questo utilizzando dei meccanismi propri di SVN.

In particolare si fa rifermento all'operazione di *tag* del codice, che permette di creare una copia del progetto o di una sua parte e assegnargli un etichetta (tag).

Un altra dipendenza è data dall'uso negli script del costrutto *External Definitions*[3] (o *SVN Externals*), che permette di includere all'interno del progetto dei link che fanno riferimento a file o directory, anche esterne al repository stesso. Quando si effettua il *checkout* del progetto o di una sua parte, i file/directory referenziati tramite SVN Externals vengono automaticamente inclusi nella copia locale dell'utente.

Nel caso specifico di tuProlog, SVN Externals viene utilizzato per includere i file *2p.exe* e *tuprolog.dll* nella sotto-cartella della versione .NET, e *tuprolog.jar* in quella dell'applicazione Android.

1 "A target is a container of tasks that cooperate to reach a desired state during the build process" [7]
2 Servizio di project-hosting per progetti open-source
3 Version Control with Subversion [8]

2.2.2. *Processo di sviluppo: Java*

Il processo di sviluppo di tuProlog per Java include uno script ANT di cui si riporta uno schema che ne evidenzia i principali target e le loro relazioni di dipendenza:

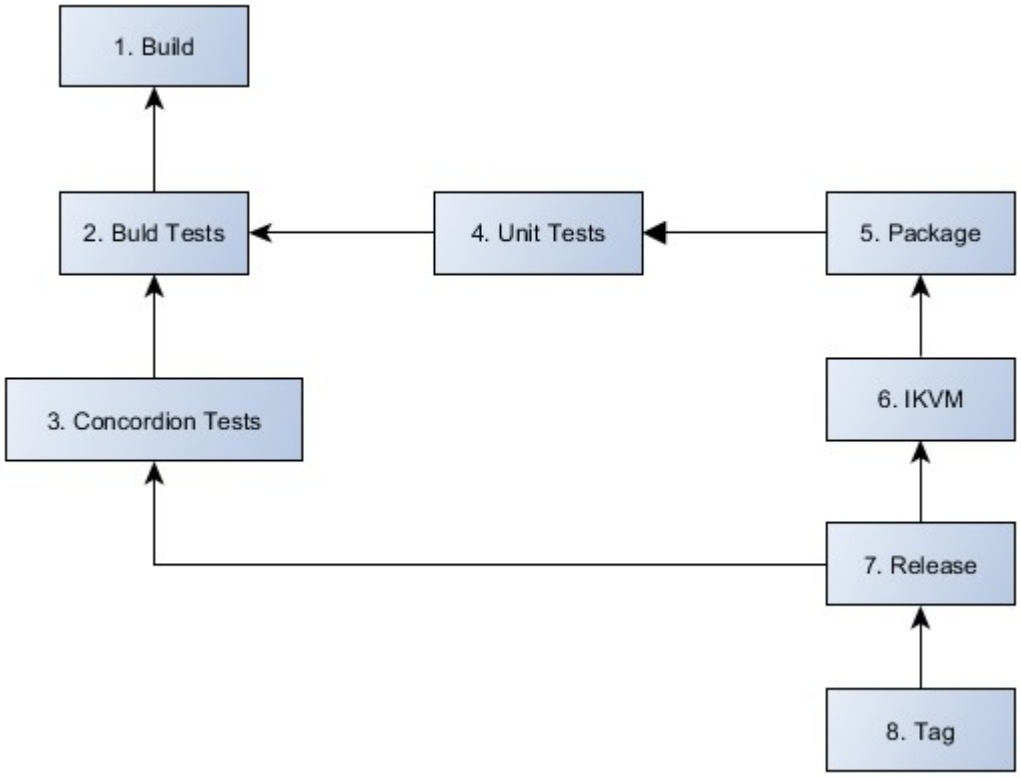


Illustrazione 2.2: Processo di sviluppo Java

Il processo è completamente automatico:

- I target *Build* e *Build Tests* compilano rispettivamente i sorgenti dell'applicazione e i relativi file di test.
- L'esecuzione e la stampa dei report relativi ai test è delegata ai target *Unit Tests* e *Concordion Tests*
- Il target *Package* genera i due file tuprolog.jar e 2p.jar, che costituiscono l'input del target IKVM, il quale utilizzando l'omonimo tool genera i due file tuprolog.dll e 2p.exe.
- A questo punto è possibile eseguire il target *Release* per generare un archivio contenente la versione Java di tuProlog, comprensiva dei file binari, il manuale e dei file di esempio.
- Infine, è possibile taggare il codice sul repository SVN utilizzando il target *Tag*.

2.2.3. *Processo di sviluppo: .NET*

La versione .NET dipende dalla versione Java e, in particolare, dai due file

2p.exe e *tuprolog.dll* generati dal target *IKVM*. Quest'ultimo inserisce i due file nel progetto Java., ma vengono inclusi nel processo .NET utilizzando il meccanismo *External Definition* di SVN: quando lo sviluppatore effettua il checkout del progetto i due file vengono automaticamente copiati nella cartella locale.

Oltre a *2p.exe* e *tuprolog.dll*, la versione .NET è costituita da un altro progetto che implementa la libreria ad oggetti di tuProlog (*OOLibrary*[1]) appositamente per .NET.

Lo script ANT incluso nel progetto .NET è molto semplice:

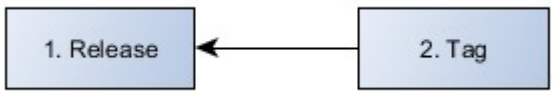


Illustrazione 2.3: Processo di sviluppo .NET

Il target *Release* utilizza il compilatore di Visual Studio (MSBuild[2] e C# Compiler), il quale genera *OOLibrary.dll*, e successivamente crea un archivio ZIP contenente l'applicazione .NET pronta per la distribuzione.

Il target *Tag* esegue l'omonima operazione sul repository SVN.

2.2.4. Processo di sviluppo: Android

La versione Android di tuProlog, come specificato in precedenza, ha bisogno dell'archivio *tuprolog.jar*, che fisicamente si trova nella sotto-cartella della versione Java di tuProlog e viene incluso tramite SVN Externals.

Non si riporta lo schema dello script ANT per la versione Android, in quanto prevede un unico target *Tag* che effettua l'omonima operazione SVN.

La versione per la distribuzione consiste in un archivio APK, ossia un installer per applicazioni Android. Tutti i tool per la generazione di questi archivi sono inclusi nel pacchetto Android SDK[3] (per approfondimenti si veda il capitolo 5).

Tuttavia, nel caso di tuProlog ci si affida al plugin ADT[4] per Eclipse, che fornisce alla IDE degli strumenti visuali per lo sviluppo di applicazioni Android, e permette direttamente di lanciare l'applicazione su un emulatore o un device collegato.

Per mettere in esecuzione l'applicazione Android, il plugin ADT genera automaticamente l'archivio APK e lo installa sull'AVD[5]. Il processo di sviluppo di tuProlog prevede di sfruttare questa caratteristica del plugin per ottenere l'applicazione pronta per la distribuzione.

1 Libreria di predicati che permette di utilizzare oggetti Java in un programma prolog [9]
2 Tool di build automation per .NET e Visual Studio
3 Software Development Kit per applicazioni Android [10]
4 Un plugin arricchisce la IDE di Eclipse con nuove funzionalità. In particolare il plugin ADT offre dei tool specifici per le applicazioni Android.
5 Android Virtual Device, necessario all'emulatore per Android [10]

3. Analisi del problema

In questo capitolo viene analizzato l'attuale processo di sviluppo di tuProlog descritto nel capitolo 2 individuandone problemi e limitazioni.

Il processo di sviluppo descritto nel capitolo precedente e la conseguente organizzazione fisica e logica, sono progettati per integrarsi con Subversion, ed in particolare ne sfruttano determinate caratteristiche per risolvere le dipendenze fisiche tramite l'uso di External Definitions[1] e definiscono delle operazioni di tag del codice che funzionano correttamente solo con Subversion.

Uno dei requisiti che ci pone è eliminare la dipendenza del processo da Subversion, in particolare per poter spostare tuProlog da GoogleCode (in via di dismissione da parte di Google) a BitBucket[2], che invece è basato su GIT[3].

In quest'ottica è quindi necessario eliminare i costrutti propri di SVN, ricordando che alcuni di questi possono essere rimossi senza conseguenze (come ad esempio i target di tipo *Tag*), mentre altri sono necessari per il corretto funzionamento del processo (ad esempio External Definitions per le piattaforme Android e .NET), e quindi si devono prevedere dei meccanismi sostitutivi.

Il problema principale che emerge analizzando il processo nella sua totalità è la *frammentazione fisica* degli script e dei file di configurazione: il processo è costituito da tre script ANT per le versioni Java, .NET e Android, inclusi nelle rispettive cartelle. Ognuno di questi referenzia dei file di configurazione, anch'essi sparsi tra le varie cartelle.

Questa frammentazione rende difficile apportare modifiche al processo stesso, ma soprattutto in alcuni casi costringe lo sviluppatore ad eseguire una sequenza di azioni poco intuitiva e più lunga del dovuto. Considerando, ad esempio, il caso in cui l'utente abbia apportato delle modifiche al core di tuProlog e voglia propagarle alla versione Android, i passi da seguire sono i seguenti:

1. Eseguire il target *Package* della versione Java, in modo da generare il file *tuprolog.jar*.
2. Copiare manualmente il file *tuprolog.jar* nella sotto-cartella Android
3. Aprire un AVD[4]
4. Mettere in esecuzione l'applicazione Android in Eclipse in modo da forzare la generazione del corrispondente archivio APK.

Benché questo procedimento funzioni correttamente, è necessario rendere il processo più chiaro e sopratutto più veloce: il target *Package* e l'apertura di un AVD sono azioni molto onerose. Un discorso simile può essere fatto per la versione .NET. In particolare la sequenza da eseguire è la seguente:

1. Eseguire il target *IKVM* della versione Java, in modo da generare i file *2p.exe* e *tuprolog.dll*
2. Copiare manualmente i due file nella sotto-cartella .NET
3. Eseguire il target *Release* della versione .NET

I due esempi riportati mostrano anche come il processo sia frammentato a

1 Meccanismo di SVN per includere automaticamente file e cartelle al momento del checkout [8]
2 Servizio di hosting per progetti basato su GIT
3 Sistema di version control distribuito e open-source[11]
4 Android Virtual Device [10]

livello *logico*, in quanto si impone all'utente di eseguire una sequenza di azioni automatiche (script ANT) e manuali (copia di file, apertura di un AVD). In questo caso sarebbe opportuno eliminare le azioni manuali e rendere quindi il processo *totalmente automatico*.

Un altro aspetto che emerge dagli esempi riportati sopra, è la mancanza di *flessibilità* del processo: in entrambi i casi l'utente è costretto ad eseguire il target *Package* (o *IKVM*) del processo Java. Come viene mostrato nella sezione 3.1, questa è un'operazione molto pesante in termini computazionali, ed inoltre esegue delle azioni che non sono necessarie per le applicazioni .NET e Android, come ad esempio, l'esecuzione dei test. In questo contesto, è opportuno dare all'utente la possibilità di selezionare esplicitamente i target da eseguire senza costringerlo ad eseguire anche le relative dipendenze.

L'introduzione di questa feature amplia enormemente la libertà dell'utente, che può scegliere se includere o escludere un determinato target dal processo, permettendo di ottimizzare il tempi di esecuzione. Tuttavia questo presuppone che l'utente sia a conoscenza dell'architettura del sistema, intesa l'insieme di tutti i target disponibili e relative dipendenze, cosa che al momento è possibile solo analizzando gli script ANT.

Anche supponendo che l'utente abbia una conoscenza completa e dettagliata dell'architettura del processo, utilizzare ANT in questo contesto risulta poco pratico: è necessario ad ogni esecuzione, specificare (da riga di comando, o tramite Eclipse) quali target includere nel processo.

E' inoltre necessario anche tenere conto dell'errore umano, in quanto l'eliminazione dell'esecuzione automatica delle dipendenze dà la possibilità all'utente di mettere in esecuzione una sequenza di target che non rispetta l'ordine dettato dalle dipendenze, e che quindi non garantisce la consistenza delle operazioni.

In conclusione, aumentare la *flessibilità* del processo implica anche aumentarne la complessità di utilizzo e configurazione. Una possibile soluzione è quella affiancare al processo di sviluppo un *interfaccia grafica* con le seguenti caratteristiche:

1. *visione di insieme*: l'utente deve avere una visione totale del processo di sviluppo, ossia dei vari target e piattaforme disponibili, e delle loro relazioni di dipendenza
2. *facilità di configurazione*: il processo di sviluppo può cambiare nel tempo, ad esempio attraverso l'integrazione di nuove piattaforme e/o l'introduzione di nuove feature per le piattaforme esistenti. E' quindi necessario che l'interfaccia sia facilmente riconfigurabile per essere aderente alle modifiche introdotte.
3. *praticità*: l'interfaccia deve permettere la selezione e l'esecuzione di un sotto-insieme dei target disponibili in maniera semplice e veloce
4. *consistenza*: relativamente ai target selezionati, si deve garantire un'esecuzione ordinata secondo le dipendenze

3.1. Analisi del processo di sviluppo: Java

Si riporta lo schema visto nel capitolo 2 che mette in evidenza i target disponibili nello script ANT per la versione Java e le relative dipendenze:

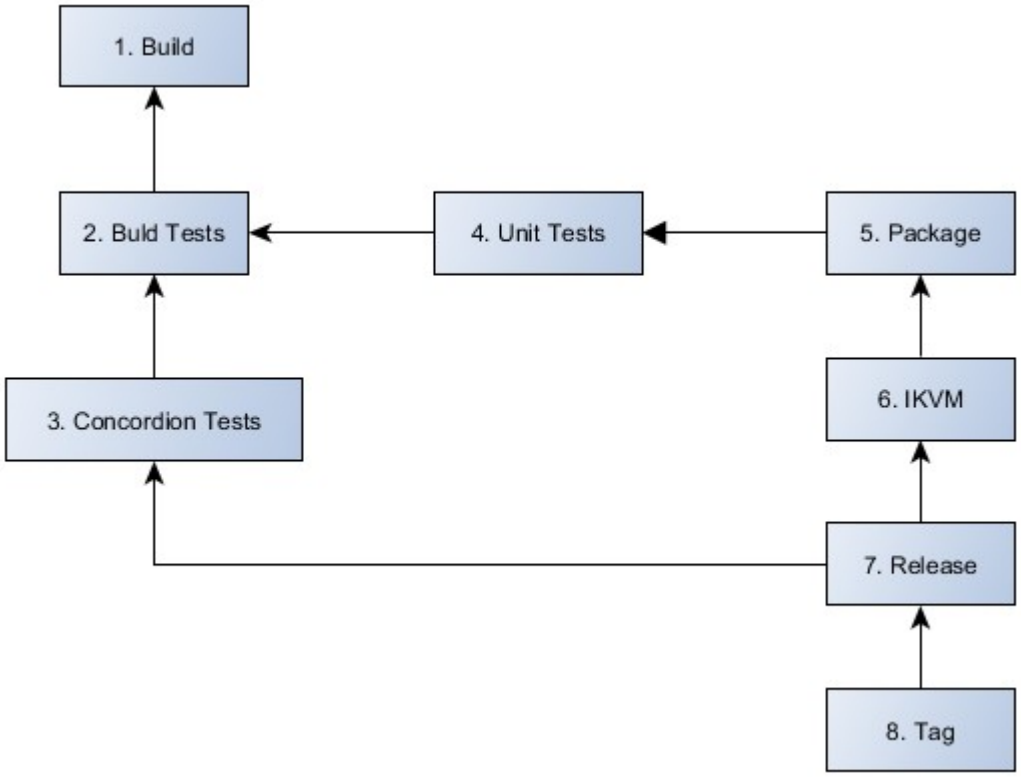


Illustrazione 3.1: Processo di sviluppo Java

Si ricorda che le frecce indicano le dipendenze tra i vari target: in particolare, l'esecuzione di un target determina l'esecuzione ricorsiva di tutti gli altri target da cui esso dipende.

Si consideri il caso in cui venga eseguito il target *Release.* La catena di esecuzione è la seguente:

Build → Build Tests → Unit Tests → Condordion Tests → Package → IKVM → Release

Si nota subito che il target *IKVM*, che lo sviluppatore è comunque costretto ad eseguire, non ha nulla a che vedere con la versione Java dell'applicazione. La scelta di inserirlo nella versione Java sta nel fatto che, se vengono apportate delle modifiche è necessario rigenerare anche i due file *2p.exe* e *tuprolog.dll* per rendere i cambiamenti effettivi anche nell'applicazione .NET.

Tuttavia è ragionevole pensare che durante lo sviluppo della versione Java l'utente non sia interessato all'esecuzione di *IKVM*, che può essere fatta nel momento in cui si decide di lavorare sulla versione .NET.

Un'altra osservazione da fare riguarda i target relativi ai test, ossia *Build Tests*, *Unit Tests* e *Concordion Tests:* è concettualmente corretto pensare che il target *Release* venga eseguito soltanto a lavoro ultimato, e che quindi i risultati dei test diano l'esito desiderato. Si considerino tuttavia i seguenti casi particolari:

1. L'utente apporta delle modifiche che non richiedono una nuova esecuzione dei test
2. L'utente esegue i test, verifica i risultati e successivamente esegue *Release*

In entrambi i casi i test vengono eseguiti più volte, il ché non è un dettaglio trascurabile in quanto la compilazione e l'esecuzione dei target relativi ai test richiedono più tempo della generazione dell'applicazione. Nel caso 1, idealmente la sequenza che si vorrebbe eseguire è la seguente: *Build → Package → Release*.

Un altro aspetto da considerare è che l'utente voglia eseguire la stessa sequenza indicata sopra, ma lanciando un target alla volta e verificando la correttezza o meno dell'output. Ad esempio, si potrebbe voler eseguire la sequenza *Build → Package*, generando l'eseguibile dell'applicazione per controllarne il corretto funzionamento, e successivamente lanciare il solo target *Release.*

Da notare che tutto questo al momento non è possibile, poiché la struttura stessa dello script porta ad esecuzioni ridondanti di uno o più target.

Analizzando invece i singoli target emergono i seguenti problemi:

- Il target *Tag* non è più utilizzabile
- i target *Build* e *Build Tests* ad ogni esecuzione compilano tutti i sorgenti, anche quelli che non sono stati modificati. Per velocizzare il processo sarebbe opportuno utilizzare una compilazione incrementale che vada a ricompilare soltanto i file necessari.
- il target *IKVM* converte gli archivi JAR compilati appositamente per la versione Java del progetto: questo può portare a delle incompatibilità nel momento in cui viene rilasciata una nuova versione di JDK[1]. Ad esempio, classi e JAR compilati con JDK8 non possono essere convertiti in bytecode .NET utilizzando IKVM7

3.2. Analisi del processo di sviluppo: .NET

Da quando il codice sorgente del progetto è stato spostato da GoogleCode a BitButcket, il target *Tag* ovviamente non è più utilizzabile, in quanto basato sul meccanismo di tag di SVN. Inoltre non è più possibile utilizzare il meccanismo Externals di SVN per includere automaticamente i due file *2p.exe* e *tuprolog.dll*, che quindi devono essere copiati manualmente.

Il target *Release* del processo .NET compila i soli file relativi alla versione .NET di tuProlog e successivamente genera un archivio ZIP contenente l'applicazione pronta per la distribuzione. Per questioni di praticità queste due fasi dovrebbero essere separate: lo sviluppatore .NET potrebbe voler compilare spesso il codice per verificarne la correttezza, senza ogni volta generare l'archivio per la distribuzione.

3.3. Analisi del processo di sviluppo: Android

Il processo di sviluppo per Android include il solo target *Tag* che, come nel caso .NET, non è più utilizzabile da quando il repository è basato su GIT, ed inoltre il file *tuprolog.jar*, usato come libreria dall'applicazione Android, non può più essere automaticamente importato tramite SVN Externals.

Un altro problema legato al file *tuprolog.jar* è che viene compilato nel processo di sviluppo per Java, e questo può creare delle incompatibilità con Android, che non

1 Java Development Kit [12]

sempre supporta il bytecode compilato con l'ultima versione di Java rilasciata: ad esempio, attualmente JDK è alla versione 8, mentre Android 5.1 supporta classi compilate con JDK 7 o inferiore.

Ci sono anche dei problemi legati alla modalità di generazione dell'archivio APK, che come specificato nel capitolo 2, viene forzata tramite il plugin ADT[1] per Eclipse. Dalla guida ufficiale di Android si conclude facilmente che questa pratica è sconsigliabile, in quanto Android ha da tempo abbandonato lo sviluppo del plugin per Eclipse ed ha deciso di migrare alla nuova IDE Android Studio.

Un altro aspetto da considerare è che, eseguendo il procedimento indicato sopra, Eclipse genera l'applicazione in versione Debug, che non è firmata e quindi non distribuibile nell'AppStore Android.

1 Plugin per Eclipse per lo sviluppo di Applicazioni Android

4. Verso un nuovo processo di sviluppo

In questo capitolo si discute come riorganizzare il processo di sviluppo di tuProlog in modo da renderlo locale, estendibile e facilmente configurabile.

4.1. Riassunto dei requisiti principali

Mediante l'analisi fatta nel capitolo 3, si definiscono i requisiti che il nuovo processo di sviluppo deve soddisfare:

1. *indipendenza da SVN*[1]: il nuovo processo non deve dipendere dal sistema di versioning utilizzato, in quanto questo può essere soggetto a cambiamenti
2. *compattezza*: il processo dovrebbe essere organizzato in modo tale che tutti i file necessari allo stesso siano facilmente reperibili, in modo da facilitare future modiche e estensioni
3. *flessibilità*: il processo risulta rigido in quanto spesso costringe l'utente ad eseguire una serie di azioni non necessarie, aspetto risulta che evidente per tutte le piattaforme considerate. Inoltre, in fase di analisi sono stati mostrati dei casi reali in cui l'utente possa voler lanciare un singolo target senza che eseguire la sua intera catena di dipendenze.
4. *estendibilità*: si ricorda che tuProlog è disponibile anche in versione per iOS, non considerata in questa tesi. Tuttavia è necessario tenere conto della sua presenza e quindi prevedere dei metodi semplici per integrare nuove piattaforme.
5. *interfaccia grafica*: come specificato in fase di analisi, aumentare la flessibilità del processo implica una maggiore complessità dello stesso in termini di utilizzo e configurazione. Si rende quindi necessario uno strumento che permetta all'utente di utilizzare in modo chiaro e semplice tutte le nuove feature introdotte, e allo stesso tempo garantire la consistenza delle operazioni.

Considerando invece i processi per le singole piattaforme, è necessario risolvere i seguenti problemi:

- Versione Java: ridurre il numero di target e definire delle catene di dipendenza meno stringenti, in modo da diminuire il tempo di esecuzione e offrire maggiore flessibilità
- Versione .NET: è necessario eliminare le azioni manuali richieste e rendere il processo completamente automatico. Inoltre, per garantire il corretto funzionamento del tool IKVM[2], è necessario permettere all'utente di selezionare la versione di JDK[3] da utilizzare per la generazione dei file *tuprolog.jar* e *2p.jar*.
- Versione Android: è necessario eliminare le azioni manuali da eseguire e la dipendenza dal plugin ADT per Eclipse. Permettere la selezione della JDK da utilizzare per compilare il file *tuprolog.jar* in modo da assicurare la compatibilità con la versione di Android SDK[4] utilizzata

1 Sistema di version control open-source [2]
2 Tool per la conversione di bytecode Java in bytecode .NET [6]
3 Java Development Kit [12]
4 Software Development Kit per applicazioni Android [10]

4.2. Progetto

Per soddisfare il requisito di *compattezza*, una possibile soluzione consiste nel:

- inserire tutti le risorse necessarie al processo di sviluppo in una cartella separata da quelle del codice sorgente dell'applicazione, in modo che l'utente possa facilmente accedere agli script e ai file di configurazione
- implementare i processi di sviluppo per tutte le piattaforme in un unico script ANT

Tenendo conto dei requisiti specifici per ogni piattaforma, si adotta un processo con la seguente struttura:

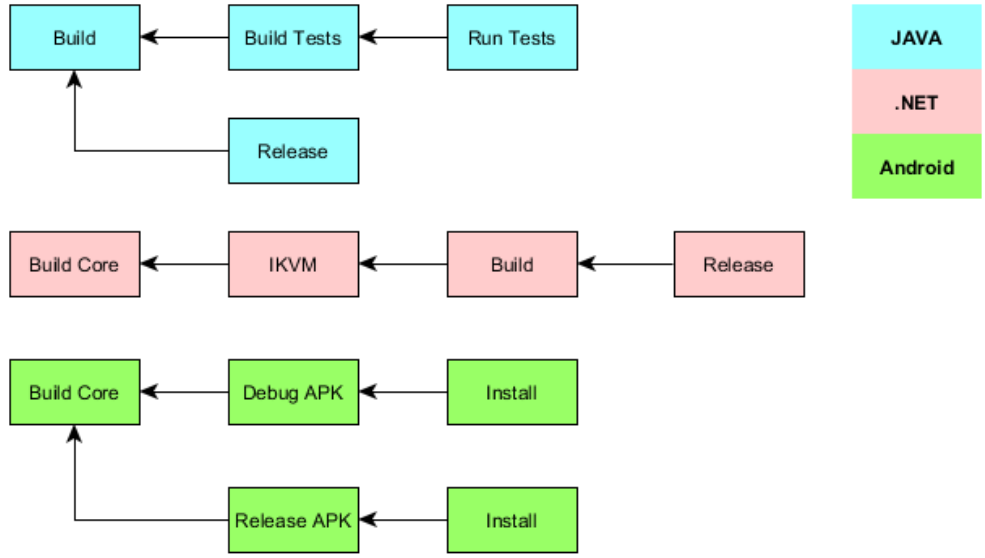


Illustrazione 4.1: Struttura del processo di sviluppo

La struttura adottata dal nuovo processo prevede la separazione logica delle piattaforme considerate definendo per ogni piattaforma un insieme di target auto contenuto (ossia non dipendente da target relativi ad altre piattaforme) che permette la gestione di tutte le fasi del processo di sviluppo per la specifica piattaforma.

Questo porta ad un vantaggio pratico per lo sviluppatore che, in base alla piattaforma considerata, è in grado di determinare quali target utilizzare senza eseguire azioni non necessarie. Nel precedente processo, si ricorda come i processi di sviluppo per Android e .NET richiedessero di eseguire dei target definiti nel processo relativo a Java.

Inoltre, questa architettura risulta facile da *estendere* per integrare nuove piattaforme: infatti, una volta definito il processo di sviluppo è sufficiente implementare i target relativi alla nuova piattaforma senza modificare i target esistenti.

Non imponendo delle dipendenze inter-piattaforma, il nuovo processo prevede delle catene meno stringenti del precedente. Tuttavia, non si può ancora considerare soddisfatto il requisito di *flessibilità*: ad esempio si supponga che l'utente voglia eseguire il target *Build* relativo alla versione .NET. In questo caso si avrebbe l'esecuzione automatica dei target da cui esso dipende (*Build Core* e *IKVM*), il cui tempo di esecuzione è tutt'altro che trascurabile. Un discorso simile può essere fatto

anche per i processi Java e Android.

La soluzione adottata è quella di eliminare le dipendenze in senso ANT, in modo da permettere l'esecuzione del singolo target senza che tutti i target da cui esso dipende vengano a loro volta eseguiti. Si delega quindi all'utente il compito di specificare esplicitamente quali target eseguire. Questo approccio da un lato fornisce la massima libertà, mentre dall'altro introduce due nuovi problemi:

- Il fatto di dover specificare i target da eseguire comporta di dover spendere più tempo per configurare il processo
- non c'è più nessuna garanzia che i target vengano eseguiti in ordine corretto: infatti è l'utente a specificare la sequenza da eseguire, il che implica che sono ammesse anche sequenze che non rispettano l'ordine dettato dalle dipendenze

Quindi, come già specificato in fasi di analisi, il nuovo processo è più complesso e non è in grado di garantire la consistenza del suo predecessore. La soluzione proposta per risolvere il problema è quella di fornire all'utente un'*interfaccia grafica* che abbia le caratteristiche indicate in fase di analisi.

Per quanto riguarda il requisito di *indipendenza da SVN*, questo viene in parte soddisfatto eliminando i target di tipo *Tag* presenti nel precedente processo. Rimane comunque il problema legato alla dipendenza dal costrutto External Definitions di SVN che viene risolto tramite i due nuovi target *Build Core* per le piattaforme Android e .NET, i quali hanno una duplice funzione:

1. Generare i file *tuprolog.jar* e *2p.jar* per le versioni .NET e Android, e inserirli nelle rispettive cartelle
2. Selezionare la versione di JDK da utilizzare per la compilazione delle classi

Il punto 1 va esattamente a sostituire l'operazione che veniva eseguita da SVN Externals, mentre il punto 2 soddisfa parte dei requisiti delle versioni Android e .NET permettendo di selezionare una specifica versione di JDK.

4.3. Organizzazione di file e cartelle

Tutti i file necessari al processo di sviluppo sono inclusi in un unica cartella di cui si riporta la struttura:

/src/ - cartella contenente i sorgenti *.java*, in particolare l'implementazione dell'interfaccia grafica

/res/ - risorse necessarie all'interfaccia grafica

/properties/ - cartella contenente i file di configurazione

/build/build-self.xml – script ANT per compilare i sorgenti dell'interfaccia grafica e generare l'archivio *antext.jar*

/release/antext.jar – archivio contenente i file *.class* relativi all'interfaccia grafica

/build.xml - lo script ANT che implementa il processo di sviluppo

4.4. Processo di sviluppo: Java

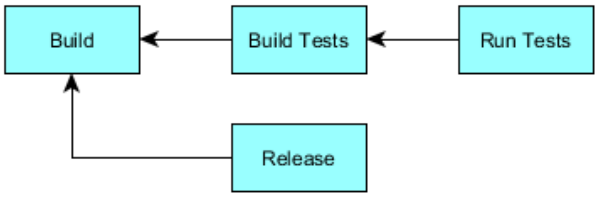


Illustrazione 4.2: Processo di sviluppo Java

Il numero di target è stato sostanzialmente ridotto rispetto al precedente processo, in modo da diminuire il tempo di esecuzione ed anche dare più libertà all'utente.

Il target *Build* compila i sorgenti e crea gli archivi *tuprolog.jar* e *2p.jar ed* è solitamente l'operazione più eseguita dallo sviluppatore, in quanto permette di verificare se ci sono errori a livello di compilazione del codice, e quindi, deve essere relativamente veloce.

Nel precedente sistema era presente il target *Package*, che faceva sostanzialmente la stessa cosa, ma costringeva l'utente a compilare anche i file di test, rendendo il processo significativamente più lungo: in particolare la compilazione dei test richiede un tempo maggiore rispetto a quello richiesto dalla generazione dei due archivi, motivo per cui si è deciso di separare i due target.

A questo punto lo sviluppatore può scegliere se generare ed eseguire i file di test, oppure creare l'archivio contente la release di tuProlog per Java.

I due target *Build Tests* e *Run Tests* vengono di solito eseguiti in sequenza, ma si è scelto di tenerli separati in quanto si possono verificare delle situazioni in cui è possibile eseguire i test compilati in precedenza.

Anche il target *Release* esegue le stesse operazioni che venivano effettuate nel precedente processo di sviluppo, tuttavia, non dipende più dalla compilazione dei test, ed inoltre non richiede di eseguire la parte relativa ad IKVM, che è stata inserita nel processo di sviluppo per .NET. Questo comporta immediatamente due vantaggi:

1. è possibile eseguire la sequenza *Build → Release*, che è molto utile nel caso si apportino modifiche minori al codice che non richiedono una nuova esecuzione dei file test
2. Non è più necessario eseguire *IKVM* per generare la release Java di tuProlog.

Si ricorda che nel precedente processo il target IKVM era incluso nel processo Java per una questione di consistenza, in quanto effettuare l'operazione *Release* faceva si che le modifiche si propagassero immediatamente anche alla versione .NET. La scelta di inserire tale operazione nel processo .NET è stata fatta sia per ridurre i tempi di esecuzione della versione Java, sia perché a livello logico è più corretto eseguire tale operazione nel momento in cui si lavora alla versione .NET.

A livello implementativo, tutti i target che effettuano compilazione di codice (*Build* e *Build Tests*) sono dotati di un meccanismo di compilazione incrementale, ossia solo i sorgenti modificati dall'ultima esecuzione vengono di fatto ricompilati, riducendo il tempo di esecuzione.

4.5. Processo di sviluppo: .NET

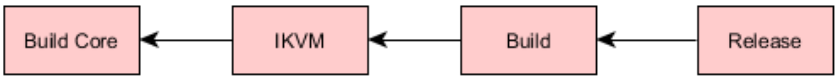


Illustrazione 4.3: Processo di sviluppo .NET

Il processo di sviluppo per la versione .NET contiene un'unica catena con 4 targets disponibili, rispetto all'unico target *Release* del procedente sistema (non si considera il target *Tag*).

In particolare si è scelto di separare le fasi di compilazione del codice (target *Build*) e generazione dell'applicazione per la distribuzione (target *Release*), che nel precedente processo erano invece eseguite da un unico target. Il motivo è l'utilità pratica: l'esecuzione del solo *Build* permette comunque di verificare il funzionamento corretto o meno dell'applicazione e si presume che lo sviluppare esegua *Release* solo a lavoro ultimato.

Il target *Build* compila soltanto il codice .NET, che al momento comprende la libreria ad oggetti di tuProlog (OOLibrary[1]) ed altri file accessori.

Il target *Build Core* genera i file *tuprolog.jar* e *2p.jar* utilizzando una specifica versione di JDK, mentre il target *IKVM* lancia l'omonimo tool per la generazione dei file *tuprolog.dll* e *2p.exe*. Si ricorda che è necessario eseguire questi target soltanto quando vengono apportate delle modiche alla versione Java di tuProlog e si vuole propagare tali cambiamenti anche alla versione .NET. Nel precedente sistema per ottenere lo stesso risultato era necessario eseguire il target *Release* della versione Java, che effettuava anche una serie di operazioni non necessarie (generazione ed esecuzione dei test, generazione dell'archivio ZIP contente la versione Java, ecc) e aveva quindi un tempo di esecuzione maggiore.

Un'altra considerazione da fare sul target *Build Core* è che elimina logicamente la dipendenza dal processo Java e rende quindi il processo più chiaro per lo sviluppatore .NET, che di fatto può lavorare in maniera autonoma utilizzando i quattro target presenti in figura 4.3.

Tutti i tool utilizzati dal processo sono configurabili tramite il file *net.properties*, che in particolare permette di specificare le posizioni di *MSBuild*[2] e *C# Compiler* (utilizzati dal target *Build*) e la versione di JDK da utilizzare (target *Build Core*).

1 Libreria di predicati che permette di utilizzare oggetti Java in un programma prolog [9]
2 Tool per la build automation per .NET e Visual Studio

4.6. Processo di sviluppo: Android

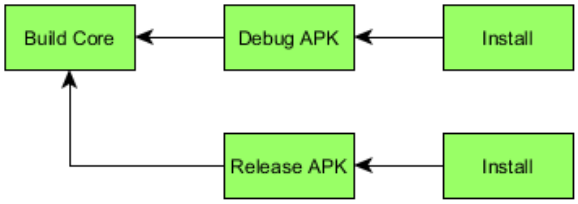


Illustrazione 4.4: Processo di sviluppo Android

I requisiti principali del processo per Android sono di eliminare la dipendenza dal plugin ADT per Eclipse e di rendere il processo completamente automatico. A tale scopo si utilizzano direttamente i tool forniti con Android SDK che permettono di gestire tutte le fasi dello sviluppo di applicazioni Android.

In particolare è possibile generare l'applicazione nelle due modalità *debug* e *release* attraverso gli omonimi target. La differenza principale tra le due è che la versione di distribuzione effettua un'ottimizzazione dell'archivio e lo firma con la chiave fornita. Inoltre, i due target eseguono automaticamente l'aggiornamento del progetto Android, generando eventuali file mancanti e aggiornando i file configurazione.

Il plugin ADT permette anche di mettere in esecuzione l'applicazione su un AVD o su un device attivo, e questa è una caratteristica che viene preservata tramite i due target *Install*, che eseguono tale operazione rispettivamente per la versione debug e release dell'applicazione.

Da notare che in quanto il processo non dipende più dal plugin ADT, e quindi da Eclipse, l'utente può lavorare con altre IDE, e questa è una caratteristica importante in ottica futura: infatti lo sviluppo del plugin è stato abbandonato da Android a favore del nuovo ambiente di sviluppo Android Studio. Questa migrazione è ancora in fase di transizione, e il plugin ADT, sebbene segnali diversi warning, è ancora funzionante, ma appare chiaro che non sarà più utilizzabile nel momento in verranno rilasciate nuove versioni di Android SDK e/o Eclipse.

Il target *Build Core* genera l'archivio *tuprolog.jar,* che funge da libreria per il progetto. Come nel caso .NET, questo target permette di separare i processi di sviluppo Android e Java. E' possibile configurare il target in modo da utilizzare una specifica versione di JDK e quindi assicurare la compatibilità con Android. Il tempo di esecuzione migliora sensibilmente rispetto al precedente processo: si ricorda che per ottenere lo stesso risultato era necessario eseguire il target *Package* incluso nella versione Java, che eseguiva una serie di operazioni non necessarie per quanto riguarda l'applicazione Android, come la generazione di *2p.jar* e la compilazione dei file di test.

Tutti i tool utilizzati dal processo devono essere configurati, motivo per cui è stato introdotto l'apposito file *android.properties*, che consente di specificare la versione delle API Android da utilizzare, l'eventuale chiave per la firma degli archivi e la versione di JDK.

4.7. Interfaccia grafica

Il nuovo processo illustrato nella sezioni precedenti offre la massima flessibilità possibile e permette quindi ridurre i tempi di esecuzione. Tuttavia, lo sviluppatore sarebbe tenuto a conoscere in dettaglio l'architettura del processo per poterne sfruttare completamente e correttamente le possibilità.

La soluzione adottata in questa tesi per eliminare la complessità introdotta nel nuovo processo consiste in un'*interfaccia grafica* avente le seguenti caratteristiche definite in fase di analisi:

- visione di insieme
- facilità di configurazione
- praticità
- consistenza

4.7.1. Funzionalità

Il requisito principale è sicuramente quello della *visione di insieme*: è necessario che fornire all'utente uno schema delle capacità messe a disposizione dal nuovo processo di sviluppo in modo veloce ed intuitivo.

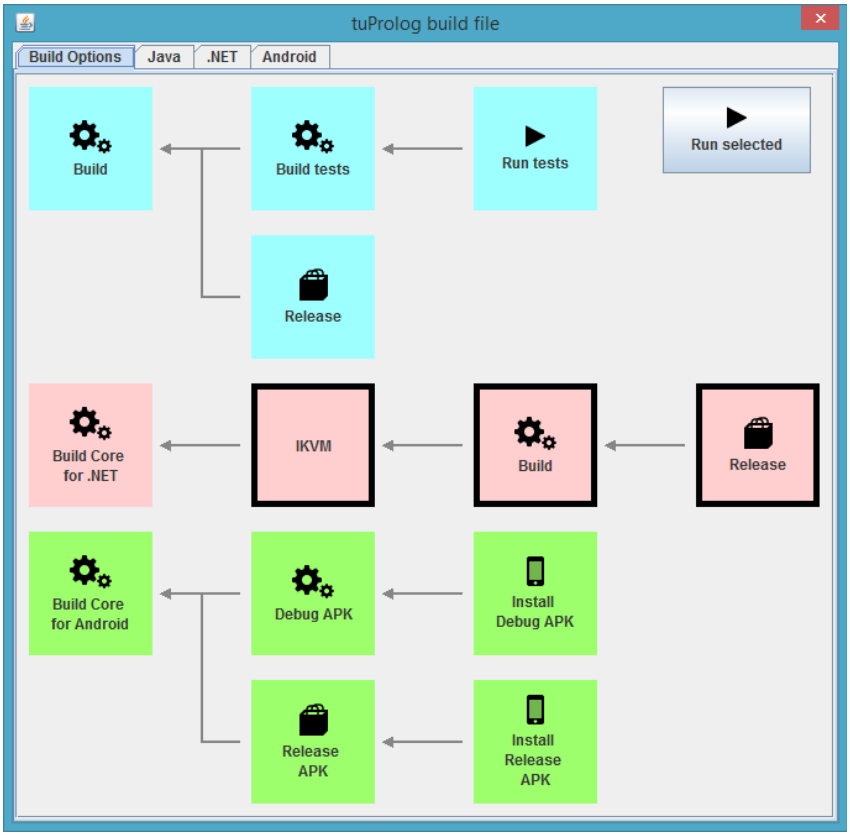


Illustrazione 4.5: Scheda "Build Options"

La scheda *Build Options* dell'interfaccia soddisfa il requisito di *visione di insieme*, in quanto riflette esattamente lo schema mostrato in figura 4.1, mostrando

tutti i target disponibili nel processo e le relative dipendenze.

Si utilizzano colori diversi per specificare le piattaforme a cui i target appartengono, in modo tale che lo sviluppatore per una specifica piattaforma sia subito in grado di determinare con quale sotto-insieme di target operare.

L'interfaccia prevede dei meccanismi di selezione singola e multipla dei target in modo da soddisfare il requisito di *praticità*, evitando quindi all'utente di dover specificare i target via ANT o Eclipse.

E' possibile mettere in esecuzione i target selezionati tramite il pulsante "Run Selected" in alto a destra, e il sistema si fa carico di assicurare la *consistenza* delle operazioni relativamente ai target selezionati: in particolare questi vengono eseguiti rispettando l'ordine mostrato in figura.

Questa soluzione non assicura tuttavia una totale consistenza: si consideri ad esempio il caso in cui l'utente elimini manualmente i file *tuprolog.jar* e *2p.jar*, che sono il prodotto del target *Build*, e successivamente esegua il target *Release*. In questo caso il sistema genera un errore dovuto alla mancanza dei due file. Per risolvere il problema si può pensare di eseguire tutti i target dipendenti da quelli selezionati: in questo caso si viola il requisito di *flessibilità* che ci si era imposti, poiché l'utente non può più selezionare un singolo target da eseguire.

Tuttavia, se lo sviluppatore non è a conoscenza dello stato attuale del sistema, l'interfaccia prevede un meccanismo per selezionare un target e tutta la relativa catena di dipendenze, in modo da assicurare la totale consistenza delle operazioni.

Per quanto riguarda la *facilità di configurazione*, è necessario fare una distinzione tra:

1. configurazione dell'architettura del processo
2. configurazione degli strumenti utilizzati dal processo

Il punto 1 viene discusso in dettaglio nella sezione 4.7.2, e si riferisce al fatto che l'interfaccia deve adattarsi ai cambiamenti che il processo può subire, quali l'integrazione di nuove piattaforme e l'introduzione di nuove feature per le piattaforme esistenti.

Il punto 2 comprende la configurazione di tutte le proprietà e i parametri necessari per poter utilizzare gli strumenti necessari al processo, quindi in particolare le diverse versioni di JDK, il tool IKVM, Android SDK, ecc.

Ogni piattaforma necessita di un determinato insieme di strumenti, e, come specificato nelle precedenti sezioni relative alle piattaforme considerate, tutti i parametri di configurazione vengono gestiti tramite dei file di configurazione con estensione *.properties*, un formato utilizzato da Java (e da ANT) per memorizzare una serie di parametri definiti come coppie chiave-valore, entrambi gestiti come stringhe. Nel contesto considerato, questi valori rappresentano generalmente percorsi di file o cartelle, parametri per compilatori, condizioni booleane, ecc.

Lo sviluppatore può ovviamente gestire questi file tramite un semplice editor di testo (per dettagli si veda la sezione 5.3.5), tuttavia, al fine di facilitarne la configurazione si è deciso di dare la possibilità all'utente di includere nell'interfaccia grafica delle schede che permettono una gestione semplificata di uno o più file *.properties* e dei parametri da questi definiti.

In particolare, si propone di gestire diversamente le modalità di input delle proprietà in base tipo di dato rappresentato: ad esempio, per selezionare file e cartelle

si utilizzano le apposite finestre, i dati booleani vengono gestiti tramite delle check-box, ecc.

Il set dei tipi di dato a disposizione è progettato per essere facilmente estendibile e quindi in grado di supportare delle modalità di input personalizzate (Per i dettagli si rimanda alla sezione 4.7.3).

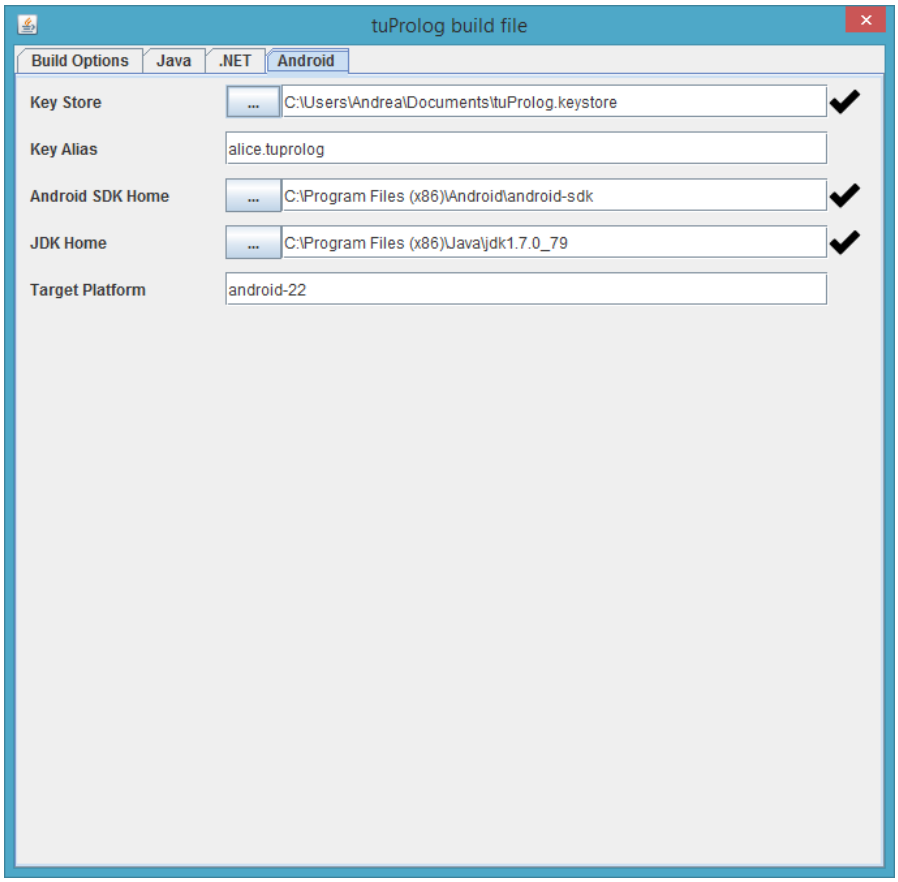


Illustrazione 4.6: Schede dei file di configurazione

In figura 4.6 è riportata la scheda relativa al file di configurazione per la piattaforma Android, che mostra come viene gestito l'input per stringhe, file e cartelle.

4.7.2. Configurazione

Come specificato in fase di analisi, l'interfaccia deve essere *facilmente configurabile* per adattarsi ai cambiamenti che il processo di sviluppo può subire nel tempo, ad esempio l'integrazione di nuove piattaforme e la modifica delle piattaforme esistenti.

Questo comporta fornire all'utente un procedimento semplice per specificare tutti i target disponibili e le relative dipendenze, e i file *.properties* da includere nelle relative schede dell'interfaccia.

La soluzione adottata consiste nell'implementare l'interfaccia tramite un *task* ANT (per dettagli si veda la sezione 5.3 relativa ad Apache ANT) in modo da integrarla con lo script che implementa il nuovo processo di sviluppo. Questo permette di definire tutte le componenti dell'interfaccia tramite il linguaggio XML, che per la sua naturale organizzazione gerarchica, risulta particolarmente pratico per la definizione di interfacce utente.

Il seguente schema mostra gli elementi XML (e la relativa organizzazione gerarchica) che permettono di definire tutte le componenti dell'interfaccia:

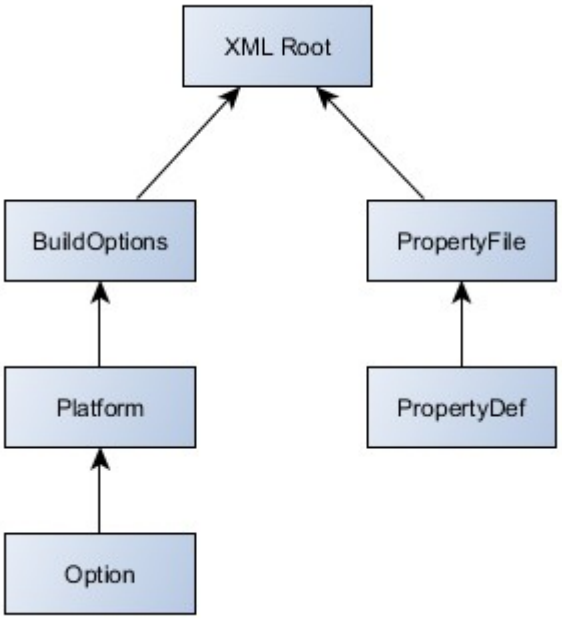


Illustrazione 4.7: Struttura XML dell'interfaccia

Ogni elemento costituisce un componente dell'interfaccia: ad esempio all'elemento *PropertyFile* corrisponde la generazione di una scheda relativa ad un file .properties specificato, mentre l'elemento *BuildOptions* permette di configurare l'omonima scheda, e quindi definire le piattaforme (*Platform*) e i target supportati (*Option*).

Nelle seguenti sezioni vengono illustrati gli elementi XML in maniera dettagliata e attraverso degli esempi pratici di utilizzo.

4.7.2.1. Elemento: PropertyFile

L'elemento *PropertyFile* specifica un file di tipo *.properties* da includere. In particolare, ogni elemento di questo tipo comporta l'inserimento nell'interfaccia della relativa scheda.

Attributo	Descrizione
path	Posizione del file
display	Titolo della scheda del file nell'interfaccia grafica

L'interfaccia caricherà automaticamente tutte le proprietà presenti nel file e ne permetterà la modifica. I dati verranno trattati come stringhe, a meno che non sia presente l'elemento innestato *PropertyDef*.

Esempio:

```
<propertyfile path=”${basedir}/default.properties”>
	<propertydef name=”app.name” display=”Application Name” type=”string”/>
	<propertydef name=”app.version” display=”Version” type=”string”/>
</propertyfile>
```

4.7.2.2. Elemento: PropertyDef

All'interno di un blocco *PropertyFile,* permette di definire alcuni parametri relativi ad una singola proprietà presente nel file.

Attributo	Descrizione
name	Nome della proprietà presente nel file
display	Testo dell'etichetta della proprietà nell'interfaccia grafica
type	Tipo di dato gestito. Attualmente i valori supportati sono: *string, boolean, file, directory.*

L'attributo *type* consente all'interfaccia di differenziare la modalità di input del valore della proprietà: i dati *boolean* vengono modellati nell'interfaccia con delle check-box, mentre per *file* e *directory*, l'interfaccia aiuta a selezionare un percorso corretto tramite un apposita finestra, ed inoltre segnala se il percorso inserito esiste o meno.

E' possibile estendere il set dei tipi di dati supportati (per i dettagli si veda la sezione 4.7.3.3)

4.7.2.3. Elemento: BuildOptions

Contiene al suo interno gli elementi necessari a definire l'omonima scheda mostrata in Figura 4.5. Tramite l'elemento innestato *Platform* è possibile definire tutte le piattaforme supportate dal processo di sviluppo e le relative opzioni (target) di sviluppo.

Esempio:

```
<buildoptions>
	<platform name="java" display="Java" color="#99FFFF">
		. . .
	</platform>
	<platform name="net" display=".NET" color="#FFCCCC">
		. . .
	</platform>
	<platform name="android" display="Android" color="#99FF66">
		. . .
	</platform>
</buildoptions>
```

4.7.2.4. Elemento: Platform

Definisce una piattaforma, intesa come un insieme di catene di opzioni di sviluppo, che possono essere definite tramite l'elemento innestato *Option.*

Attributo	Descrizione
name	Nome univoco della piattaforma
display	Riservato per uso futuro
color	Colore delle opzioni di sviluppo per questa piattaforma

4.7.2.5. Elemento: Option

Costituisce un opzione di sviluppo relativamente all'elemento *Platform* in cui è inserito. Oltre a vari parametri grafici è possibile definire un insieme di target da eseguire.

Attributo	Descrizione
name	Nome univoco dell'opzione
display	Testo dell'opzione
targets	Lista dei targets ANT separati da ",” da invocare quando l'opzione viene eseguita.
icon	Immagine dell'opzione

Il seguente codice definisce il platform “Android” e al suo interno le due opzioni indipendenti “Debug” e “Release”:

```
<platform name=”android” color=”#99FF66”>
	<option name=”debug” targets=”init,debug” display=”Debug” icon=”gears”/>
	<option name=”release” targets=”init,release”
		display=”Release” icon=”shopping-bag”/>
</platform>
```

E' possibile creare delle catene di opzioni innestando ricorsivamente elementi di tipo *Option*. Ad esempio:

```
<platform name="android" display="Android" color="#00ff00">
	<option display="Build" icon="gears">
		<option display="Debug APK" icon="gears">
			<option display="Install Debug APK" icon="phone" />
		</option>
		<option display="Release APK" icon="shopping-bag">
			<option display="Install Release APK" icon="phone"/>
```

```
            </option>
        </option>
</platform>
```

Per semplificare la lettura sono stati omessi alcuni attributi degli elementi *Option.* Il codice crea la piattaforma "Android", e definisce la seguente struttura di opzioni:

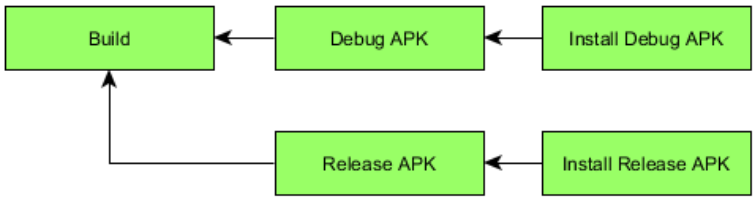


Illustrazione 4.8: Esempio di processo di sviluppo per Android

4.7.3. Estendibilità

In questa sezione vengono descritte le modalità di estensione e personalizzazione dell'interfaccia: in particolare ci si riferisce alla possibilità di aggiungere nuove piattaforme (come nel caso di tuProlog e iOS) e di rendere l'utilizzo più semplice ed intuitivo definendo nuove modalità per l'input delle proprietà.

4.7.3.1. Aggiungere nuove piattaforme

La semplice sintassi XML rende semplice l'estensione dell'interfaccia descritta per includere nuove piattaforme. Ad esempio, nel caso di tuProlog è presente la versione per iOS, non inserita nel processo sviluppato in questa tesi:

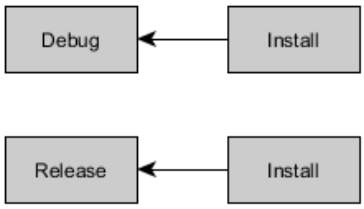


Illustrazione 4.9: Processo di sviluppo per iOS

In questo caso, Apple fornisce insieme ad XCode[1] una serie di tool invocabili da riga di comando simili a quelli inclusi in Android SDK, e quindi possiamo ipotizzare una catena di target simile a quella vista per Android.

Il seguente codice aggiunge la piattaforma "iOS" e le opzioni di sviluppo definite in figura 4.9:

```
<platform name="ios" display="iOS" color="#cccccc">
```

1 IDE per lo sviluppo di applicazioni iOS e MacOS

```
	<option name=”debug-ios” display=”Debug”>
		<option name=”install-debug-ios” display=”Install”/>
	</option>
	<option name=”release-ios” display=”Release”>
		<option name=”install-release-ios” display=”Install”/>
	</option>
</platform>
```

4.7.3.2. Immagini per gli elementi Option

Le immagini in formato PNG presenti all'interno della cartella **res** vengono automaticamente caricate e sono disponibili per essere usate come icone per gli elementi *option* tramite l'attributo *icon*, utilizzando come valore il nome del fine privo dell'estensione.

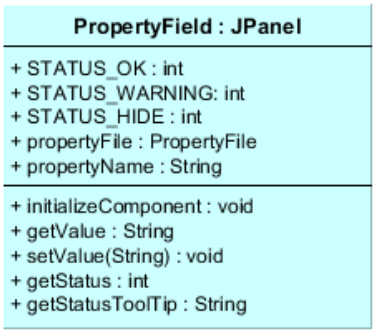
Ad esempio, includendo nella cartella **res** i file *myicon1.png* e *myicon2.png* , è possibile utilizzarli come icone tramite il seguente codice:

```
<option name=”option-1” display=”My First Option” icon=”myicon1”>
	<option name=”option-2” display=”My Second Option” icon=”myicon2”/>
</option>
```

Da ricordare che la cartella **res** deve essere inclusa nell'archivio *antext.jar*, quindi, nel momento in cui vengono inserite nuove immagini è necessario eseguire lo script **build-self.xml** per generare nuovamente l'archivio.

4.7.3.3. Nuovi tipi di dato per le proprietà

E' possibile aggiungere nuovi tipi di dato estendendo la classe astratta *antext.ui.PropertyField:*



Questa è un panello Java Swing (*JPanel*) che utilizza un *BorderLayout* per disporre i componenti al suo interno. Sono automaticamente inclusi i seguenti elementi:

- sulla sinistra (*BorderLayout.EAST*) un'etichetta dove viene visualizzato il nome della proprietà
- sulla destra (*BorderLayout.WEST*) un'icona di stato, che specifica se il valore inserito è corretto o meno.

Lo sviluppatore deve aggiungere nella parte centrale del pannello (*BorderLayout.CENTER*) gli elementi necessari per la visualizzazione e la modifica del valore della proprietà, effettuando l'override del metodo astratto *initializeComponent*.

Ad esempio, nel semplice caso in cui si voglia gestire delle stringhe, è necessario aggiungere un campo di testo:

```
@Override
public void initializeComponent() {
	myTextField = new JTextField();
	add(myTextField, BorderLayout.CENTER)
}
```

Gli altri metodi astratti da implementare sono i seguenti:

- *setValue/getValue*: permettono di impostare/reperire il valore della proprietà. Da notare che entrambi prevedono che il dato sia gestito come stringa, poiché i file *.properties* sono file di testo
- *getStatus: restituisce un intero che specifica se il valore inserito è corretto (PropertyField.STATUS_OK) o meno (PropertyField.STATUS_WARNING).* Nel caso in cui il dato non richieda alcun controllo di correttezza è possibile usare il valore *PropertyField.STATUS_HIDE.*
- *getStatusToolTip*: restituisce una stringa che specifica il messaggio di errore da visualizzare se il dato inserito non è corretto.

Una volta completata l'implementazione della classe per il nuovo tipo di dato, questa deve essere aggiunta alla HashMap *PropertyField.DATA_TYPES*, che contiene i tutti i tipi consentiti e riconosciuti dall'applicazione:

```
class PropertyField extends JPanel {

	. . .

	static {
		DATA_TYPES = new HashMap<>();

		DATA_TYPES.put("string", StringPropertyField.class);
		DATA_TYPES.put("boolean", BooleanPropertyField.class);
		DATA_TYPES.put("file", FilePropertyField.class);
		DATA_TYPES.put("directory", DirectoryPropertyField.class);

		DATA_TYPES.put(“mydatatype”, MyDataTypePropertyField.class);
	}

	. . .
```

```
}
```

A questo punto è necessario ricompilare i sorgenti tramite lo script **build-self.xml**, e il valore “mydatatype” può essere utilizzato per l'attributo *type* degli elementi *Option*.

5. Strumenti utilizzati

In questo capitolo vengono descritti in dettaglio tutti gli strumenti utilizzati per l'implementazione del processo di sviluppo, in modo da fornire le basi per il loro utilizzo pratico.

In particolare quindi vengono illustrati *JDK*[1] e *IKVM*[2], che venivano impiegati anche nel procedente processo di sviluppo, mostrando come utilizzare i rispettivi compilatori e run-time da riga di comando.

Particolare attenzione viene dedicata ad *Apache ANT*[3], del quale, oltre a definirne i concetti di base, si mostrano alcune delle best-practice da adottare, e si fanno alcuni esempi di utilizzo in casi reali. Una sezione è dedicata completamente alle modalità di estensione dei *task*, metodo utilizzato in questa tesi per l'implementazione dell'interfaccia grafica illustrata nel capitolo 4.

L'ultima sezione è dedicata ad *Android SDK*[4], che fornisce tutti gli strumenti per lo sviluppo di applicazioni Android. Nel precedente processo di sviluppo questo pacchetto era necessario per generare l'installer dell'applicazione, tuttavia, veniva utilizzato indirettamente tramite il plugin ADT[5] di Eclipse. Il nuovo processo utilizza direttamente i tool forniti, e quindi è necessario dedicare una sezione per illustrarne le possibilità e le modalità d'uso.

1 Java Development Kit [12]
2 Tool per la conversione di bytecode Java in bytecode .NET [6]
3 Strumento per rendere automatici i processi di sviluppo software [3]
4 Software Development Kit per applicazioni Android [10]
5 Plugin per Eclipse per lo sviluppo di Applicazioni Android

5.1. JDK

JDK è il pacchetto distribuito da Oracle per gli sviluppatori Java. Al suo interno è incluso il JRE (Java Runtime Environment), necessario ad eseguire applicazioni e applets scritte in Java, e tutti i tool necessari allo sviluppatore, come compilatore, debugger, ecc.

In questa sezione verranno fornite le basi per il loro utilizzo; in particolare verranno mostrati il compilatore *javac*[1] e il runtime *java*[2]. La documentazione completa di questi e di tutti gli altri tool è fornita con JDK e anche consultabile online sul sito ufficiale (http://docs.oracle.com/en/java).

5.1.1. Installazione

La versione più recente è JDK 8 (o JDK 1.8), che andremo ad utilizzare per compilare la versione Java di tuProlog. Tuttavia avremo bisogno anche di JDK7 (o 1.7) per quanto riguarda la parte Android. Entrambe le versioni sono reperibili sul sito ufficiale di Oracle.

Versioni diverse di JDK possono coesistere in quando stanno in directory separate, quindi non c'è alcun problema di conflitti. Da adesso in poi ci riferiremo alle directory di installazione con *<jdk_home>* per JDK e *<jre_home>* per il runtime.

I tool citati sopra si trovano nella cartella *<jdk_home>/bin*, quindi, se si desidera utilizzarli da riga di comando, è necessario aggiungere tale cartella alla variabile d'ambiente *PATH*.

5.1.2. Compilatore

Tra i tool è presente il compilatore *javac*. Il suo utilizzo di base è molto semplice; supponiamo di aver creato il seguente file *Hello.java*:

```
// Hello.java

public class Hello {
	public static void main(String[] args) {
		System.out.println("Hello, Java!");
	}
}
```

Per compilarlo, posizionarsi con una shell nella cartella dove risiede e lanciare il comando:

```
javac Hello.java
```

Questo genererà il file .class compilato, *Hello.class.*

Un argomento importante è *-classpath path1;path2;..* (o *-cp*), che permette di specificare dove il compilatore deve andare a cercare le librerie/classi referenziati. I vari *path1,path2*, ecc possono essere archivi zip o jar, o cartelle dove cercare file

1 [14] Java programming language compiler
2 [13] Java runtime

.class.

Supponendo quindi di avere un progetto java con la seguente struttura di directory:

/src/ - sorgenti .java

/bin/ - destinazione per i file .class

/lib/ - cartella contenete le librerie necessarie

per compilarle questo progetto ci si posiziona sulla root del progetto e si lancia il comando:

```
javac src/*.java -d bin -classpath /lib/*.jar
```

5.1.3. Archivi JAR

Il formato JAR (**J**ava **AR**chive) è utilizzato per creare dei package contenenti classi Java. Utilizza internamente il tool *zip* per comprimere l'archivio e può essere utilizzato sia per creare dei package di tipo libreria, sia per applicazioni (JAR eseguibile).

In particolare, nell'archivio è definito il file */META-INF/MANIFEST.MF* che contiene dei metadati, organizzati in diverse sezioni e come coppie (nome, valore).

Supponendo di avere a disposizione la classe *Hello.java* definita nella sezione precedente (5.1.2) e di averla già compilata con *javac*, è possibile creare un archivio JAR utilizzando il seguente comando:

```
jar -cf hello.jar Hello.class
```

L'opzione **c** sta per "create" e l'opzione **f** indica che vogliamo specificare un file di destinazione (altrimenti il comando *jar* va a scrivere sullo standard output).

Di seguito si inserisce il nome dell'archivio "hello.jar", e infine, tutti i file che si desidera includere (in questo caso, solo *Hello.class*).

Il tool crea automaticamente il file Manifest, anche se, in questo caso non conterrà alcuna proprietà.

Il file JAR appena creato non ha un *entry-point*, ossia non è specificata la classe che contiene il metodo *main*, da eseguire quando viene lanciata l'applicazione. JAR di questo tipo, ossia senza entry-point, si utilizzano quando si vuole creare una libreria di classi per altre applicazioni.

Per fare in modo che la classe *Hello* (o meglio il suo metodo *main*) sia utilizzata come entry-point, si esegue il seguente comando:

```
jar -cfe hello.jar Hello Hello.class
```

L'opzione **e** specifica appunto che vogliamo definire l'entry-point dell'archivio, in questo caso *Hello* (senza l'estensione .class).

5.1.4. Esecuzione

JDK mette a disposizione il tool *java*, che permette di mettere in esecuzione applicazioni scritte in Java.

Supponendo sempre di avere la classe *Hello.java* definita nella sezione 5.1.2 e il corrispondente file *Hello.class* compilato, per metterlo in esecuzione lanciare il comando:

```
java Hello
```

Se invece abbiamo un JAR eseguibile (ad esempio *hello.jar*) il comando da eseguire è:

```
java -jar hello.jar
```

Come per il compilatore *javac,* è possibile specificare il *classpath*:

```
java -classpath mylib1.jar;mylib2.jar Hello
```

5.2. IKVM

IKVM è un'implementazione della Java Virtual Machine (JVM) per i runtime .NET e Mono che permette la traduzione automatica da bytecode Java a bytecode .NET. Ci sono due modi di utilizzo:

1. dinamico – in questa modalità i file .class e gli archivi JAR vengono utilizzati direttamente nel runtime .NET. Il bytecode Java viene tradotto "al volo" in CIL (Common Intermediate Language) ed eseguito senza ulteriori passi necessari
2. statico – questa modalità permette l'utilizzo di codice Java in applicazioni .NET. Per rendere possibile ciò è necessario che i file .class e JAR vengano convertiti in DLL (o eseguibili).

5.2.1. Modalità dinamica

Supponiamo di avere lo stesso file *Hello.java* definito nella sezione 5.1.2 e di averlo già compilato con *javac*, e quindi aver creato il corrispondente *Hello.class*.

Per eseguire l'applicazione con IKVM in modalità dinamica, posizionarsi con una shell nella cartella dove è contenuto il file *Hello.class* e lanciare il comando:

```
ikvm Hello
```

Allo stesso modo è possibile eseguire dei JAR che al loro interno definiscono l'attributo *Main-Class*:

```
ikvm -jar hello.jar
```

5.2.2. Modalità statica

IKVM include il tool *ikvmc* (IKVM Compiler) che converte archivi JAR in DLL o eseguibili .NET. Per compilare il file *hello.jar* lanciare il seguente comando:

```
ikvmc hello.jar
```

L'output sarà il file *hello.exe* nel caso in cui *hello.jar* definisca un entry-point, oppure *hello.dll*.

Le DLL e gli eseguibili compilati in questo modo si aspettano di trovare le librerie *IKVM.OpenJDK.*.dll* nella directory corrente o nella Global Assembly Cache. Quindi se lanciando *hello.exe* si riscontra un errore del tipo *FileNotFound*, installare le DLL nella GAC oppure copiarle nella directory corrente.

5.3. Apache ANT

Apache ANT è uno strumento sviluppato in Java che permette di automatizzare il processo di build di applicazioni, anche se complesse e multi-piattaforma con nel caso di tuProlog. Il fatto di essere sviluppato in Java offre il vantaggio che è possibile estendere la capacità espressiva di ANT tramite opportune classi Java, feature che viene sfruttata in questa tesi per l'implementazione dell'interfaccia grafica del processo di sviluppo.

E' possibile scaricare ANT dal sito ufficiale (https://ant.apache.org) sotto forma di archivio. Scompattare l'archivio in una cartella (ad esempio, "C:\Program Files (x86)\Apache ANT" per Windows) che da adesso in poi indicheremo con *<ant_home>.*

Per approfondimenti su tutti gli aspetti trattati si rimanda alla documentazione ufficale[1] di Apache ANT.

5.3.1. Esecuzione di uno script

ANT offre un file batch per l'esecuzione di script da riga di comando. Per poterlo eseguire è necessario aggiungere alla variabile d'ambiente *PATH* il percorso *<ant_home>/bin*.

Dato che ANT è sviluppato in Java, occorre anche includere il percorso di JRE (o JDK) nella variabile d'ambiente *PATH*.

Fatto ciò, aprire una shell e posizionarsi in una cartella che contiene uno script ANT con nome "build.xml", che ANT riconosce automaticamente come script da eseguire. E' quindi sufficiente eseguire il comando:

```
ant
```

e lo script *build.xml* verrà automaticamente eseguito utilizzando il target (vedi sezione 5.3.2 per la definizione di *target)* di default.

E' anche possibile specificare il target da eseguire. Supponiamo che il file di build abbia un target chiamato *release*. Per eseguirlo basta lanciare il comando:

```
ant release
```

5.3.2. Organizzazione di uno script

Gli script ANT sono scritti in linguaggio XML. La root del file è costituita dall'elemento *project*:

```
<project name="MyProject" default="help" basedir="..">
      <!-- Inserire il codice -->
</project>
```

L'elemento project ha tre attributi:

1 Apache Ant User Manual [7]

1. **name** – nome del progetto
2. **default** – target di default per il progetto (vedi dopo)
3. **basedir** – la directory con la quale vengono fatti tutti i calcoli per path **relativi** dei file presenti nello script

5.3.2.1. Targets

Ogni progetto definisce diversi elementi di tipo *target* al suo interno. Un *target* non è altro che una sequenza di azioni (*task*) che devono essere eseguite.

L'attributo **default** dell'elemento *project* definisce appunto il target da lanciare quando si esegue lo script.

E' possibile impostare relazioni di dipendenza tra i target. Considerare il seguente esempio:

```
<project name=”myProject” basedir=”..” default=”build”>
	<target name=”init”>
		<!-- Inizializzazione dello script -->
	</target>
	<target name=”build” depends=”init”>
		<!-- Build del progetto -->
	</target>
</project>
```

Il target *build* dipende dal target *init*, quindi se si esegue lo script, ANT cercherà di lanciare il target di default, ossia *build*, ma prima è necessario che tutti i target da cui questo dipende vengano a loro volta eseguiti. Quindi la sequenza di esecuzione è, in questo caso: *init → build*

Il meccanismo avviene in modo ricorsivo e l'ordine con cui i vari target vengono eseguiti non è sempre banale come nel caso riportato sopra. Considerare il seguente esempio:

```
<project name=”myProject” basedir=”..” default=”D”>
	<target name="A"/>
	<target name="B" depends="A"/>
	<target name="C" depends="B"/>
	<target name="D" depends="C,B,A"/>
</project>
```

Si potrebbe pensare che la sequenza sia: *C → B → A → D*. Non è così. Ecco ciò che in realtà succede:

1. ANT tenta di eseguire D, che dipende da C
2. ANT tenta di eseguire C, che dipende da B
3. ANT tenta di eseguire B, che dipende da A
4. ANT esegue A

5. ANT risale la catena ed esegue B
6. ANT risale la catena ed esegue C
7. ANT risale la catena ed esegue D. Da notare che questa operazione non esegue nuovamente i target B e C, in quanto sono già stati eseguiti dai passi precedenti.

La sequenza corretta è: *A → B → C → D*. Quindi le dipendenze ci assicurano che tutti i target siano (ricorsivamente) eseguiti. Tuttavia se si vuole controllarne l'ordine di esecuzione è necessario prestare particolare attenzione all'impostazione delle dipendenze.

5.3.2.2. Tasks

Ogni *target* contiene al proprio interno dei *task*, ossia delle porzioni di codice da eseguire. ANT include un vasto insieme di azioni predefinite a disposizione dell'utente, come ad esempio copiare file, creare archivi .zip o .jar, compilare codice Java, eseguire programmi esterni, ecc. In questa sezione verrà descritto il concetto di task attraverso alcuni esempi.

Un *task* è un elemento XML dello script che può prevedere degli attributi e ammettere degli elementi innestati. Un task molto importante è *property*:

```
<target name=”init”>
       <property name=”src.dir” value=”src”/>
       <property name=”bin.dir” value=”bin”/>
</target>
<target name=”build” depends=”init”>
       <echo>SRC dir is ${src.dir)</echo>
       <echo>BIN dir is ${bin.dir}</echo>
</target>
```

Il task *property* valorizza una proprietà del progetto, definita dalla coppia (**name**, **value**). La proprietà è accessibile nelle altre parti dello script tramite la sintassi *${<nome proprietà>}*.

Possiamo anche utilizzare il task *property* per caricare un file contenente coppie (nome, valore) che vengono aggiunte al progetto come proprietà:

```
. . .
<property file=”ant.properties”/>
. . .
```

In questo modo tutte le proprietà contenute nel file *ant.properties* vengono aggiunte al progetto e ci si può riferire al loro valore con la sintassi indicata sopra. Per quanto riguarda la struttura di un file *.properties* si rimanda alla sezione 5.3.5.

Un altro task che verrà utilizzato spesso è *copy*, che permette di copiare file e directory:

```
<target name=”copy-my-files”>
	<property name=”my.file” value=”miofile.txt”/>
	<property name=”dest.dir” value=”C:\temp”/>
	<copy file=”${my.file}” todir=”${dest.dir}”/>
</target>
```

Si utilizza l'attributo **file**, che specifica il file da copiare e l'attributo **todir** che specifica la directory di destinazione. Questo metodo risulta poco pratico se si deve copiare molti file, per esempio, l'intero contenuto di una directory. Per questo, il task *copy* ammette elementi innestati di tipo *fileset*, che permettono di definire un insieme di file/cartelle con poche righe di codice:

```
<target name=”copy-my-files”>
	<property name=”dest.dir” value=”C:\dest”/>
	<property name=”source.dir” value=”C:\source”/>
	<copy todir=”${dest.dir}”>
		<fileset dir=”${source.dir}”/>
			<include name=”**/*.java”/>
			<exclude name=”**/*Test.java”/>
		</fileset>
	</copy>
</target>
```

Nell'esempio riportato sopra si vuole copiare i file contenuti in “C:\source” nella cartella “C:\dest”, ma soltanto quelli con estensione “.java” e escludendo i file di test. Un file viene incluso in un fileset se:

1. E' specificato in un elemento di tipo *include*
2. Non è specificato in alcun elemento di tipo *exclude*

Quindi, tornando all'esempio, l'elemento *include* tenta di includere tutti i file “.java” nella cartella “C:\source” e sottocartelle, ma l'elemento *exclude* esclude quelli il cui nome termina con “Test.java”.

Un altro task che particolarmente usato in questa tesi è *javac*, che appunto ci permette di invocare il compilatore *javac.* Anche questo ammette degli elementi innestati di tipo *fileset* per specificare la posizione dei sorgenti:

```
<javac destdir="${build}" classpath="xyz.jar">
    <src path="${src}"/>
    <src path="${src2}"/>
    <include name="mypackage/p1/**"/>
    <include name="mypackage/p2/**"/>
    <exclude name="mypackage/p1/testpackage/**"/>
</javac>
```

L'attributo **destdir** imposta la destinazione dei file compilati.

Gli elementi innestati di tipo *src* definiscono le cartelle dove andare a cercare i file sorgenti. Gli elementi di tipo *include* e *exclude* sono analoghi a quelli visti per i fileset.

Per determinare la posizione fisica del compilatore *javac*, ANT esamina il valore della variabile d'ambiente *JAVA_HOME*, che solitamente non punta alla directory di installazione di JDK, ma a quella di JRE, il che porta ad errori in quanto il compilatore "javac" non è incluso nel runtime.

Ci sono vari modi per ovviare a questo problema. Il più immediato è modificare la variabile *JAVA_HOME*. Tuttavia, considerando anche il caso di tuProlog, questo non è il metodo più indicato, in quanto è necessario sia il compilatore di JDK8 che quello di JDK7. E' possibile specificare la posizione del compilatore *javac* come attributo dell'omonimo task:

```
<javac destdir="${build}"
       classpath="xyz.jar"
       fork="yes"
       executable="${jdk7.dir}/bin/javac">
    <src path="${src}"/>
    <src path="${src2}"/>
    <include name="mypackage/p1/**"/>
    <include name="mypackage/p2/**"/>
    <exclude name="mypackage/p1/testpackage/**"/>
</javac>
```

L'attributo **executable** permette di specificare il percorso del compilatore da utilizzare, e quindi rende possibile l'utilizzo di diverse versioni di JDK all'interno dello stesso script.

Per la documentazione completa sui tutti i task disponibili si rimanda al sito ufficiale di ANT (https://ant.apache.org/manual/)

5.3.2.3. Macrodef

I blocchi *macrodef* permettono di definire delle porzioni di codice utilizzabile più volte (ossia delle macro) all'interno dello stesso build file. Un *macrodef* supporta l'elemento innestato *attribute*, che permette di definire un parametro della macro. Le azioni da eseguire vengono inserite all'interno dell'elemento innestato *sequential*.

```
<macrodef name="build-java">
       <attribute name="srcdir"/>
       <attribute name="jardest"/>
       <sequential>
              <delete dir="build/classes"/>
              <mkdir dir="build/classes"/>
              <javac srcdir="@{srcdir}" destdir="build/classes"/>
              <jar basedir="build/classes" destfile="@{jardest}"/>
       </sequential>
```

```
</macrodef>
```

L'esempio riportato sopra definisce una semplice macro che compila i sorgenti di un progetto Java e poi crea un archivio JAR.

La macro definisce due attributi che specificano il path della cartella dei sorgenti e il nome del file jar da generare. Da notare che, all'interno della macro ci si riferisce agli attributi attraverso la sintassi *@{<nome_attributo>}*, simile a quella vista per *property*. Una volta definita, è possibile invocare la macro da qualsiasi target dello script file nel seguente modo:

```
<target name="build-all">
	<build-java srcdir="src" destjar="package.jar"/>
	<build-java srcdir="test" destjar="test-package.jar"/>
</target>
```

Quindi la macro diventa di fatto un nuovo task, il cui nome riflette il valore dell'attributo **name** nella definizione della macro stessa. In questo esempio la macro viene utilizzata per compilare separatamente i sorgenti e i file di test di un ipotetico progetto e generare i due corrispondenti archivi JAR.

5.3.3. Compilare un progetto Java con Apache ANT

In questa sezione viene data un'idea di come scrivere uno script per ANT che permetta fare il build di un intero progetto Java.

Si suppone di avere un progetto con la seguente struttura di directory e file:

- **/src/** - cartella dei sorgenti
- **/lib/** - cartella delle librerie da includere nel progetto
- **/build/classes/** – cartella dove inserire i file .class compilati
- **/build/archives/** – cartella dove inserire gli archivi JAR
- **/build.xml** – il build file che andremo a scrivere

Essendo questo un progetto relativamente semplice si utilizza uno script ANT con tre target:

```
<project name="MyAPP" basedir="." default="build">
	<target name="init">
		<!-- Init properties -->
	</target>
	<target name="compile" depends="init">
		<!-- Compile Java sources -->
	</target>
	<target name="build" depends="compile">
		<!-- Build JAR archives -->
	</target>
```

```
</project>
```

La sequenza di dipendenze è: *init → compile → build*

Nel target *init* si inserisce l'inizializzazione dello script:

```
<target name="init">
		<!-- Init propierties -->
		<property name="main.class" value="myapp.Main"/>

		<property name="src.dir" value="src"/>
		<property name="lib.dir" value="lib"/>
		<property name="build.dir" value="build"/>
		<property name="build.classes.dir"
			value="${build.dir}/classes"/>
		<property name="build.archives.dir"
			value="${build.dir}/archives"/>

</target>
```

In particolare l'intera struttura di cartelle del progetto viene mappata in apposite proprietà. Questa prassi è particolarmente consigliata, poiché questi valori vengono utilizzati più volte all'interno dello script.

Si definisce inoltre una proprietà *main.class contenente* l'entry-point dell'applicazione.

```
<target name="compile" depends="init">
	<!-- Compile Java sources -->
	<javac srcdir="${src.dir}" destdir="${build.classes.dir}">
		<classpath>
			<fileset dir="${lib.dir}" includes="*.jar"/>
		</classpath>
	</javac>
</target>
```

Il target *compile* ha il compito di compilare i sorgenti del progetto. Si utilizza ovviamente il task *javac,* configurato per compilare i file presenti nella cartella *src* e inserire le classi compilate nella cartella *build/classes*.

Da notare l'elemento innestato *classpath*, che permette di impostare l'omonimo parametro del compilatore (in questo caso, tutti i file .jar inclusi nella cartella *lib*).

```
<target name="build" depends="compile">
	<!-- Build JAR archives -->
```

```
      <jar destfile="${build.archives.dir}/myapp.jar">


            <fileset dir="${build.classes.dir}"
            includes="**/*.class"/>

            <zipgroupfileset dir="${lib.dir}" includes="*.jar"/>
            <manifest>
                  <attribute name="Main-Class"
                        value="${main.class}"/>

                  <attribute name="Class-Path" value="."/>
            </manifest>
      </jar>
</target>
```

Il target *build* ha lo scopo di creare l'archivio *myapp.jar*, facendo il merge tra le classi compilate del progetto (contenute nella cartella *build/classes*) e quelle delle librerie necessarie (contenute negli archivi JAR della cartella *lib*).

Il blocco **manifest** permette di impostare le proprietà contenute nel Manifest del JAR. In questo caso viene configurato l'attributo "Main-Class" (ossia, l'entry-point). L'archivio creato è un JAR eseguibile tramite la sintassi definita nella sezione 5.1.4.

5.3.4. *Estensione di un task*

Oltre che usando i *macrodef*, è possibile creare un task personalizzato estendendo la classe *org.apache.tools.ant.Task:*

```
package com.andrea.bucaletti.tasks;
public class MyFirstTask extends org.apache.tools.ant.Task {
	. . .
}
```

Per utilizzare *MyFirstTask* in uno script è necessario definirlo all'interno dello script stesso:

```
<target name="execute-my-task">
	<taskdef name="my-first-task" class="com.andrea.bucaletti.tasks.MyFirstTask"
	classpath="classes"/>

	<my-first-task>
</target>
```

Il blocco *taskdef* definisce:

1. il nome con cui ci riferiremo al task nello script (attributo **name**)
2. la classe Java che implementa il task (attributo **class**)
3. il percorso o l'archivio JAR dove il task è definito (attributo **classpath**). Solitamente, se il task è complesso, servono più classi per implementarne il comportamento, quindi, è opportuno creare un archivio JAR che contenga il task e tutte le classi accessorie necessarie all'esecuzione

A questo punto, il blocco *my-first-task* viene riconosciuto da ANT, che provvede a istanziare il task e ad invocarne il metodo *execute*.

Un task può prevedere anche degli attributi; ad esempio il task *javac* può avere gli attributi **srcdir**, **destdir**, ecc. Per inserire degli attributi in un task personalizzato, ANT si aspetta che vengano definiti dei metodi di tipo *setter*. Ad esempio, se il task ha un attributo **name**, è necessario definire un metodo *setName(String)* nella classe che implementa il task:

```
package com.andrea.bucaletti.tasks;
public class MyFirstTask extends ogn.apache.tools.ant.Task {
	protected String name;

	public void setName(String name) {
			this.name = name;
	}
}
```

Per valorizzare l'attributo *name* nello script ANT:

```
. . .
<my-first-task name=”Andrea”/>
. . .
```

ANT riesce automaticamente a determinare quale metodo invocare tramite Reflection. Se il metodo *setName* non fosse definito, o ci fossero degli errori di battitura, si verificherebbe un errore del tipo “l'attributo *name* non è supportato dal task”.

Allo stesso modo degli attributi (ossia tramite Reflection), ANT permette di definire gli elementi innestati all'interno del task. Si suppone di voler creare l'elemento innestato *friend*:

```
<my-first-task name=”Andrea”>
	<friend/>
	<friend/>
</mytask>
```

Per fare ciò, ANT si aspetta che venga definito all'interno della classe MyFirstTask il metodo *createFriend:*

```
package com.andrea.bucaletti.tasks;
public class MyFirstTask extends org.apache.tools.ant.Task {
	protected ArrayList<Friend> friends;
	protected String name;

	public MyFirstTask {
			friends = new ArrayList<>();
	}

	public void setName(String name) {
			this.name = name;
	}

	public Friend createFriend() {
			Friend f = new Friend();
			friends.add(f);
			return f;
	}
}
```

Qui ci sono diverse considerazioni da fare:

1. Il metodo *createFriend* restituisce un oggetto di tipo *Friend*. Quindi è necessario implementare quell'oggetto tramite un'altra classe Java, **tuttavia non è**

necessario che abbia lo stesso nome dell'elemento che vogliamo definire.

2. La classe *Friend* può essere qualsiasi cosa: in particolare non è necessario estendere una particolare classe come avviene per il task.
3. Da notare che all'interno del metodo *createFriend* aggiungiamo l'elemento appena creato ad una collection interna al task (*ArrayList<Friend> friends*). Questa è una delle best-practice consigliate dal manuale ufficiale di ANT, e permette di tenere traccia di tutti gli elementi interni di tipo *friend* che sono stati inseriti nel task.

A questo punto, si vorrebbe che anche gli elementi di tipo *friend* avessero i propri attributi e i propri elementi innestati:

```
<my-first-task name=”Andrea”>
	<friend name=”Paolo”>
		<like value=”soccer”/>
	</friend
	<friend name=”Marco”>
		<like value=”Beatles”/>
		<like value=”basketball”/>
	</friend>
</mytask>
```

La procedura da seguire è esattamente la stessa indicata sopra, quindi:

1. Si implementa la classe *Friend* (o il nome che si preferisce, non è importante)
2. Al suo interno definiamo i metodi *setName(String)* e *createLike()*
3. Si implementa la classe *Like*, definendo al suo interno il metodo *setValue(String)*

```
// Friend.java
public class Friend {
	protected String name;
	protected ArrayList<Like> likes;

	public void setName(String name) {
		this.name = name;
	}

	public Like createLike() {
		Like l = new Like();
		likes.add(l);
		return l;
	}
}

//Like.java
public class Like {
```

```
	protected String value;

	public void setValue(String value) {
			this.value = value;
	}
}
```

Una volta definita la struttura del task, è necessario specificare le azioni da eseguire implementando il metodo *execute()*:

```
package com.andrea.bucaletti.tasks;
public class MyFirstTask extends org.apache.tools.ant.Task {
	protected ArrayList<Friend> friends;
	protected String name;

	. . . . . .

	public void execute() {
		for(Friend f : friends) {

			System.out.print(this.name +
				" is friend with " + f.getName() + ". ");

			for(Like l : f.getLikes()) {
				System.out.print("He likes " +
					l.getValue() + ". ");
			}

			System.out.print("\n");
		}
	}
}
```

Eseguendo il task all'interno dello script visto in precedenza, si ottiene il seguente output:

```
Andrea is friend with Paolo. He likes soccer.
Andrea is friend with Marco. He likes Beatles. He likes basketball.
```

Diamo adesso uno sguardo al ciclo di vita di un task, costituito dai seguenti passi:

1. Dopo il parsing dello script ANT, il task viene istanziato e al suo interno viene aggiornato il riferimento al progetto a cui appartiene
2. Il task viene aggiornato con il target a cui appartiene
3. Viene invocato il metodo *init()*
4. Tutti gli elementi innestati vengono creati invocando gli opportuni metodi *createXXX()*, definiti dall'utente

5. Tutti gli attributi del task sono valorizzati invocando gli opportuni *setXXX(),* definiti dall'utente
6. Viene invocato il metodo *execute()*.

La cose importanti da notare sono le seguenti:

- quando il metodo *execute* è invocato, tutti gli elementi innestati sono stati ricorsivamente creati, e gli attributi sono del task sono stati valorizzati
- il task ottiene dei riferimenti all'oggetto *project* a cui appartiene e al *target* nel quale è inserito. Questa è una caratteristica molto importante, in quanto è possibile reperire l'oggetto *project* tramite il metodo *getProject* della classe *Task*.
- L'oggetto *Project* è l'albero astratto del file di script, creato da ANT dopo il parsing dello stesso. Da esso è possibile accedere a tutti gli elementi presenti e non solo: è possibile inserirne di nuovi, invocare dei target, reperire e/o impostare i valori delle proprietà, ecc.. Questa caratteristica permette di creare degli script incredibilmente dinamici che possono modificare il loro comportamento e la loro struttura a runtime.

5.3.5. Sintassi dei file .properties

In un file .properties tipicamente ogni riga contiene una proprietà, intesa come coppia (nome, valore). La sintassi più utilizzata è la seguente:

```
# Questo è un commento, ignorato dal parser
user.name=Andrea Bucaletti
home.dir=C\:\\Users\\Andrea
```

Gli eventuali spazi bianchi prima e dopo il segno "=" vengono eliminati dal parser:

```
# Equivalente a user.name=Andrea.Bucaletti
user.name = Andrea Bucaletti
```

Il carattere backslash ("\") è utilizzato per effettuare l'escape del carattere successivo. In genere ogni proprietà è specificata in una linea, tuttavia, è possibile utilizzare il carattere backslash per definire una proprietà su più linee. Gli spazi bianchi all'inizio della linea sono ignorati:

```
# Equivalente a user.name=Andrea Bucaletti.
user.name=Andrea \
            Bucaletti
```

5.4. Android SDK

Android SDK è il toolkit per lo sviluppo di applicazioni messo a disposizione da Android. Può essere scaricato gratuitamente dal sito ufficiale ed arriva completo di tutti i tool necessari allo sviluppo di applicazioni. Attualmente può essere scaricato in due versioni:

- SDK
- Android Studio

La differenza principale è che la seconda è corredata di una IDE (Android Studio) basata su IntelliJ, che offre avanzati editor di codice e build dei progetti completamente automatici.

Pare inoltre che, in Android Studio il build dei progetti venga effettuato tramite Gradle, e non utilizzando Apache ANT.

In questa tesi si utilizza la prima versione, in quanto contiene tutto quello che serve per gestire e fare il build di progetti Android, e perché attualmente gli sviluppatori di tuProlog utilizzano Eclipse come IDE.

5.4.1. Struttura di un progetto Android

Un progetto Android è definito dalla seguente struttura di cartelle, che può essere generata tramite i tool inclusi nell'SDK:

/bin/ - In questa cartella andranno tutti i file necessari al compilatore per create l'archivio APK finale, incluse le classi e le librerie compilate, e le risorse necessarie all'applicazione

/libs/ - Le librerie JAR esterne vanno messe in questa cartella. Verranno automaticamente aggiunte al classpath

/res/ - questa cartella contiene tutte le risorse necessarie all'applicazione, incluse immagini e i layout.

/src/ - in questa cartella vengono inseriti i sorgenti veri e propri dell'applicazione

/AndroidManifest.xml – tutte le applicazioni Android devono includere questo file nella cartella di root del progetto. Il file contiene tutte le informazioni necessarie al sistema Android per lanciare l'applicazione (ad esempio, la versione di Android, le autorizzazioni richieste, ecc) e definisce la natura di tutti i componenti dell'applicazione

/local.properties – proprietà locali dello script di build

/project.properties – contiene delle proprietà necessarie al processo di build

/build.xml – lo script ANT che permette di fare il build dell'applicazione.

Come detto in precedenza, tutta questa struttura può essere automaticamente generata dai tool inclusi nell'SDK. In particolare riferirsi alle sezioni 5.4.2.1 e 5.4.2.2.

5.4.2. Gestione dei progetti Android

In questa sezione viene illustrato come gestire i progetti Android da riga di

comando mediante l'apposito file batch, fornito con l'SDK.

Per poter utilizzare questi comandi è necessario includere nella variabile d'ambiente *PATH* la cartella "<Cartella-Android-SDK>/tools"

5.4.2.1. Creazione di un progetto

Per create un nuovo progetto Android, aprire una shell e lanciare il seguente comando:

```
android create project
      --name HelloWorld --path "C:\HelloWorld"
      --target android-22 --package com.android.hello
      --activity MainActivity
```

Come si può facilmente intuire, si imposta un nome per il progetto (*--name*), il percorso della cartella di root (*--path*), il target Android[1] (*--target*), il package di default dei sorgenti Java (*--package*) e l'Activity[2] che sarà eseguita quando si lancia l'applicazione.

Questo comando crea tutta la struttura di file e cartelle citata nella sezione 5.4.1. In generale, tutti i file di configurazione non dovrebbero essere modificati, in quanto all'atto dell'aggiornamento del progetto (vedi 5.4.2.2), questi verranno sovrascritti.

Se si vuole aggiungere delle proprietà è possibile creare manualmente il file *ant.properties*, che, se esiste, è automaticamente caricato dal file di *build.xml*.

5.4.2.2. Aggiornamento di un progetto esistente

Se invece è necessario aggiornare un progetto Android esistente, è sufficiente lanciare il comando:

```
android update project --name HelloWorld2 --path "C:\HelloWorld"
                       --target android-21
```

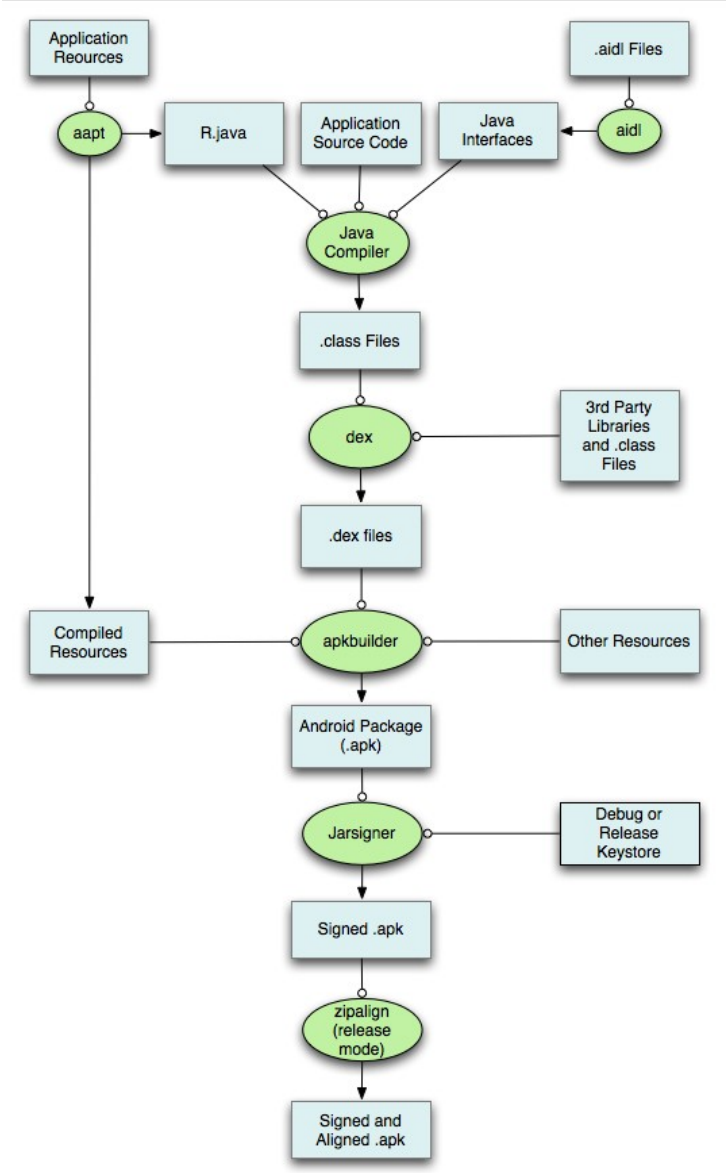
Questo comando aggiorna il progetto da una versione di SDK precedente, oppure crea un nuovo progetto da codice esistente.

Anche questo comando genera la struttura di file e directory tipica di Android. Inoltre è anche possibile ridefinire i parametri, come ad esempio, il nome del progetto e il suo target Android.

1 Il target permette di specificare quale versione delle API Android utilizzare [10]
2 "Componente applicativo che fornisce un'interfaccia con la quale l'utente può interagire per fare una determinata azione, come ad esempio scattare una foto, digitare un numero di telefono, ecc" [10]

5.4.3. *Processo di sviluppo*

Il processo di sviluppo di applicazioni Android è abbastanza complesso, e può essere schematizzato come segue.



Non è obiettivo di questa tesi scendere nei dettagli di questo processo, comunque i passi più importanti sono i seguenti:

1. I file sorgenti .java vengono compilati tramite Javac
2. Viene eseguito il tool *dex*, che converte i file .class e le librerie (incluse nella cartella *libs*) in bytecode Dalvik[1]
3. A questo punto il tool *apkbuilder* crea l'installer dell'applicazione (file .apk), inserendo i file .dex e le altre risorse necessarie
4. Il file APK deve poi essere firmato, utilizzando la chiave di debug o la chiave di release
5. Infine, se è stata usata la chiave di release, il file passa attraverso il tool

1 Dalvik è la macchina virtuale che esegue le applicazioni Android

zipalign, che diminuisce l'utilizzo di memoria da parte dell'applicazione

Una cosa molto importante da dire riguarda il passo 2: attualmente la versione 5.1 di Android, che è la più recente, **non supporta** file .class (o archivi .jar che li contengono) compilati con JDK8. Quindi è necessario compilare le librerie necessarie all'applicazione con al più JDK7.

Tutti i tool presenti nello schema (rappresentati dai blocchi verdi), sono inclusi nell'SDK di Android, quindi volendo è possibile scrivere un script ANT che esegua esattamente la sequenza di azioni indicata. Tuttavia questo non conviene 1) perché i processi di creazione e aggiornamento dei progetti Android creano automaticamente il file *build.xml* e 2) perché è possibile che in futuro il processo di sviluppo cambi.

Il file *build.xml* ci mette a disposizione due diversi target per generare l'archivio APK:

1. *debug* – crea l'archivio e lo firma con la chiave di debug
2. *release* – crea l'archivio e lo firma con il Keystore fornito

Per eseguire la prima modalità è sufficiente posizionarsi nella cartella root del progetto con una shell e lanciare ANT con target "debug":

```
ant debug
```

La seconda modalità, invece, richiede prima di generare una chiave con cui firmare l'archivio APK. Per effettuare questa operazione si utilizza il *keytool* fornito con la JDK:

```
keytool -genkey -keystore "C:\mykeystore.keystore" \
      -alias com.android.mykeystore -keyalg RSA
      -keysize 2048 -validity 10000
```

Questo comando genera una chiave nel file *C:\mykeystore.keystore* con alias "com.android.mykeystore" valida per 10000 giorni.

Il tool chiede all'utente di inserire una password per la chiave, e altri dati (nome e cognome, organizzazione, ecc) che vanno a costituire il DN (Distinguished Name) dell'utente.

Una volta creata la chiave è necessario che il sistema di build la trovi tra le sue proprietà, quindi si crea il file *ant.properties* (se non è già presente) nella cartella di root del progetto e si aggiungono le seguenti proprietà:

key.store=C\:\\mykeystore.keystore

key.alias=com.android.mykeystore

A questo punto basta aprire una shell nella root del progetto e lanciare ANT con target "release":

```
ant release
```

Se tutto va a buon fine, alla fine del processo viene mostrato un prompt dove

inserire la password del keystore creato in precedenza. Il file APK generato è pronto per essere direttamente installato su SmartPhone/Tablet oppure essere inserito nell'AppStore di Android.

5.4.4. Installazione su AVD

Per installare l'applicazione su AVD[1], lo script *build.xml* ci mette a disposizione i seguenti target:

- *debug install* – esegue il build in versione debug e installa l'applicazione su una AVD in esecuzione o un device collegato
- release install – esegue il build in modalità release e installa l'applicazione su una AVD in esecuzione o un device collegato
- *installd* – installa la versione di debug **precedentemente generata** su AVD in esecuzione o un device collegato
- *installr* – installa la versione release **precedentemente generata** su AVD in esecuzione o un device collegato

Come al solito, per eseguire uno di questi target, aprire una shell nella cartella di root del progetto e lanciare ANT:

```
ant debug install
ant installr
...
```

1 Android Virtual Device, necessario all'emulatore per Android [10]

6. Specifica JSR223: Scripting for the Java Platform

Oltre a supportare il processo di sviluppo di tuProlog per le piattaforme considerate, un altro aspetto rilevante è incrementare gli utilizzi "orizzontali". In questo senso appare interessante la specifica JSR-223[1], che definisce delle interfacce standard per l'implementazione di *script engine*, ossia componenti Java in grado di eseguire programmi in un determinato linguaggio di scripting.

La specifica fornisce un meccanismo di discovery che, a run-time, consente di richiedere uno script engine per uno specifico linguaggio, e se disponibile, viene fornita un'implementazione che l'utente può utilizzare tramite le interfacce standard definite dalla specifica.

Quindi se questa specifica fosse supportata, sarebbe possibile utilizzare tuProlog come script engine all'interno di applicazioni Java utilizzando le interfacce standard definite, nascondendone la reale l'implementazione, e quindi estendendo i possibili utilizzi.

6.1. Overview

Originariamente questa specifica era stata pensata per definire uno standard per la generazione di contenuti WEB. In quest'ottica è necessario definire un'insieme di interfacce che permettano l'esecuzione di script e di effettuare un'associazione (*binding*) tra gli oggetti dell'applicazione host e l'ambiente di esecuzione dello script. In particolare, uno script engine che sia compliant con la specifica deve obbligatoriamente implementare le due interfacce *ScriptEngine* e *ScriptEngineFactory.*

Un'istanza di *ScriptEngine* è un componente che mette a disposizione dei metodi per l'interpretazione e l'esecuzione di script per il linguaggio considerato ed espone i binding tra gli oggetti dell'applicazione host e l'ambiente di esecuzione dello script. In questo contesto, un *binding* è definito da una coppia (*String key, Object value*), dove *key* rappresenta di solito il nome della variabile di script associata all'oggetto *value* nell'applicazione host.

Ad ogni istanza di *ScriptEngine* è associato un *contesto locale*, costituito dal set di binding associato alla particolare istanza, e da degli stream a caratteri che devono essere utilizzati per gestire lo standard I/O dell'ambiente di esecuzione degli script.

ScriptEngineFactory è invece utilizzata dal meccanismo di discovery per creare istanze di *ScriptEngine* relativamente al linguaggio considerato, ed inoltre espone una serie di metadati che descrivono le caratteristiche specifiche dell'implementazione.

6.2. Analisi del problema

In questa sezione vengono analizzate le principali differenze tra le funzionalità messe a disposizione da tuProlog per l'esecuzione di programmi prolog, e le funzionalità che invece devono essere nelle interfacce definite dalla specifica JSR-223, in modo da verificare la fattibilità dell'implementazione.

6.2.1. Definizione di script

Considerando un tipico linguaggio di scripting, è naturale definire *script* un programma, inteso come sequenza di istruzioni che devono essere poi interpretate ed

1 Scripting for the Java Platform [15]

eseguite dallo script engine per lo specifico linguaggio.

Tuttavia, nel caso di tuProlog, è necessario specificare cosa si intende per *script:* un programma prolog è infatti costituito da un insieme di regole e fatti detto *teoria* e da un *goal* (o *query*) che l'utente pone al sistema. Si pone quindi la questione se considerare script la teoria, il goal, o l'unione di entrambi.

E' ragionevole pensare che l'utente che utilizza tuProlog si aspetti dei metodi per gestire la teoria e un metodo per porre al sistema un determinato goal. La specifica JSR-223 non fa questa distinzione e definisce il solo metodo:

```
Object eval(String script)
```

nell'interfaccia *ScriptEngine*, che ha la funzione di valutare uno script e restituirne il risultato. Nel caso di tuProlog, si fa l'assunzione che l'utente utilizzi il metodo *eval* per porre al sistema un determinato goal, mentre la teoria può essere gestita tramite un *binding riservato*[1] nel contesto locale.

6.2.2. Valutazione degli script

Gli script vengono valutati utilizzando il metodo *eval*, il cui tipo di ritorno è un generico *Object* che la specifica definisce in maniera molto vaga come "risultato dello script".

L'utente che esegue un programma prolog è tuttavia interessato al risultato, che comprende la veridicità o meno della query, i valori delle variabili unificate se presenti, e se il goal può ammettere soluzioni alternative o meno. Tutte queste informazioni non possono essere inserite in un generico *Object*. Una possibile soluzione è fare in modo che il metodo *eval*, una volta terminata la valutazione dello script, inserisca dei binding nel contesto locale con tutte le informazioni che descrivono la soluzione trovata.

Un altro aspetto da considerare è appunto il caso che il goal posto abbia soluzioni alternative: in questo caso tuProlog mette a disposizione il metodo *solve* che restituisce la prima soluzione trovata, e il metodo *solveNext* che consente di reperire le eventuali soluzioni alternative. E' necessario quindi implementare il metodo *eval* di *ScriptEngine* in modo tale da gestire entrambe i comportamenti.

6.2.3. Standard I/O

La specifica prevede che nel contesto locale di un'istanza di *ScriptEngine* siano definiti degli stream a caratteri per gestire lo standard I/O relativo all'ambiente di esecuzione degli script. Sono quindi previsti dei metodi mediante i quali l'utente può specificare delle istanze di *Reader* e *Writer* che devono essere utilizzati come standard input, output e error relativamente all'esecuzione degli script.

tuProlog è dotato di una libreria (IOLibrary) che mette a disposizione dei predicati per la gestione dell'I/O, la quale definisce standard input e output, utilizzando però degli stream binari.

Questo costituisce un problema, in quanto non è possibile redirigere lo standard

1 Un binding con chiave nella forma "javax.script.XXX" [15]

I/O della IOLibary[1] sugli stream specificati nel contesto locale di *ScriptEngine*. Una possibile soluzione è implementare delle classi di tipo *adapter* in grado di incapsulare stream a caratteri in stream binari.

6.3. Progetto e implementazione

L'analisi fatta nella sezione precedente mostra come l'implementazione della specifica JSR-223 per tuProlog sia fisicamente possibile sotto determinate condizioni:

- implementare il metodo *eval* di *ScriptContext* in modo da tale da includere le caratteristiche tipiche di un programma prolog
- eliminare l'incompatibilità nella modalità di gestione dell'I/O

6.3.1. Valutazione degli script

Per la corretta esecuzione di uno script prolog, è necessario che l'utente carichi la teoria, che viene gestita tramite un binding riservato nel contesto locale avente la seguente chiave:

```
public static final String THEORY = "javax.script.tuprolog.theory";
```

La specifica impone che lo script abbia accesso ai binding del contesto locale: in questo caso si sfrutta la OOLibrary[2] di tuProlog, utilizzando il metodo *register* di quest'ultima per inserire tutti gli oggetti che sono attualmente presenti nei binding del contesto locale. Questa operazione è computazionalmente onerosa, tuttavia necessaria per soddisfare la specifica:

```
for(Map.Entry<String, Object> keyPair: bindings.entrySet()) {
    try {
        ooLibrary.register(new Struct(keyPair.getKey()), keyPair.getValue());
    }
    catch(InvalidObjectIdException ex) {
        . . .
    }
}
```

E' necessario implementare il metodo *eval* in modo tale da permettere all'utente di reperire tutte le soluzioni possibili relative ad un determinato goal. Una possibile soluzione è che, nel caso in cui il metodo *eval* venga invocato più volte per lo stesso script (confrontato a livello di stringa), di utilizzare il metodo *solve* di tuProlog per la prima invocazione e il metodo *solveNext* per quelle successive:

```
if(!script.equals(previousScript))
    useSolveNext = false;
```

1 Libreria di predicati prolog per gestire le operazioni di I/O [9]
2 Libreria ad oggetti di tuProlog [9]

```
if(useSolveNext)
    solveInfo = prolog.solveNext();
else
    solveInfo = prolog.solve(script);
```

I metodi *solve* e *solveNext* di tuProlog restituiscono un oggetto *SolveInfo* che contiene tutte le informazioni relative alla soluzione trovata, in particolare:

- se la query ha avuto esito positivo o meno (metodo *SolveInfo.isSuccess)*
- se l'esecuzione del programma è stata interrotta (metodo *SolveInfo.isHalted*)
- se sono ammesse soluzioni alternative (metodo *SolveInfo.hasOpenAlternatives*)
- le variabili unificate e il relative valore (metodo *SolveInfo*.*getBindingVars*)

E' necessario che l'utente abbia modo di accedere a queste informazioni, ricordando che non è possibile utilizzare il valore di ritorno del metodo *eval* per restituire un oggetto di tipo *SolveInfo*: questo infatti fa parte dell'implementazione di tuProlog, a cui l'utente non ha accesso.

Una possibile soluzione consiste nell'inserire dei binding riservati del contesto locale tutte le informazione contenute nell'oggetto *SolveInfo:*

```
public static final String IS_SUCCESS =  "javax.script.tuprolog.isSuccess";
public static final String IS_HALTED = "javax.script.tuprolog.isHalted";
public static final String HAS_OPEN_ALTERNATIVES =
"javax.script.tuprolog.hasOpenAlternatives";

. . .

bindings.put(IS_SUCCESS, solveInfo.isSuccess());
bindings.put(IS_HALTED, solveInfo.isHalted());
bindings.put(HAS_OPEN_ALTERNATIVES, solveInfo.hasOpenAlternatives());
```

Le variabili unificate vengono inserite del contesto locale come coppie (nome variabile, valore unificato):

```
if(solveInfo.isSuccess()) {
    solveVars = solveInfo.getBindingVars();
    for(Var v : solveVars)
        bindings.put(v.getName(), v.getTerm().toString());
}
```

6.3.2. Gestione dello standard I/O

In fase di analisi si è evidenziato come la specifica JSR-223 e tuProlog gestiscano lo standard I/O in maniera differente e apparentemente non compatibile.

Per quanto riguarda standard output e standard error, tuProlog mette a disposizione delle interfacce di tipo listener che definiscono i seguenti metodi:

```
public void onException(ExceptionEvent e)
public void onOutput(OutputEvent e)
```

Questi vengono invocati da tuProlog in caso di scrittura sullo standard output (*onOutput*) e in caso di errore (*onException*). Sfruttando questi metodi è possibile evitare i problemi di incompatibilità evidenziati in fase di analisi:

```
public void onException(ExceptionEvent e) {
	. . .
	stdErr.write(e.getMsg());
	. . .
}

@Override
public void onOutput(OutputEvent e) {
	. . .
	stdOut.write(e.getMsg());
	. . .
}
```

dove *stdOut* e *stdErr* sono i due oggetti classe *Writer* che rappresentano, rispettivamente, standard output e standard error associati al contesto locale.

Un discorso a parte è necessario per quanto riguarda lo standard input: in questo caso è obbligatorio redirigere lo stream (a caratteri) specificato dal contesto locale su quello (binario) definito nella *IOLibrary* di tuProlog. Una possibile soluzione è quella di implementare un classe di tipo *adapter* che permetta di incapsulare un *Reader* all'interno di un *InputStream*:

```
public class InputStreamAdapter extends InputStream {

	private Reader reader;

	public InputStreamAdapter(Reader rd) {
		this.reader = rd;
	}

	@Override
	public int read() throws IOException {
		int x = reader.read();

		if(x == -1) return -1;
		else return x & 0xFF;
	}

	. . .
```

```
}
```

La classe *InputStreamAdapter* è un estensione di *InputStream* che permette, con delle limitazioni, di effettuare operazioni di lettura di byte da un *Reader.*

Normalmente questo non è possibile, poiché il metodo *read* della classe *Reader* restituisce un dato di tipo *char*, che in Java ha dimensione di 2 byte. Tuttavia, se suppone che il carattere letto sia nel range ASCII, questo è contenuto nel byte meno pesante del *char* letto.

E' possibile utilizzare la classe *InputStreamAdapter* per redirigere quindi lo standard input della *IOLibary* di tuProlog sullo stream a caratteri definito nel contesto locale:

```
ioLibrary.setStandardInput(new InputStreamAdapter(localContext.getReader()));
```

7. Discussione

Il progetto illustrato nel capitolo 4 soddisfa tutti i requisiti definiti in fase di analisi, e quindi anche gli obiettivi che ci si era posti inizialmente:

- Il nuovo processo di sviluppo è completamente *locale* e non dipende dallo specifico repository utilizzato, il che ha reso possibile il passaggio da GoogleCode a BitBucket.
- L'architettura definita separa logicamente tutte le piattaforme considerate, quindi in questo senso è possibile *estendere* il processo in modo da integrare nuove piattaforme senza apportare modifiche a quelle esistenti
- La *riconfigurabilità* del processo e della sua architettura è garantita in parte dall'interfaccia grafica, che permette di gestire in modo semplice e intuitivo tutti i parametri definiti dai file di configurazione del processo, e in parte dalla modalità di implementazione che è stata adottata per l'interfaccia grafica stessa: in particolare, la sintassi XML utilizzata permette di riconfigurare l'architettura del progetto per aderire a cambiamenti anche importanti, come l'introduzione di nuove piattaforme e la modifica dell'architettura di quelle esistenti.

7.1. Efficienza e flessibilità

In fase di analisi, oltre a considerare i requisiti di base per il nuovo processo, sono stati messi in luce degli aspetti architetturali e implementativi che evidenziavano come il precedente processo fosse, in molti casi di utilizzo reale, poco flessibile e poco efficiente:

- La mancanza di efficienza ha influito molto sulla definizione della nuova architettura: in particolare ci si riferisce al processo di sviluppo per Java, che precedentemente costringeva l'utente a seguire una catena ben definita di azioni anche non necessarie al deployment dell'applicazione. Il nuovo processo in questo senso, permette di generare l'applicazione pronta per la distribuzione eseguendo solo le azioni necessarie, ossia *Build* e *Release*.
- La mancanza di flessibilità ha invece determinato l'eliminazione del sistema di dipendenze in senso ANT che caratterizzavano il precedente processo, e di conseguenza ha portato alla necessità di realizzare un'interfaccia grafica per gestire la maggiore complessità del nuovo processo e garantire la consistenza che prima era gestita implicitamente da ANT stesso attraverso il sistema di dipendenze.

7.2. Consistenza

In fase di progettazione si è mostrato un caso particolare che evidenzia come il nuovo processo non assicuri una consistenza totale: in particolare se l'utente interviene manualmente su dei file, generati automaticamente dal processo, che costituiscono l'input di un particolare target e successivamente esegue tale target, il sistema genera un errore dovuto alla mancanza dei file.

Questo problema poteva essere facilmente risolto forzando l'esecuzione di tutta la catena di dipendenze del target in questione, e quindi di fatto facendo "un passo indietro" in termini di flessibilità. Si è tuttavia deciso di privilegiare questo ultimo aspetto in quanto, in molti casi di utilizzo pratico, porta ad un deciso miglioramento

dei tempi d'esecuzione:

- Relativamente al processo di sviluppo Java, il nuovo processo permette l'esecuzione della sequenza di target *Build → Run Tests*, sotto l'ipotesi che i file di test siano stati compilati in precedenza. Questo scenario è reale e si verifica quando lo sviluppatore apporta delle modiche alle funzionalità core di tuProlog e vuole poi verificarne il corretto funzionamento.
- Relativamente al processo .NET, il target più utilizzato è *Build*, che compila il solo codice platform-dependant, mentre i target *Build Core* e *IKVM,* da cui *Build* dipende logicamente, vengono eseguiti solo a fronte di modifiche della versione Java di tuProlog. In questo contesto appare quindi necessario permettere l'esecuzione stand-alone del target *Build*
- Relativamente al processo Android, lo sviluppatore lavora per la maggior parte del tempo sul codice platform-dependant (ossia sull'interfaccia grafica) e per verificarne la correttezza utilizza la sequenza di target *Debug APK → Install*. Il target *Build Core*, invece viene utilizzato solo a fronte di cambiamenti delle funzionalità core di tuProlog.

7.3. Versioni di JDK per Android e .NET

Il precedente processo era pensato per utilizzare soltanto JDK7 per compilare i sorgenti. Non erano quindi noti problemi di compatibilità che si potevano verificare per il deplyoment delle applicazioni Android e .NET, che sono emersi nel momento in cui si è passati a JDK8.

La soluzione adottata in questa tesi risolve questo problema e copre anche tutti i possibili scenari futuri, permettendo, nel caso limite, di utilizzare una specifica versione di JDK per piattaforma.

Tuttavia è necessario precisare che, se nel core di tuProlog si utilizza, a livello di codice, dei costrutti introdotti con l'ultima versione di JDK disponibile, il codice non è compilabile con versioni di JDK precedenti.

Ad esempio, con JDK8 sono state introdotte le *lamba-expression* ed è in fase di sviluppo una libreria di predicati prolog per integrare questo costrutto. Si ricorda tuttavia che, allo stato attuale, non è possibile integrare tale funzionalità nell'applicazione Android, che deve invece essere compilata con JDK7 o inferiore.

7.4. Considerazioni finali sull'interfaccia grafica presentata

Come specificato in fase di introduzione, il precedente processo di sviluppo includeva solo una documentazione parziale e quindi per comprenderne l'architettura e le possibilità è stato necessario analizzarne anche l'implementazione.

Anche se la documentazione fosse stata completa, l'utente sarebbe comunque tenuto a conoscere l'architettura del processo per poterne sfruttare tutte le funzionalità al meglio. In questo senso, l'interfaccia grafica introdotta elimina parzialmente questa necessità, in quanto l'architettura è presentata in maniera semplificata mettendone in evidenza gli aspetti fondamentali, ossia le piattaforme, i target e il flusso delle operazioni.

Un altro aspetto da considerare sono i vincoli che un processo di sviluppo deve rispettare per poter integrare l'interfaccia: in particolare, in fase di progettazione si fa unicamente l'assunzione che il processo di sviluppo sia realizzato tramite Apache ANT. Tutti i meccanismi dell'interfaccia sono infatti basati sul concetto di target (in senso

ANT) e sui file di configurazione *.properties*, il formato utilizzato da ANT e Java per descrivere i parametri del processo.

Non si fa invece nessuna assunzione per quanto riguarda le piattaforme supportate dal processo e gli strumenti utilizzati da quest'ultimo, il che rende l'interfaccia riutilizzabile per altre applicazioni il cui processo di sviluppo è implementato tramite ANT. Infatti, supponendo di essere in possesso di tale implementazione, l'interfaccia può essere integrata all'interno dello script mediante la sintassi XML descritta in fase di progettazione.

Il vincolo di utilizzare ANT per il processo di sviluppo può sembrare una restrizione forte, in quanto esistono molti altri tool di build automation e con caratteristiche diverse. Tuttavia, in molti casi questi tool sono intercambiabili, nel senso che in termini di espressività e di possibilità, offrono le funzionalità necessarie a qualsiasi processo di sviluppo, che consistono in un set di azioni (operazioni sul file system, compilazione, esecuzione di programmi esterni, ecc) estendibile e un linguaggio per definire l'architettura del processo e il flusso delle operazioni.

Si può quindi concludere che l'interfaccia grafica realizzata, relativamente ai processi implementati tramite Apache ANT, costituisce un *approccio generale* per la configurazione e l'utilizzo dei processi di sviluppo per applicazioni multi-piattaforma e multi-linguaggio.

8. Conclusioni

L'obiettivo che ci si era proposti era il refactoring dell'attuale processo di sviluppo di tuProlog, in modo tale da renderlo locale, estendibile e facilmente riconfigurabile.

In fase di analisi sono stati messi in luce i problemi architetturali e implementativi del precedente processo, quali la frammentazione fisica e logica che lo rendevano difficilmente estendibile e configurabile. Si è inoltre mostrato come, in molti casi di utilizzo pratico, il processo fosse poco flessibile ed poco efficiente.

La soluzione proposta ha definito una nuova architettura per il nuovo processo che elimina la dipendenza da SVN, rendendo quindi il processo locale. Inoltre, la nuova architettura ha separato logicamente i processi di sviluppo per le singole piattaforme in modo da risultare facilmente estendibile.

I target e le relative dipendenze sono stati riorganizzati in modo tale da offrire maggiore flessibilità all'utente e introdurre nuove funzionalità che hanno reso il processo completamente automatico e indipendente da tool obsoleti, conservando comunque le feature che questi mettono a disposizione.

L'introduzione dell'interfaccia grafica ha reso il nuovo processo ancora più flessibile ed efficiente, permettendo all'utente di escludere delle azioni non necessarie e di minimizzare il tempo di configurazione dei tool utilizzati dal processo.

La maggiore complessità del nuovo processo processo rispetto al suo predecessore è compensata dalla visione di insieme e dalla praticità di utilizzo che l'interfaccia fornisce, ed inoltre la generalità dei vincoli adottati in fase di progettazione permette l'utilizzo dell'interfaccia per altri processi di sviluppo che utilizzano Apache ANT, e quindi in questo contesto costituisce un approccio generale al problema.

Tuttavia, in alcuni casi di utilizzo improprio si paga un prezzo in termini di consistenza che si ritiene accettabile a fronte di tutti i vantaggi introdotti.

Riferimenti

[1] APICe, tuProlog, 2015, http://apice.unibo.it/xwiki/bin/view/Tuprolog/

[2] The Apache Software Foundation, Apache Subversion Documentation, 2011, https://subversion.apache.org/docs/

[3] The Apache Software Foundation, Apache ANT Website, 2015, http://ant.apache.org/

[4] The Eclipse Foundation, Eclipse Website, 2015, https://eclipse.org/

[5] Enrico Denti, GoogleCode - tuProlog Project Wiki, 2015, https://code.google.com/p/tuprolog/w/list

[6] Jeroen Frijter, IKVM User's Guide, 2014, http://www.ikvm.net/

[7] Stephane Bailliez e altri, Apache Ant User Manual, 2015, https://ant.apache.org/manual/

[8] Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato, Version Control with Subversion, 2004, http://svnbook.red-bean.com/en/1.0/

[9] Enrico Denti, tuProlog Manual, 2015, https://bitbucket.org/tuprologteam/tuprolog/downloads

[10] Google Inc. , Android for Developers, 2015, http://developer.android.com/index.html

[11] Scott Chacon, Ben Straub , Pro Git, 2014, https://git-scm.com/book/en/v2

[12] Oracle, JavaSE Website, 2015, http://www.oracle.com/technetwork/java/javase/overview/index.html

[13] Oracle, java, 2014, http://docs.oracle.com/javase/7/docs/technotes/tools/windows/java.html

[14] Oracle , javac - Java programming language compiler, 2014, http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html

[15] Mike Grogan, Scripting for the Java™ Platform, 2006, https://www.jcp.org/en/jsr/detail?id=223