ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in
Fondamenti di Informatica L-B

Estensione dell'interfaccia grafica di tuProlog per Android

CANDIDATO
Mirco Mastrovito

RELATORE:
Chiar.mo Prof. Enrico Denti

Anno Accademico 2012/13

Sessione II

Al termine di questo lungo, e a tratti impervio, percorso di studio
sento il bisogno di ringraziare la mia Famiglia che è sempre stata al mio fianco
spalleggiandomi e spronandomi a dare il meglio.
Da qualche anno ha preso a far parte di questa Famiglia anche Sara,
trasformando questi sforzi in un obiettivo di vita comune.

Un caloroso ringraziamento va anche a tutte le splendide persone
che ho conosciuto lungo il mio percorso di studente e di uomo:
gli amici di gioventù, i compagni di università, i colleghi e tutti i folli
che fanno tuttora parte della mia vita e con i quali condivido passioni ed obiettivi.
Grazie a voi non ho mollato e non lo farò mai!

Sommario

Introduzione 2
Capitolo 1 - *tuProlog* 4
1.1 Definizione predicati 4
1.2 Definizione librerie 5
1.3 Programmazione multi-paradigma 6
1.4 Analogie e differenze tra le versioni 7
Capitolo 2 8
2.1 La piattaforma Android 8
2.2 Applicazioni Android 9
2.3 Analisi delle versioni per Java e Android 11
2.3.1 tuProlog su piattaforma Java 12
2.3.2 tuProlog su piattaforma Android 14
2.4 Librerie 15
Capitolo 3 – Analisi del problema 16
3.1 Analisi dei requisiti 17
Capitolo 4 – Progetto della soluzione 19
4.1 Modifiche alla IOLibrary 19
4.1.1 Inserimento strato di separazione 19
4.1.2 Metodi per il controllo dell'esecuzione 20
4.1.3 Aggiornamento predicati see e seen 20
4.2 Modifiche alla ISOIOLibrary 22
4.3 La classe Stream 22
4.4 Modifiche alla GUI Java 23
4.5 Modifiche alla CUIConsole Android 25
Capitolo 5 – Collaudo 27
5.1 Testing delle librerie 27
5.2 Testing dell'interfaccia grafica Java 27
5.3 Testing dell'interfaccia grafica Android 29
5.3.1 Dispositivi virtuali 29
5.3.2 Dispositivi reali 30
Conclusioni 32
Appendice A 42
Bibliografia 43

Introduzione

L'obiettivo di questa tesi è quello di estendere l'interfaccia grafica di tuProlog nella versione per Android, il più diffuso sistema operativo per tablet e smartphone.

TuProlog è un' interprete Prolog interamente scritto in java, leggero e open–source. L'applicazione è disponibile sotto forma di archivio JAR eseguibile e può essere utilizzato tramite un'interfaccia a riga di comando, nella versione Java, o per mezzo di interfacce grafiche negli ambienti Java, .NET e Android.

La versione per Android supporta pienamente Java e la maggior parte delle librerie dell'applicativo per JVM. Lo sviluppo di applicazioni per dispositivi mobile, però, limita lo sviluppatore in termini di complessità delle elaborazioni effettuabili dal programma e comprensibilità dell'interfaccia grafica; per questi e altri motivi, la struttura dell'applicativo in versione Android, fatta eccezione per il core Prolog, è diversa dalle versioni per altri ambienti.

L'applicazione, giunta ora alla versione 2.7.2, manca della possibilità di input da console in tutte le versioni ad interfaccia grafica.

Scopo di questa tesi è quindi integrare tale funzionalità, inserendola all'interno del contesto applicativo senza modificare il normale flusso delle operazioni, intervenendo in modo mirato, il meno invasivo possibile e garantendo l'espandibilità della modifica ad estensioni future.

Nel primo capitolo verrà presentata la struttura di tuProlog analizzando la composizione e il funzionamento nei diversi ambienti di esecuzione.

Nel secondo capitolo verranno brevemente presentati il sistema operativo Android e le caratteristiche salienti delle applicazioni eseguibili su di esso.

Nel terzo capitolo si effettuerà poi l'analisi della struttura del sistema software e dei requisiti richiesti per individuare le modalità e strategie di intervento ottimali allo sviluppo della soluzione.

La progettazione della soluzione sarà la tematica principale del quarto capitolo, nel quale si discuteranno le modifiche da introdurre e gli effetti che queste avranno sul sistema.

Nel quinto capitolo verranno presentati i risultati dei test effettuati e le modalità di collaudo del sistema ottenuto.

In conclusione verranno analizzati: il risultato ottenuto, le motivazioni delle scelte realizzative effettuate e possibili sviluppi futuri.

In ultimo, viene riportato in Appendice A il dettaglio delle modifiche alla libreria di IO e le classi introdotte per raggiungere l'obiettivo preposto.

Capitolo 1 *tuProlog*

tuProlog è un sistema Prolog leggero, open-source per applicazioni e infrastrutture distribuite, intenzionalmente progettato intorno ad un nucleo minimo contenente solo le proprietà più essenziali di un motore Prolog. Quest'ultimo può essere configurato staticamente e dinamicamente caricando e scaricando librerie di predicati.

Le responsabilità vengono suddivise e gestite singolarmente da vari manager, ognuno dei quali controlla parte dell'esecuzione del motore Prolog. Questa modularità, unita alla leggerezza del sistema, lo rendono strumento fruibile su molteplici piattaforme .

1.1 Definizione Predicati

Il sistema tuProlog prevede tre tipi diversi di predicati organizzati in altrettante categorie:

- **predicati built-in** – Rappresentano predicati incorporati in qualsiasi motore tuProlog. Qualunque modifica effettuata al motore prima o durante l'esecuzione, non ha alcun effetto su questa categoria di predicati.

- **predicati library** – Sono i predicati che è possibile caricare all'interno del motore tuProlog per mezzo di una libreria. Il set di predicati di libreria di un motore tuProlog non è fisso (in quanto le librerie possono essere caricate, scaricate sia staticamente sia dinamicamente) ma varia da motore a motore e nello stesso motore nel tempo. I predicati di libreria possono essere sovrascritti da predicati definiti a livello di teoria Prolog (descritti in seguito). Entrambi i tipi di predicati non posso essere individualmente rimossi dal motore, ma risulta necessario rimuovere l'intera libreria nella quale sono contenuti.

- **predicati theory** – I predicati caricati in un motore tuProlog per mezzo di una teoria tuProlog sono chiamati predicati teoria. L'insieme dei predicati teoria di ogni motore tuProlog non è fisso e può variare da motore a motore o nello stesso motore in tempi diversi, in quanto le teorie possono essere caricate e scaricate anch'esse dinamicamente.

Seppure i predicati di libreria e quelli di teoria possano sembrare simili, sono gestiti in modi differenti e risultano concettualmente diversi. Infatti, mentre l'uso dei predicati di teoria è quello di rappresentare assiomaticamente la conoscenza del dominio al momento in cui viene eseguita la query, i predicati libreria rappresentano più la conoscenza "stabile", incapsulata una volta per tutte all'interno di un contenitore, le librerie appunto.

1.2 Definizione Librerie

Le librerie sono il mezzo attraverso il quale tuProlog realizza la sua caratteristica di configurabilità dinamica. Il motore è un nucleo minimale e leggero che include solo pochi predicati, definiti staticamente al suo interno. Ogni libreria fornisce al motore tuProlog un insieme di predicati, funtori, operatori nuovi (implementati nel linguaggio prescelto per la piattaforma di esercizio) e la relativa teoria. Il LibraryManager gestisce il caricamento dei predicati da libreria, anche a runtime.

Le seguenti librerie sono caricate automaticamente nel motore dal costruttore di default al momento dell'avvio del programma e forniscono allo stesso le caratteristiche e le funzionalità di base:

- *BasicLibrary*: ("*alice.tuprolog.lib.BasicLibrary*") fornisce i predicati, i funtori e gli operatori di base di Prolog, come ad esempio i predicati necessari per la creazione, unificazione e decomposizione di termini Prolog, per il confronto di termini e di espressioni, per la creazione e distruzione dinamica di clausole, per impostare, aggiungere e leggere teorie, funtori per la valutazione di espressioni;

- *IOLibrary*: ("*alice.tuprolog.lib.IOLibrary*") fornisce alcuni predicati di base per l'I/O, con l'obiettivo, ad esempio, di leggere un termine Prolog da uno stream di input o di scriverne uno su uno stream di output;

- *ISOLibrary*: ("*alice.tuprolog.lib.ISOLibrary*") fornisce predicati e funtori che fanno parte della sezione built-in nello standard ISO e che non sono forniti dalle precedenti librerie, come ad esempio predicati per il controllo dei tipi, per la gestione e l'elaborazione degli atomi, funtori per svolgere operazioni matematiche;

- *ISOIOLibrary*: ("*alice.tuprolog.lib.ISOIOLibrary*") estende la sopracitata IOLibrary con l'aggiunta di predicati di IO in standard ISO;

- *JavaLibrary*: ("*alice.tuprolog.lib.JavaLibrary*") fornisce predicati e funtori che consentono di creare, accedere ed utilizzare le risorse Java (come classi ed oggetti) come ad esempio predicati che consentono di istanziare oggetti di una certa classe e di invocare metodi su di essi, e predicati per la gestione di array;

- *DCGLibrary*: ("*alice.tuprolog.lib.DCGLibrary*") fornisce il supporto per la "Definite Clause Grammar", un'estensione delle grammatiche "context free" usata per descrivere linguaggi naturali e formali. Questa libreria non viene caricata automaticamente quando viene creato un motore tuProlog.

1.3 Programmazione multi-paradigma

tuProlog supporta nativamente la programmazione multi-paradigma, fornendo un modello di integrazione pulito, senza soluzione di continuità tra Prolog e linguaggi orientati agli oggetti tradizionali, cioè Java, per la versione tuProlog Java e linguaggi basati su .NET (C#, F# ..), per tuProlog .NET. E' anche facilmente implementabile, richiede solamente la presenza di una macchina virtuale Java/CLR e un'invocazione su un singolo file di archivio autonomo.

Le versioni attuali :

- Java SE : 2.7.2
- Android : 2.7.2
- . NET : 2.7.2
- Eclipse : 2.7.2.1

1.4 *Analogie e differenze tra le versioni*

Tutte le versioni tuProlog sono basate sullo stesso JAR e pertanto hanno in comune il core e le librerie fondamentali; differiscono, invece, nell'interfaccia grafica e, naturalmente, nella struttura del progetto che riflette le specificità delle singole piattaforme.

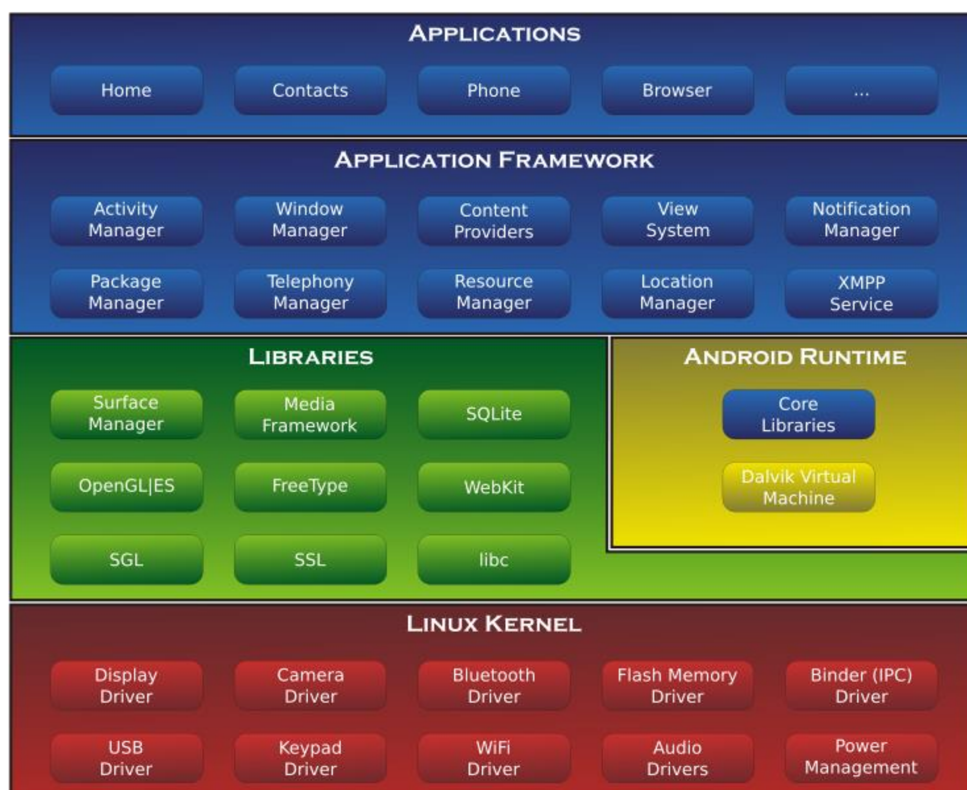
La GUI di tuProlog per Java, infatti, segue il tradizionale approccio basato su librerie Swing e ne adotta la struttura tipica, mentre nella versione Android, non soltanto si fa riferimento a componenti tipici delle librerie grafiche di Android, ma la struttura è organizzata in Activity – una per ogni schermata proposta all'utente dell'applicazione – con conseguente notevole diversità organizzativa.

Tali diversità sono comunque incapsulate nelle classi che realizzano le GUI, che risultano quindi intercambiabili rispetto al core e alle librerie invarianti. Questa caratteristica, fondamentale per mantenere il rapporto multi-piattaforma manutenibile nel tempo, deve dunque essere garantita anche per il futuro e costituisce quindi il requisito sostanziale alla base di qualunque modifica o estensione.

2.1 La piattaforma Android

Android è un sistema operativo open-source per dispositivi mobile organizzato in una struttura software che comprende un sistema operativo di base (su kernel linux), i middleware per le comunicazioni e le applicazioni di base. Giunto alla versione "4.4.2 KitKat" è oggi il sistema per smartphone e tablet più diffuso al mondo, arrivando ad equipaggiare 900 milioni di dispositivi nel 2013 e proponendo quasi un milione di app solo sullo store ufficiale.

La struttura del sistema è costituita da livelli indipendenti:



- Il cuore del sistema è basato sul kernel di Linux (versione 2.6) che costituisce il livello di astrazione di tutto l'hardware sottostante che può includere wi-fi, bluetooth, GPS, fotocamera, touchscreen. Grazie all'astrazione dell'hardware i livelli soprastanti non si accorgono dei cambiamenti hardware, permettendo una programmazione ad alto livello omogenea ed una *user experience* indipendente dal device. L'affidabilità è più importante delle prestazioni in un dispositivo mobile che deve principalmente garantire

il servizio di telefonia. Gli utenti si aspettano quindi tale affidabilità, ma allo stesso tempo hanno bisogno di un dispositivo che possa garantire servizi più evoluti: Linux permette di raggiungere entrambi gli scopi.

- Al penultimo livello è possibile rintracciare i gestori e le applicazioni di base del sistema. Ci sono gestori per le risorse, per le applicazioni installate, per le telefonate, il file system e altro ancora.

- Al livello più alto risiedono le applicazioni utente. Le funzionalità base del sistema, come per esempio il telefono, non sono altro che applicazioni utente scritte in Java e che girano ognuna nella sua JVM. Da notare che non è la distribuzione Java ME quella che viene eseguita, ma una versione appositamente sviluppata per Dalvik: il codice Java infatti viene compilato e tradotto in *byte code*, dopodiché subisce una ulteriore trasformazione in un formato chiamato *dex file*.

2.2 Applicazioni Android

Le Applicazioni Android sono scritte in linguaggio Java e una volta compilate viene creato un file autoinstallante con estensione .apk.

La struttura caratteristica delle applicazioni Android è a componenti: ogni componente è un punto attraverso cui il sistema e l'utente possono comunicare con l'applicazione. I componenti si suddividono in varie categorie e non tutti sono direttamente visibili all'utente: pur dipendendo spesso gli uni dagli altri ai fini dell'applicazione, rimangono elementi ben distinti con metodi, variabili e cicli di vita propri.

Esistono quattro tipologie di componenti:

1. Activity: rappresenta una singola schermata dell'applicazione con una interfaccia utente che permette l'interazione da parte di quest'ultimo. Pur potendo sembrare tutte interconnesse, restano svincolate le une dalle altre e possono essere richiamate anche indipendentemente dallo stato di esecuzione del programma;
2. Service: è un componente che esegue in background e viene utilizzato per operazioni impegnative o richieste da processi remoti. Per questi motivi non presentano un'interfaccia grafica visibile all'utente;

3. Content provider: rappresenta il gestore di dati condivisi di un'applicazione salvati in una posizione persistente e accessibile. Viene utilizzato dalle applicazioni per accedere a questi dati o per modificare dati privati che non verranno poi condivisi.
4. Broadcast Recever: è un componente che risponde alle richieste in broadcast lanciate a livello di sistema, non dispone di interfaccia utente ma può interagire tramite notifiche nella barra di stato del dispositivo. Sono tipicamente utilizzati come gateway per indirizzare le applicazioni a compiere certe operazioni alla ricezione di un evento. I broadcast sono inviati e ricevuti sotto forma di oggetti Intent.

Ogni tipo di componente ha, come anticipato, il proprio ciclo di vita e può essere attivato da un'altra applicazione attraverso una richiesta, dando così l'impressione generale di eseguire una applicazione unica che risulta in realtà costituita da più Activity.

Gli intent sono messaggi asincroni utilizzati per avviare o richiamare un componente: al suo interno, in caso di activity e service, viene specificato il tipo di azione da richiedere, una URI che indica a quale componente deve essere effettuata e se ci dobbiamo aspettare un risultato da questa invocazione. I broadcast receivers ricevono invece solamente la descrizione dell'evento comunicato in broadcast. I content providers vengono attivati tramite richieste di un Content Resolver, ovvero un gestore di tutte le comunicazioni con il provider, questo crea uno strato di astrazione che attua un primo livello di sicurezza.

Quando un componente riceve un Intent, il sistema avvia il processo di quella applicazione e ne istanzia le classi necessarie al funzionamento per poi rilasciare le risorse impegnate quando non saranno più necessarie.

Le interfacce, i componenti grafici che le implementano e tutte le risorse che riguardano l'aspetto visivo dell'applicazione, vengono dichiarati all'interno di file .xml nel package resource. A differenza delle librerie grafiche di Java, quindi, questo permette una facile creazione, modifica e manutenzione di tutto ciò che riguarda l'aspetto grafico dell'applicazione senza dover intervenire sul codice del file sorgente.

2.3 Analisi delle versioni per piattaforme Java e Android

Come anticipato nell'introduzione, tuProlog è disponibile in varie versioni eseguibile su quattro piattaforme diverse: JVM, .NET, Android e come plug-in Eclipse. Ai fini dell'analisi verranno analizzate e confrontate nello specifico la versione Java e Android.

Dalle figure sottostanti si nota chiaramente che la versione di tuProlog per Java è più completa della rispettiva in Android, in quanto implementa un'interfaccia grafica con più opzioni, supporto al multi-threading e alla comunicazione di rete.

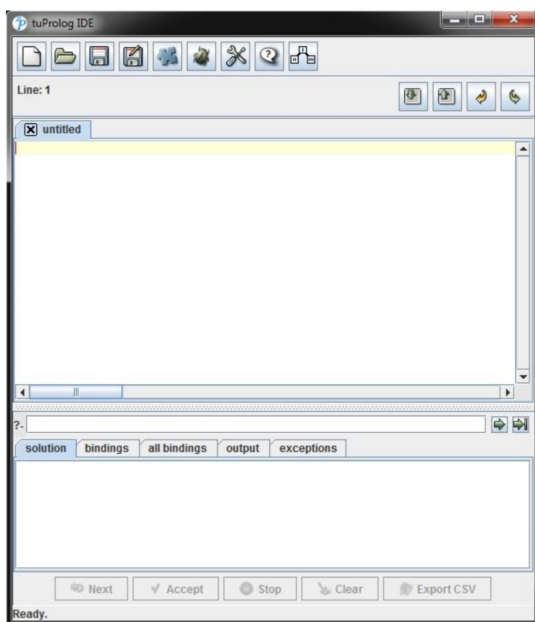


Figura 2.1: Interfaccia Grafica Java

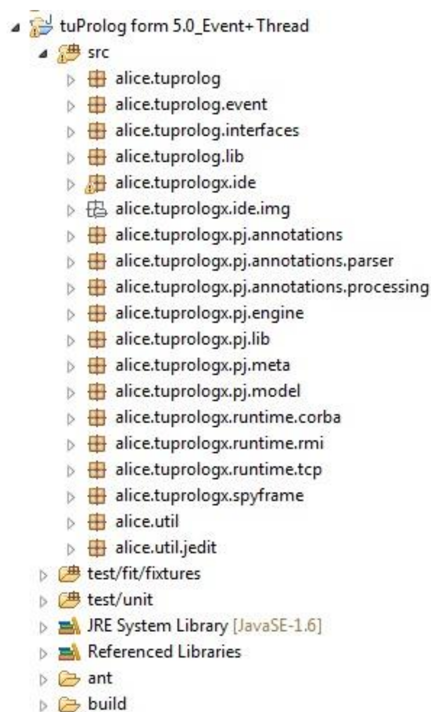


Figura 2.2: Build Path Java

Per soddisfare i criteri di modularità, portabilità e leggerezza ampiamente descritti, il sistema è stato progettato in modo tale da avere una forte suddivisione in classi differenti e l'organizzazione di queste in package con nomi atti ad autodefinirne il contenuto.

Un esempio di quanto questo sistema sia portabile ne è sicuramente la versione Android. Nella versione mobile, infatti, troviamo le stesse classi che nella versione Java costituiscono il motore Prolog, organizzate all'interno di un archivio JAR che verrà poi gestito dalle differenti classi che costituiscono l'applicazione e che la rappresentano graficamente.

2.3.1 tuProlog su piattaforma Java

La versione per Java ha due possibili modalità di esecuzioni: da linea di comando e grafica.

Per poter eseguire l'avvio da console è necessario digitare il comando per avviare da prompt la classe specifica all'interno del file JAR.

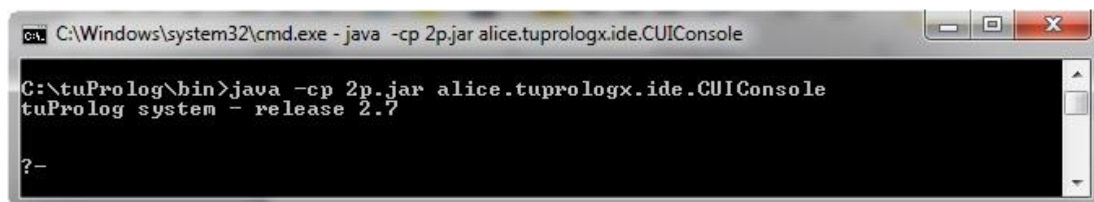


Figura 2.3: Avvio da linea di comando

La modalità da linea di comando, pur non essendo così intuitiva e user friendly come quella grafica, non riporta gli stessi problemi di quest'ultima con l'input da console.

Come evidenziato dai due esempi sottostanti, richiedendo l'esecuzione di due metodi presenti all'interno della libreria di IO, non vi è problema nell'acquisizione del valore immesso e della sua assegnazione alla variabile indicata.

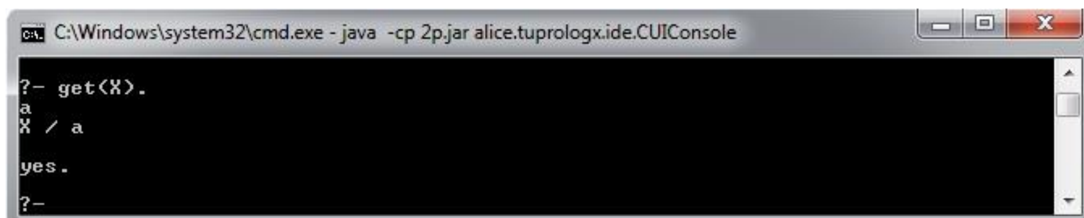


Figura 2.4: risoluzione query get(X). eseguendo da console

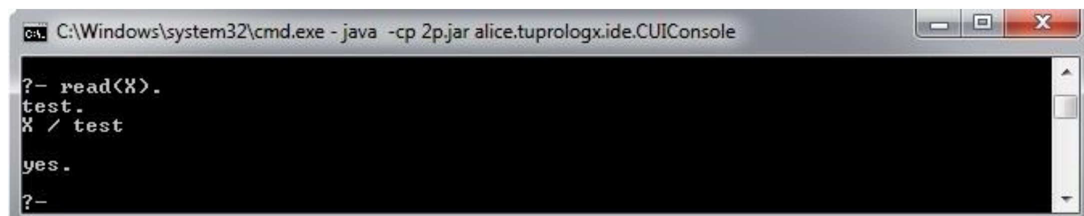


Figura 2.5: risoluzione query read(X). eseguendo da console

Per l'esecuzione grafica, invece, è sufficiente avviare il JAR eseguibile. Al momento dell'avvio il gestore dell'interfaccia grafica provvederà a creare e unire tutti i singoli panel e toolbar che andranno a costituirla i quali a loro volta inizializzeranno i vari Manager: di gestione del motore Prolog, caricamento librerie, teorie etc.

Di fatto l’interfaccia grafica risulta concettualmente divisa in due settori: quello superiore nel quale è possibile settare il funzionamento del motore, ovvero importare le librerie, caricare le teorie, modificarle e salvarle; e una parte inferiore nella quale vengono di fatto effettuate le query e visualizzate le soluzioni, gli assegnamenti logici, l’output e le eccezioni.

Ai fini del lavoro di tesi, ci focalizzeremo sul funzionamento della parte inferiore. In seguito all’inserimento della query e alla pressione del tasto Solve, si susseguono le chiamate per la risoluzione della stessa che passano da un Manager all’altro fino ad arrivare al core del motore, nel quale avviene l’esecuzione del metodo richiesto se risulta realmente presente in una delle librerie caricate.

Al momento, pur essendo le librerie di IO e ISOIO perfettamente caricate all’interno del motore Prolog, l’esecuzione di metodi che prevedono un input da console genera un errore e la terminazione dell’esecuzione.

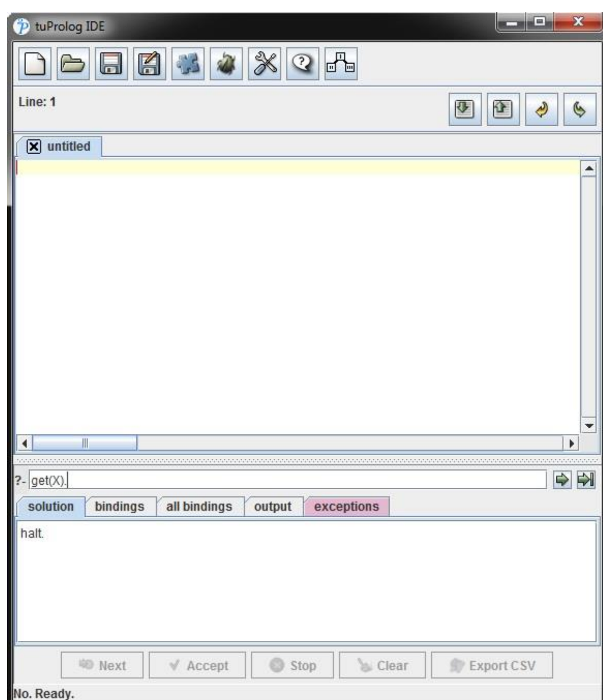


Figura 2.5: risoluzione get(X). eseguendo da GUI

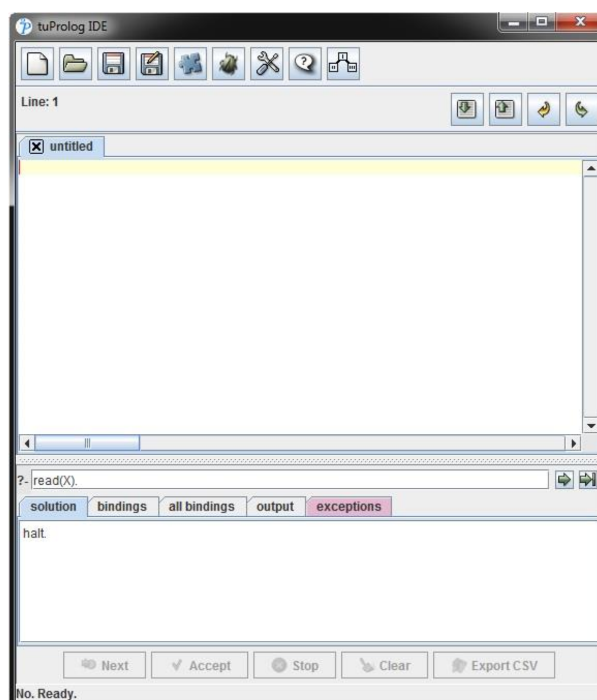


Figura 2.6: risoluzione read(X). eseguendo da GUI

2.3.2 tuProlog su piattaforma Android

L'applicazione per Android è sviluppata graficamente e concettualmente in maniera sostanzialmente diversa da quella per Java. La differenze grafiche trovano spiegazione nella tipologia di applicazione, ovvero mobile: ci si trova, infatti, ad eseguire un'applicazione su un monitor nettamente più piccolo rispetto a quello di un Personal Computer e i comandi vanno principalmente attivati e/o digitati premendo direttamente sul display con tastiera a schermo. Le differenze concettuali invece riguardano la composizione strutturale dell'applicazione per la tipica suddivisione di quest'ultima in componenti di quattro tipologie.

Le activity che gestiscono l'interazione dell'utente con l'applicazione sono state sviluppate ottimizzando gli spazi disponibili e la gestione delle risorse hardware. Dalla foto seguente è netto il divario di complessità tra la versione Android e la versione Java.

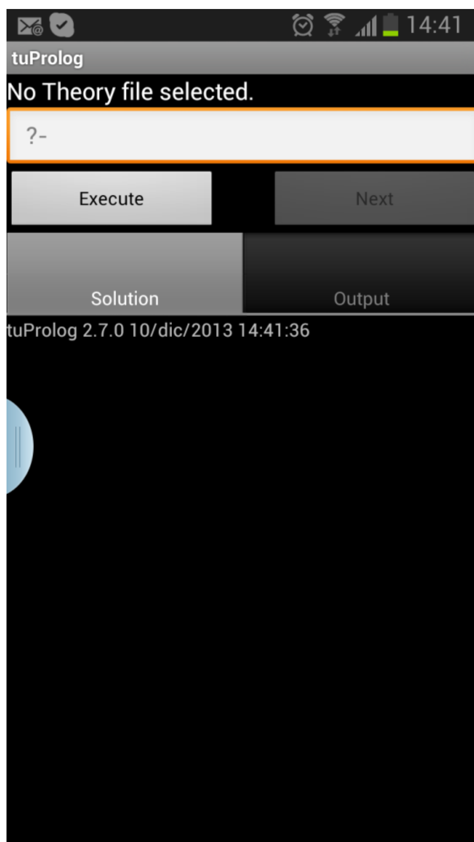


Figura 2.7: Interfaccia Grafica Android

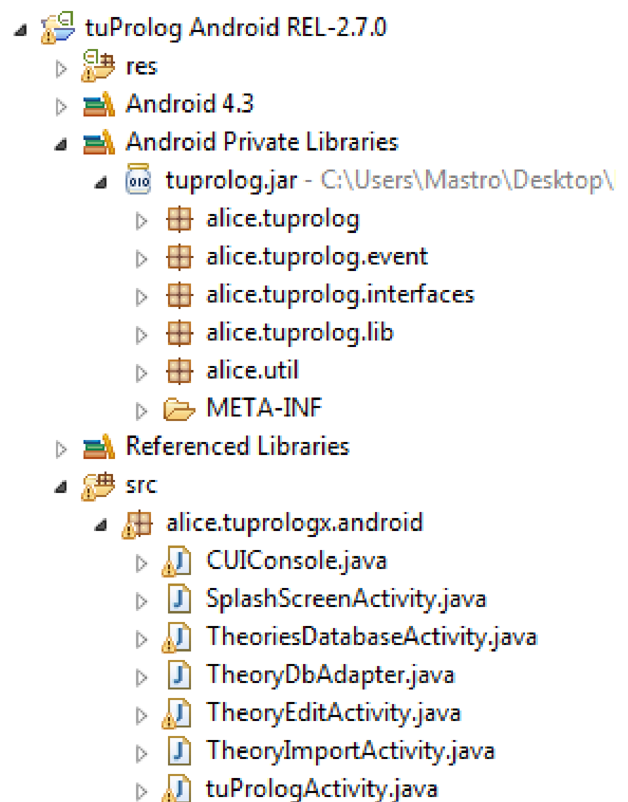


Figura 2.8: Build Path Android

Come si evince dalla figura 2.8, seppur presentandosi in maniera del tutto differente, l'applicativo Android utilizza le stesse classi e librerie che costituiscono il motore Prolog della versione per JVM, implementando il Jar che le racchiude all'interno del progetto in fase di programmazione.

Come nel caso precedente, quindi, al momento della risoluzione della query, il flusso di invocazioni ed eventi di entrambe le architetture è il medesimo. A conferma di ciò, infatti, allo stesso modo ma con risultato diverso, anche la versione Android non supporta l'input da console.

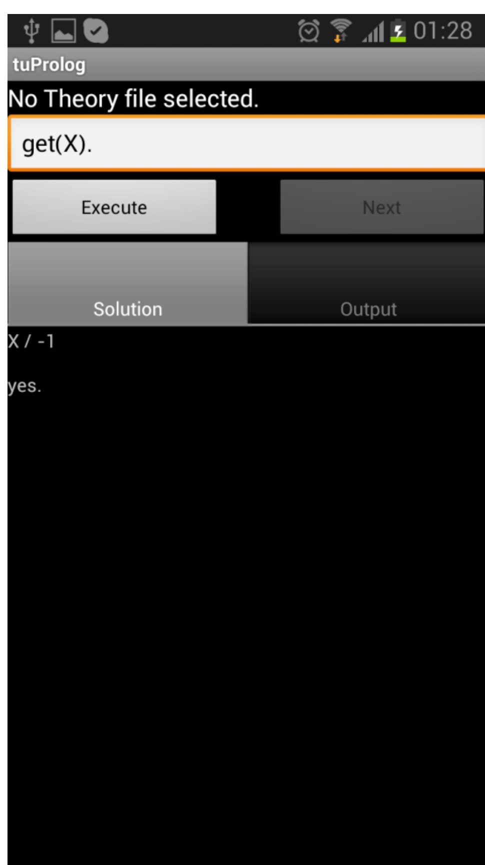


Figura 2.9: risoluzione get(X). eseguita su Android

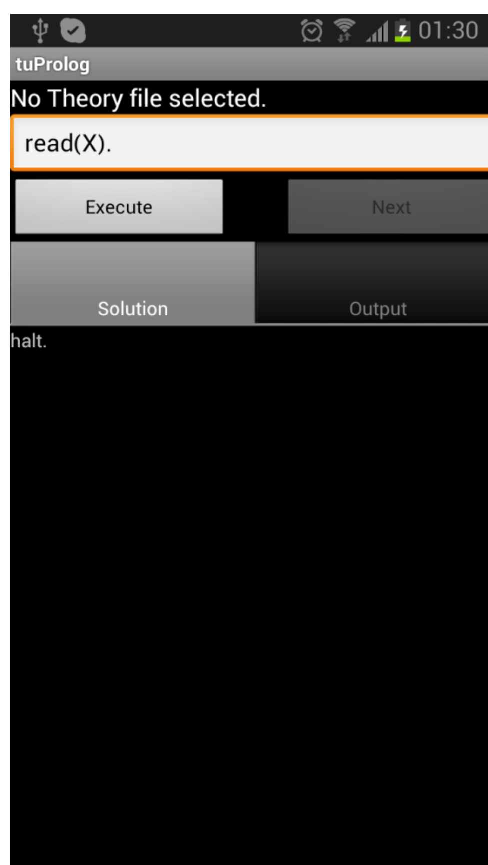


Figura 2.10: risoluzione read(X). eseguita su Android

2.4 Librerie

La IOLibrary e la ISOIOLibrary contengono tutti i predicati che gestiscono l'interazione del motore Prolog con l'input e l'output da/su console. Come anticipato nei paragrafi precedenti, quindi, indipendentemente dall'ambiente di esecuzione di tuProlog, i predicati di I/O sono sempre confinati all'interno di queste librerie standard.

L'attuale organizzazione della IOLibrary prevede come canale di input predefinito lo standard input (System.in) e come canale di output lo standard output (System.out).
Dal momento che la ISOIOLibrary estende la IOLibrary, questi vincoli di mantengono anche in tale libreria.

In particolare, per quanto riguarda l'input, attualmente all'esecuzione di un metodo contenuto in una delle due librerie che richiede un input, il motore si aspetta di ottenere il carattere o la stringa richiesta da System.in. Ciò, a meno che non si stia eseguendo tuProlog da linea di comando, genera inevitabilmente una risposta sbagliata del sistema: *halt.* nella versione Java e *-1* nella versione per Android in quanto essendo l'applicazione eseguita in modalità grafica, non ha le credenziali e la possibilità di accedere allo standard input.

Nella figura seguente si mette in evidenza la cascata di chiamate durante l'esecuzione di una query e l'errore causato nel tentativo di accedere allo standard input.

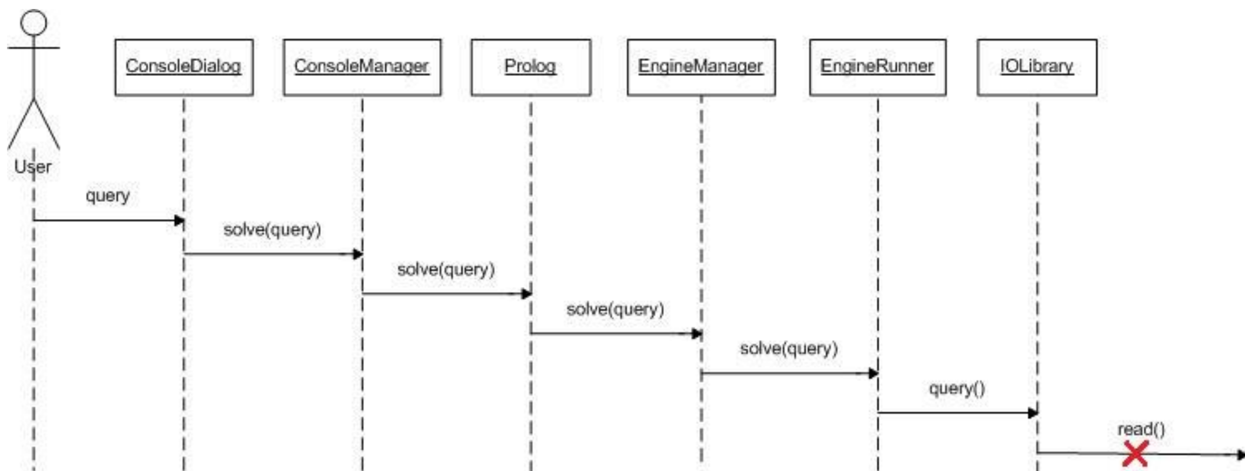


Figura 3.1: Flusso di chiamate per l'esecuzione di query

Come evidenziato dal listato seguente, analizzando il codice delle librerie si nota che i flussi in input e output sono inizializzati staticamente e ne viene passato un riferimento ai metodi

```java
public class IOLibrary extends Library {

    protected String inputStreamName = "stdin";
    protected InputStream inputStream = System.in;
    protected String outputStreamName = "stdout";
    protected OutputStream outputStream = System.out;
    private Random gen = new Random();
```

Figura 3.2: Field della IOLibrary

Da un'analisi più approfondita all'interno del corpo dei vari metodi, è possibile intuire come la libreria sia stata ideata senza prevedere una possibile futura espansione al fine di accettare modalità differenti di input rispetto a quella da console tradizionale. Ciò è evidenziato dal fatto che taluni controlli sulla validità dello stream di input vengono effettuati non comparando il termine in ingresso con il valore della variabile appositamente inizializzata, bensì, con la dicitura "stdin" in modo cablato.

```java
if (inputStream != System.in) {
    try {
        inputStream.close();
    } catch (IOException e) {
        return false;
    }
    inputStream = System.in;
    inputStreamName = "stdin";
}
```

Figura 3.3: Particolare del predicato seen

```java
Struct arg0 = (Struct) arg.getTerm();
if (inputStream != System.in)
    try {
        inputStream.close();
    } catch (IOException e) {
        return false;
    }
if (arg0.getName().equals("stdin")) {
    inputStream = System.in;
} else {
    try {
        inputStream = new FileInputStream(((Struct) arg0).getName());
    } catch (FileNotFoundException e) {
        throw PrologError.domain_error(engine.getEngineManager(), 1,
                "stream", arg0);
    }
}
inputStreamName = ((Struct) arg0).getName();
return true;
```

Figura 3.4: Particolare del predicato see

3.1 Analisi dei requisiti

Per rispettare il requisito di esecuzione duale, si rende necessario separare logicamente il dispositivo di acquisizione dell'elaborazione dell'input, introducendo uno strato software che renda indipendente l'uno dall'altro. Ciò permetterà di introdurre un componente grafico specifico per interagire con l'utente in fase di input, mantenendo però, nel caso di

esecuzione da prompt dei comandi (quindi senza interfaccia grafica), la possibilità di agganciare l'input a stdin come nella versione attuale, senza che la differenza venga percepita dalla libreria.

Per evitare di rendere inutilmente complessa la GUI, il componente in questione dovrà essere visualizzato in foreground solo quando il contesto lo richieda: questo è particolarmente importante nella versione per Android, dove l'aggiunta di un componente grafico sempre in primo piano causerebbe uno spreco di risorse e "invaderebbe" inutilmente lo spazio a disposizione sul display.

Lo strato software di separazione, destinato a fungere da intermediario tra i metodi della libreria e i vari dispositivi di input, dovrà naturalmente intercettare l'intenzione da parte dei vari predicati di input, di leggere da InputStream: ciò pone come requisito che esso incapsuli concettualmente un InputStream, ridefinendone il metodo read come opportuno così da poter restituire ai metodi l'input coerente con la modalità di esecuzione dell'applicazione.

Affinché la modifica possa essere supportata nelle diverse versioni di tuProlog per le varie piattaforme, le eventuali modifiche alle librerie, nonché le nuove classi di supporto allo strato software di separazione, dovranno risiedere all'interno dei package comuni, mentre le componenti grafiche dovranno essere rigorosamente separate.

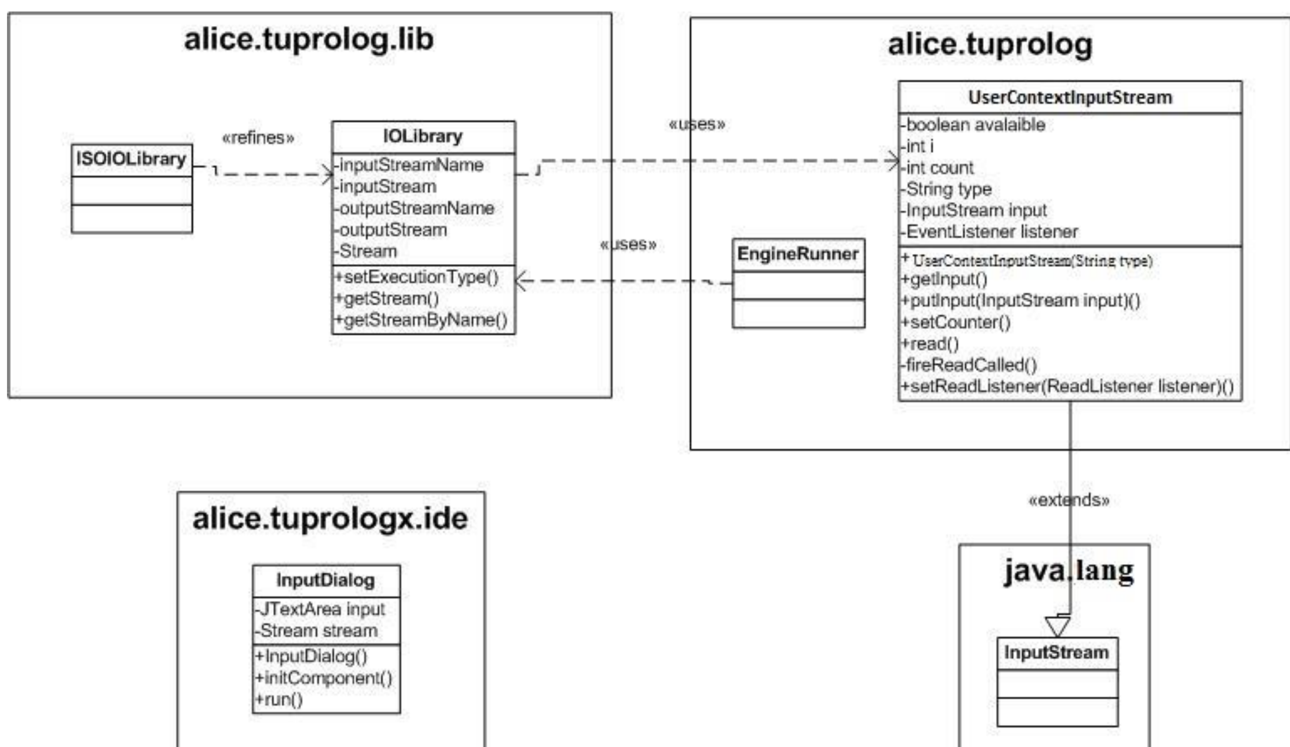


Figura 3.5: Packaging da mantenere per rispettare i requisiti

Le modifiche al software verranno effettuate utilizzando l'ambiente di sviluppo Eclipse, utilizzando l'SDK Android e il linguaggio adottato sarà Java. La versione tuProlog di partenza per effettuare l'estensione sarà la 2.7.2 sia per le modifiche all'ambiente Java sia per le modifiche in Android.

4.1 Modifiche alla IOLibrary

4.1.1 Inserimento strato di separazione

Le modifiche apportate alla libreria mirano a mantenere la possibile esecuzione duale: da linea di comando e da interfaccia grafica. Per questo motivo vengono dapprima introdotte due variabili globali che, appositamente settate, consentono alla libreria, in fase di caricamento dell'applicazione, di reperire il riferimento allo stream di input a cui dovrà fare riferimento.

Un'altra specifica da rispettare è quella della retro compatibilità della libreria, per mantenere la quale vengono mantenuti invariati i valori assegnati a inputStreamName (tipo String) e inputStream (tipo InputStream).

Dalla struttura della libreria è possibile notare come gli unici metodi della classe InputStream a cui viene fatto riferimento durante l'esecuzione dei predicati sono read() e close(). Dal momento che l'input si rende necessario solo all'invocazione del metodo read(), è opportuno fare in modo che il flusso di byte si renda disponibile solo alla sua invocazione. Per far si che ciò possa avvenire senza introdurre alcuna variazione all'interno dei metodi della libreria, rendendo la modifica il più trasparente possibile e facilitando le possibili future espansioni, viene introdotta una nuova classe UserContextInputStream che estende la *java.io.InputStream.*

Inizializzata concorrentemente alla IOLibrary, consente di modificare il metodo read() ereditato dalla InputStream in modo tale che in base al contesto di esecuzione restituisca ai metodi della libreria il flusso di input adeguato. La variabile inputStream della libreria viene quindi inizializzata con un'istanza di UserContextInputStream correttamente formata.

4.1.2 Metodi per il controllo dell'esecuzione

Le modifiche introdotte devono poter essere configurabili dalla entità preposte del sistema.
Affinché ciò sia possibile è necessario aggiungere all'interno della IOLibrary due metodi pubblici che consentano l'impostazione della tipologia di esecuzione e il reperimento del riferimento alla classe che implementa lo strato di separazione.
Queste due proprietà (setExecutionType e getUserContextInputStream) consentono alla GUI di inizializzare la classe UserContextInputStream in modo che reindirizzi le azioni di lettura dallo standard input al Dialog preposto e, in secondo luogo, ottenere un riferimento alla classe appena creata per, come verrà analizzato nel seguito, mettersi in ascolto dell'evento lanciato da questa quando.

Il metodo setExecutionType imposta i campi inputStreamName, inputStream, e inizializza la classe UserContextInputStream in base al contesto di esecuzione corrente.

Il metodo getUserContextInputStream () restituisce al chiamante un riferimento alla classe UserContextInputStream precedentemente inizializzata.

```
public UserContextInputStream getUserContextInputStream()
{
    return this.input;
}

public void setExecutionType(String input)
{
    this.inputStreamName = input;
    this.input = new UserContextInputStream(inputStreamName);
    inputStream = this.input;
}
```

Figura 4.1: Nuovi metodi IOLibrary

4.1.3 Aggiornamento predicati see e seen

Alcuni predicati della IOLibrary svolgono un'azione di controllo sulla validità dell'inputStream corrente effettuando una comparazione con delle costanti. Nel vecchio approccio, questa metodologia non avrebbe creato problemi e si sarebbero intraprese azioni differenti in base al risultato della comparazione.

Con le modifiche introdotte risultano attendibile e valido un flusso di input, correttamente inizializzato dai metodi di cui sopra, anche se questi differisce dallo standard input.
Per poter modificare le modalità di comparazione dei predicati in esame, si rende necessario un metodo che in base alla modalità di esecuzione del software, restituisca l'inputStream corrente con il quale fare la comparazione.

Per ottenere ciò è stato introdotto il metodo getInputStreamByContext(String typeExecution) descritto nella figura seguente.

```
public InputStream getInputStreamByContext(String inputStreamName)
{
    InputStream result = null;

    if(inputStreamName.compareTo("stdin") == 0)
        result = System.in;
    else if (inputStreamName.compareTo("graphic") == 0)
        result = input;
    return result;
}
```

Figura 4.2: Codice della proprietà getInputStreamByContext(String str)

Il metodo see è usato per creare o aprire un input stream: il predicato è vero se lo StreamName è una stringa rappresentante il nome di un file da creare o accedere come flusso di input; se invece la comparazione con la string “stdin” ha successo, viene utilizzato lo standard input stream corrente.

Dal momento che le modifiche alla libreria consentiranno la lettura da un input anche differente da quello che ha rappresentato lo standard fino ad ora, è necessario modificare i termini di comparazione. In questo modo è possibile eseguire un input da interfaccia grafica senza che questo venga interpretato come differente da “stdin” e quindi rappresentante il nome di un file.

```
Struct arg0 = (Struct) arg.getTerm();
if (inputStream != System.in)
    try {
        inputStream.close();
    } catch (IOException e) {
        return false;
    }
if (arg0.getName().equals(inputStreamName)) {
    inputStream = getInputStreamByContext(inputStreamName);
} else {
    try {
        inputStream = new FileInputStream(((Struct) arg0).getName());
    } catch (FileNotFoundException e) {
        throw PrologError.domain_error(engine.getEngineManager(), 1,
                "stream", arg0);
    }
}
inputStreamName = ((Struct) arg0).getName();
return true;
```

Figura 4.3: Particolare del predicato see modificato

Il metodo seen è utilizzato per chiudere il flusso di input precedente aperto. Il predicato ha risultato positivo se l’azione di chiusura risulta possibile.

Allo stesso modo del precedente, viene effettuata la modifica concettuale in modo che possa essere considerato valido anche l'input stream da interfaccia grafica e non solo quello da System.in.

```
if (inputStream != getInputStreamByContext(inputStreamName)) {
    try {
        inputStream.close();
    } catch (IOException e) {
        return false;
    }
    inputStream = getInputStreamByContext(inputStreamName);
}
return true;
```

Figura 4.4: Particolare del predicato seen modificato

Il dettaglio delle modifiche effettuate è approfondito in Appendice A.

4.2 Modifiche alla ISOIOLibrary

La libreria ISOIOLibrary estende la IOLibrary e da questa, infatti, ottiene i riferimenti ai flussi di input e output. Le modifiche dei campi e dei metodi effettuate alla IOLibrary hanno perciò effetto anche sui predicati della ISOIO.

Anche in questa libreria, si è resa necessaria la stessa modifica effettuata nei predicati della IOLibrary, in quanto anche qui non veniva presa in considerazione la possibilità dell'esistenza di un flusso di input valido alternativo a stdin.

I metodi interessati dalla modifica sono: close_1(), close_2(), get_char_2(), get_code_2(), peek_char_1(), peek_char_2(), e find_output_stream().

4.3 La classe UserContextInputStream

La classe UserContextInputStream viene ideata e introdotta all'interno del package *alice.tuprolog* per attuare le modifiche richieste in modo del tutto trasparente e affidabile.

Il ruolo chiave di questa classe è svolto dal metodo read(). Questo metodo viene invocato dai predicati della IOLibrary per poter ottenere l'input necessario alla risoluzione del goal.

Al momento dell'invocazione, la read() effettuerà il confronto con la variabile della classe UserContextInputStream, inizializzata dal costruttore di quest'ultima, che rappresenta la tipologia di esecuzione corrente (linea di comando o GUI).

Nel caso il programma fosse eseguito da console, la read() restituirebbe il flusso di byte proveniente da System.in. Al contrario, se ci si trovasse avviati in modalità grafica, verrebbe intrapresa una strada alternativa.

Il metodo read() andrebbe a lanciare un ReadEvent, appositamente creato, e si metterebbe in attesa di un flusso di input valido tramite una delle due proprietà synchronized. La classe che implementa l'ascoltatore dell'evento genererà un nuovo Thread (tipicamente una classe grafica) che si occuperà di presentare al cliente una finestra nella quale inserire i dati richiesti, della trasformazione di questi in un flusso di byte e della loro restituzione al metodo read(), sbloccando così il Thread principale del programma ancora in attesa di un input valido.
Di seguito si evidenzia lo schema delle chiamate che si scatena al momento della risoluzione di una query comparando la situazione precedente e seguente le modifiche apportate.

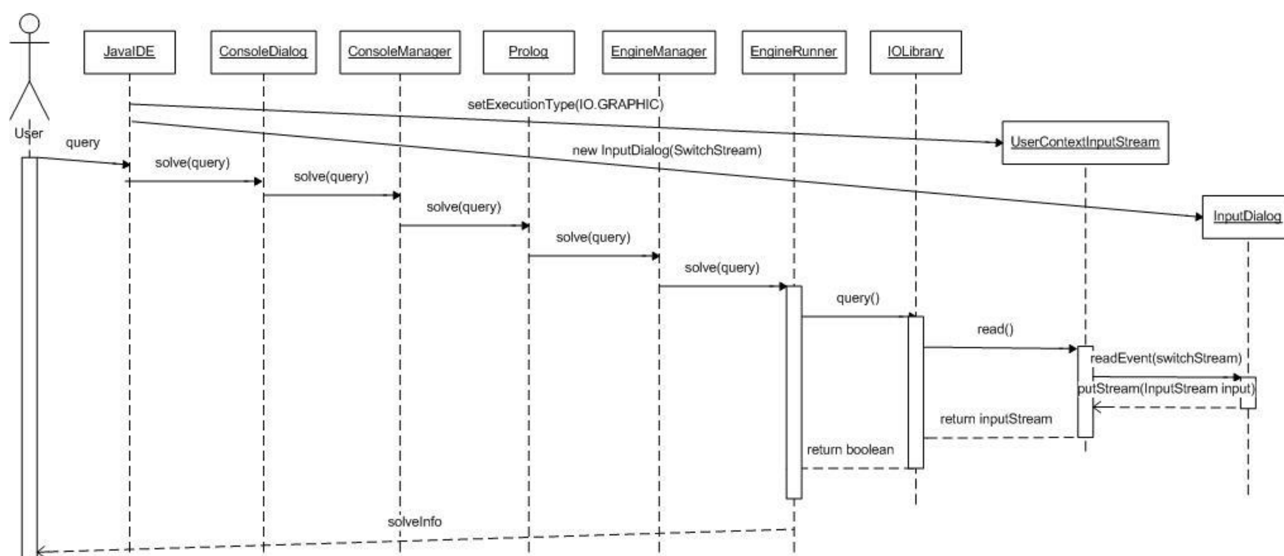


Figura 4.5: Nuovo flusso di chiamate per l'esecuzione di query dopo la modifica effettuata

4.4 Modifiche alla GUI Java

Si rende altresì necessaria la modifica delle classi che implementano l'interfaccia grafica dell'applicativo per l'ambiente Java in modo da poter impostare e gestire il tipo di esecuzione e livello di libreria IO.

Una volta progettato lo strato di intermediazione necessario, infatti, è opportuno individuare in che modo le restanti componenti dell'applicazione possano interagire con questo per un'esecuzione ottimale

Dal punto di vista logico, si necessita di un agente che si incarichi di utilizzare il metodo setExecutionType(type) della IOLibrary settando la modalità di esecuzione, in questo caso grafica. Allo stesso tempo, questa entità deve possedere, o essere in grado di ottenere un riferimento alla IOLibrary per poter interagire con essa.

La classe JavaIDE.java è preposta all'assemblaggio di tutte le componenti grafiche costituenti l'interfaccia utente; in aggiunta all'aspetto grafico è la prima classe, in base alla suddivisione dei compiti scelta dal progettista, che implementa e inizializza il motore Prolog.

Avendo quindi qui un'istanza del motore Prolog, il quale carica a run-time le librerie standard citate nei capitoli precedenti, si ha la possibilità di recuperare l'istanza della IOLibrary e, tramite i metodi di nuova introduzione, settare la tipologia di esecuzione (in questo caso grafica).

Contestualmente, sfruttando il metodo getUserContextInputStream() della IOLibrary, si definisce e inizializza la classe inputDialog che estende JFrame e realizza la finestra vera e propria per l'inserimento dell'input.

La classe inputDialog, inserita nel package *alice.tuprologx.ide*, al momento della inizializzazione si imposta come listener del ReadEvent presso l'istanza della classe UserContextInputStream passata come parametro al costruttore. Il gestore dell'evento da vita a un nuovo Thread nel quale si esegue una nuova istanza di inputDialog; all'evento "pressione tasto enter" la stringa inserita viene trasformata in un flusso di byte e restituita alla classe UserContextInputStream sbloccando così il lock e ristabilendo il flusso primario di esecuzione.

In ultimo, è stato introdotto nel gestore dell'evento onClick del Solve Button all'interno della classe ConsoleManager.java, una chiamata al metodo setCounter() della classe UserContextInputStream. Il metodo in esame consente di resettare ad ogni nuova query il contatore della classe in modo che, all'avvio della read(), se il contatore è uguale a zero, viene generato l'evento, creato il nuovo Thread e inserito il dato; in caso contrario significa che il predicato che ha invocato la read() necessita di un ulteriore accesso al flusso e quindi prosegue esaurendo tutto il flusso di input presente.

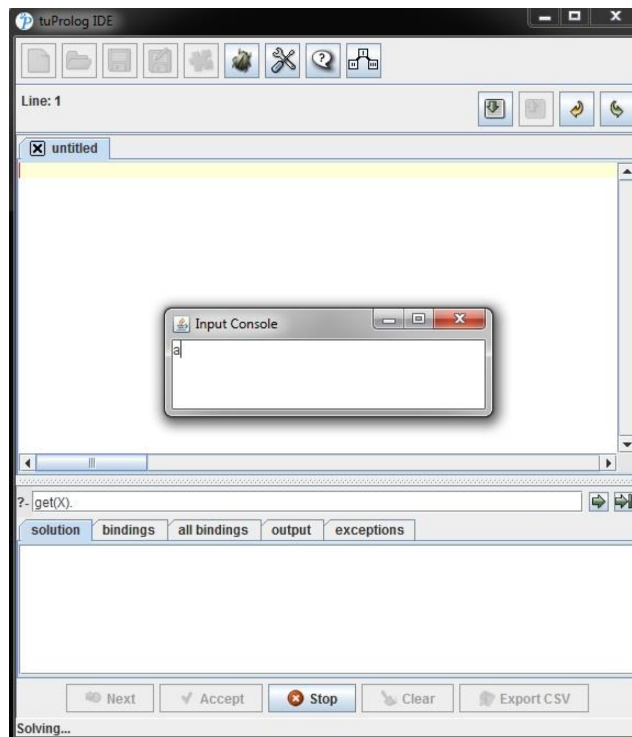


Figura 4.6: Interfaccia grafica Java dopo la modifica

4.5 Modifiche alla CUIConsole Android

Le modifiche sinora introdotte avevano come fine ultimo, oltre alla stabilità, la portabilità su altri ambienti di esecuzione.

Il riscontro dell'efficacia della soluzione adottata la si può notare dal fatto che le modifiche alla libreria e l'introduzione della nuova classe UserContextInputStream risiedono all'interno dei package contenuti nell'archivio jar implementato da tuProlog Android per l'esecuzione.

In base alle modifiche attuate alla versione Java ed ereditate sulla piattaforma Android, si ipotizza che in caso di query che necessiti di input da console, la classe UserContextInputStream lanci un evento e attenda un input stream valido per sbloccare il Thread principale.

Dal momento che le applicazioni per Android sono organizzate in modo completamente diverso da quelle per Java pur essendo scritte nello stesso linguaggio, e che la complessità della versione per Android è molto inferiore alla precedente, la metodologia di risoluzione si discosterà un po' da quella precedentemente introdotta seppur mantenendo lo stesso filo logico.

La tuPrologActivity (activity principale del programma) al momento della creazione crea un nuovo Thread nel quale esegue la classe CUIConsole che si occupa della gestione del motore Prolog e l'interazione dello stesso con l'interfaccia grafica implementata dalla Activity primaria.

Similmente alla JavaIDE, spetta quindi alla CUIConsole reperire il riferimento alla IOLibrary caricata dal motore Prolog e settare la tipologia di esecuzione in corso. Per mezzo della proprietà getUserContextInputStream() della libreria è poi possibile recuperare un riferimento alla UserContextInputStream correttamente inizializzata e registrarsi presso di essa come ascoltatore dell'evento Read.

Durante l'esecuzione di un predicato della libreria di IO, quindi, sarà la CUIConsole ad intercettare il ReadEvent dando vita ad un nuovo Thread che provvederà a richiedere all'utente il valore da immettere e restituire tramite il metodo synchronized putInput(InputStream input) al Thread principale in attesa, il flusso di byte in input.

Per motivi di chiarezza e velocità di esecuzione, è stata scelta un'istanza della classe grafica AlertDialog come metodologia di presentazione della richiesta all'utente. Alla pressione del tasto <OK> viene restituito il valore inserito convertito in flusso di byte, alla pressione del tasto <CANCEL>, invece, viene restituito alla UserContextInputStream il terminatore di riga "-1".

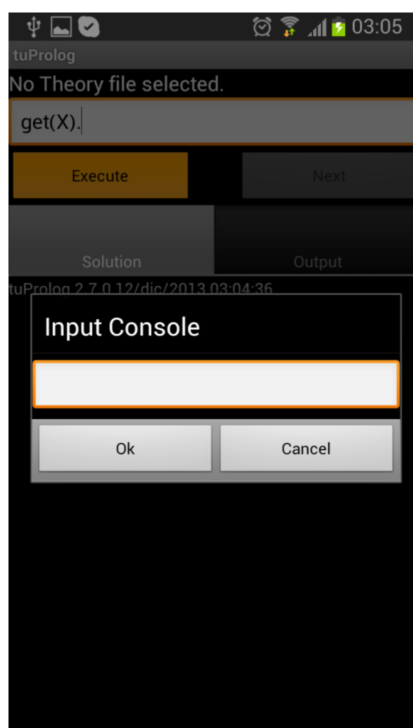


Figura 4.7: Interfaccia grafica Android dopo la modifica

Capitolo 5 *Collaudo*

Le procedure di testing sono fondamentali per sincerarsi della effettiva funzionalità e stabilità delle modifiche introdotte.

L'esecuzione di un test può portare a tre diversi risultati:

- Successo (success), se il risultato ottenuto coincide con ciò che ci si aspettava;
- Fallimento (failure), quando l'oggetto del test non ha prodotto un risultato coincidente con quello atteso;
- Errore (error), quando vi sono errori nell'esecuzione del test o nell'oggetto stesso del test.

5.1 *Testing delle librerie*

I test di libreria presenti si dividono in test dei metodi e test delle eccezioni lanciate dai metodi.

Il requisito di retro compatibilità delle modifiche effettuate trova riscontro nell'esito positivo di tutti i test previsti per le librerie prese in esame.

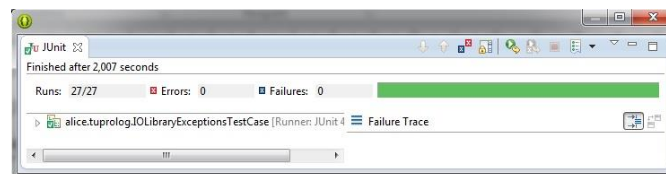


Figura 5.1: Esito test eccezioni IOLibrary

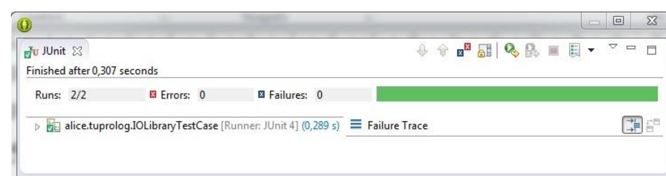


Figura 5.2: Esito test metodi IOLibrary

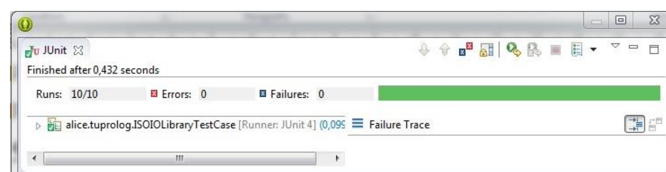


Figura 5.3: Esito test metodi ISOIOLibrary

5.2 Collaudo dell’interfaccia grafica Java

Per quanto riguarda il testing dell’interfaccia grafica, è necessario testare l’applicazione per ogni tipo di input o scelta che l’utente possa effettuare durante l’utilizzo.

Nei successivi screenshots si evidenzia il comportamento del sistema utilizzando i due metodi get() e read() della IOLibrary in presenza di un input valido, di un input inesistente e di una chiusura forzata della finestra tramite.

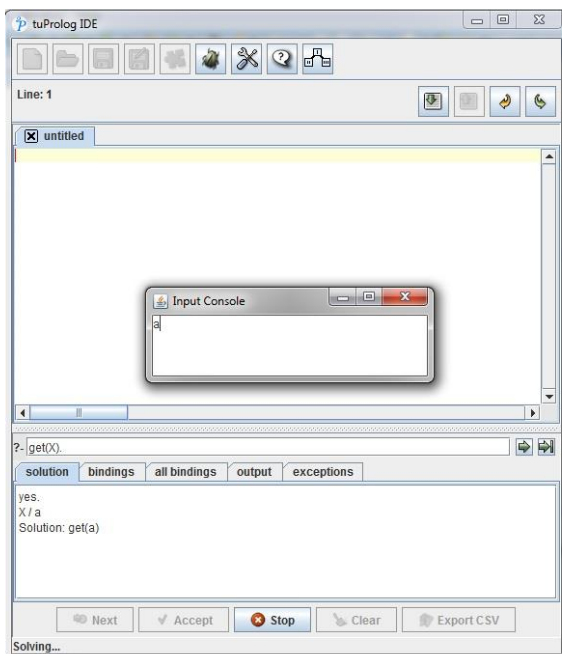


Figura 5.4: Esito predicato get con valore corretto

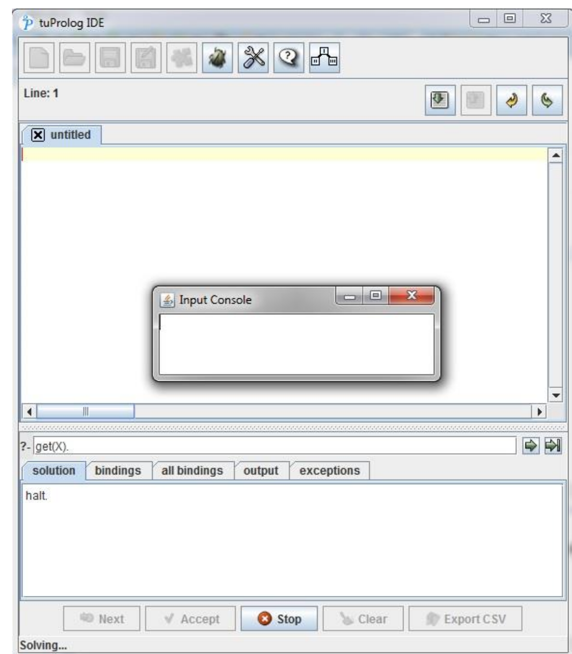


Figura 5.5: Esito predicato get con valore scorretto

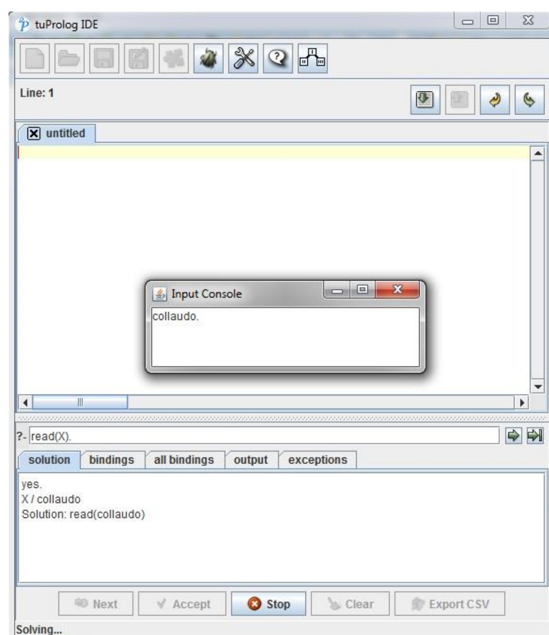


Figura 5.6: Esito predicato read con valore corretto

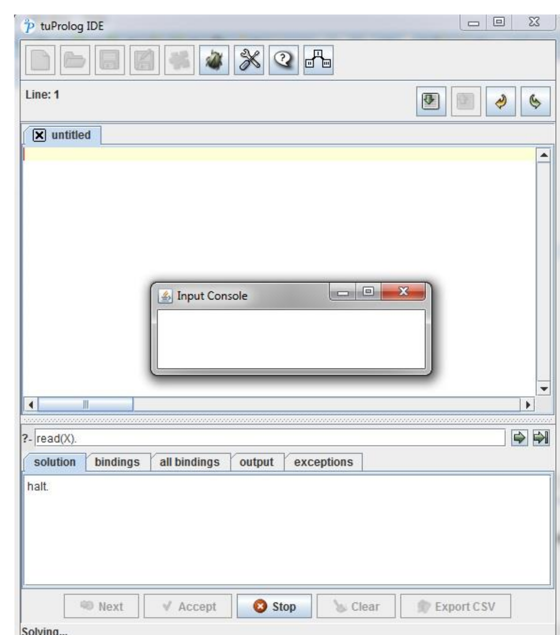


Figura 5.7: Esito predicato read con valore scorretto

5.3 Collaudo dell'interfaccia grafica Android

L'SDK Android mette a disposizione due tipologie differenti di testing delle applicazioni con esso sviluppate:

1. Testing su dispositivi virtuali
2. Testing su dispositivi reali

5.3.1 Dispositivi virtuali

Il testing su dispositivi virtuali viene utilizzato intensivamente durante le fasi di programmazione per testare gli approcci intrapresi e le metodologie utilizzate. L'applicazione verrà testata su macchina virtuale con versione di Android numero 4.4 .

Similmente alla versione per Java, viene messa alla prova l'implementazione effettuata in Android constatando la medesima risposta del sistema ai vari test.

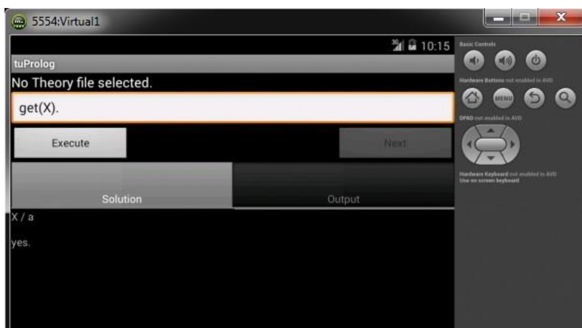


Figura 5.8: Esito predicato get con valore corretto

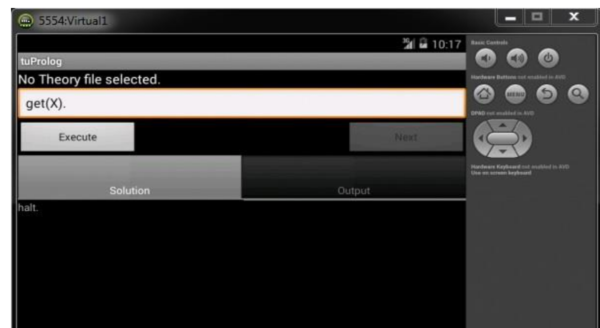


Figura 5.9: Esito predicato get con valore scorretto

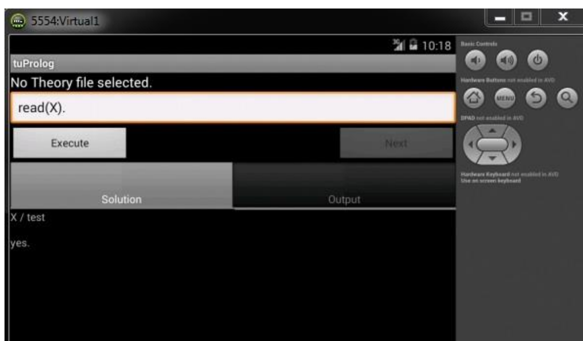


Figura 5.10: Esito predicato read con valore corretto

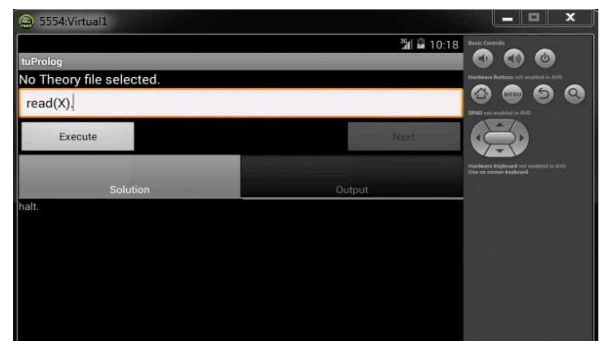


Figura 5.11: Esito predicato read con valore scorretto

5.3.2 Dispositivi reali

Altrettanto importante è il testing su dispositivi reali per verificare eventuali incompatibilità con varie versioni di sistema operativo e, non meno importante, l'aspetto grafico in base alle dimensioni dello stesso.

Per questi motivi sono stati scelti due dispositivi di tipologie differenti tra loro:

- Tablet Kennex T7021 con Android v 4.0.4;
- Smartphone Samsung Galaxy S3 (GT-I9300) con Android v 4.1.2.

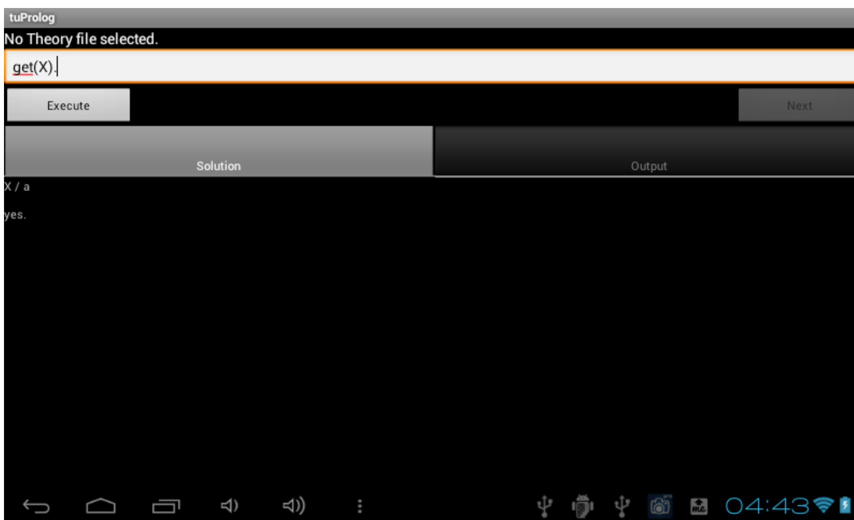


Figura 5.12: get con valore corretto su Tablet

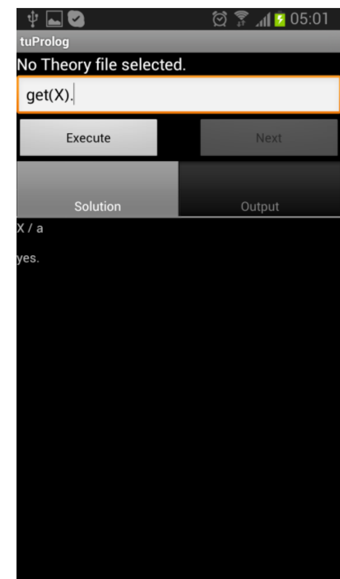


Figura 5.14: get con valore corretto su Smartphone

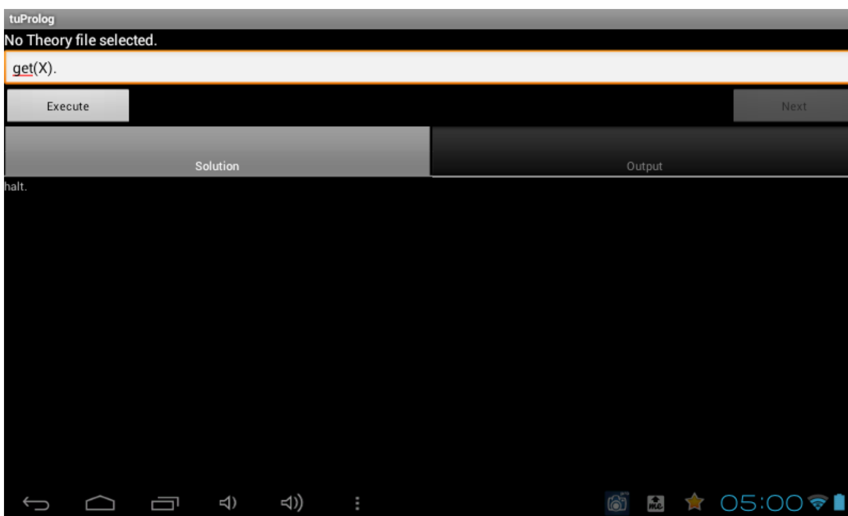


Figura 5.13: Esito predicato get con valore scorretto su Tablet

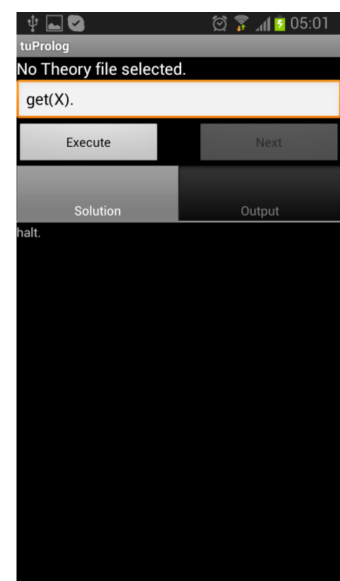


Figura 5.15: get con valore scorretto su Smartphone

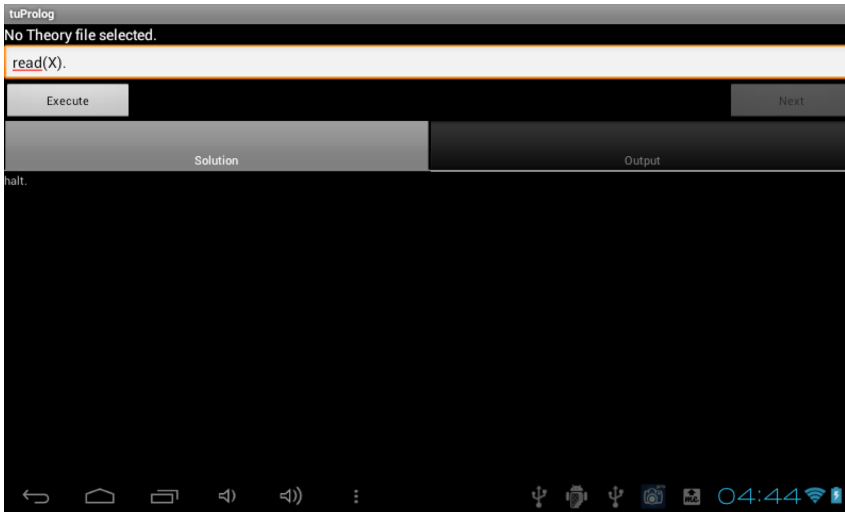


Figura 5.16: read con valore corretto su Tablet

Figura 5.18: read con valore corretto su Smartphone

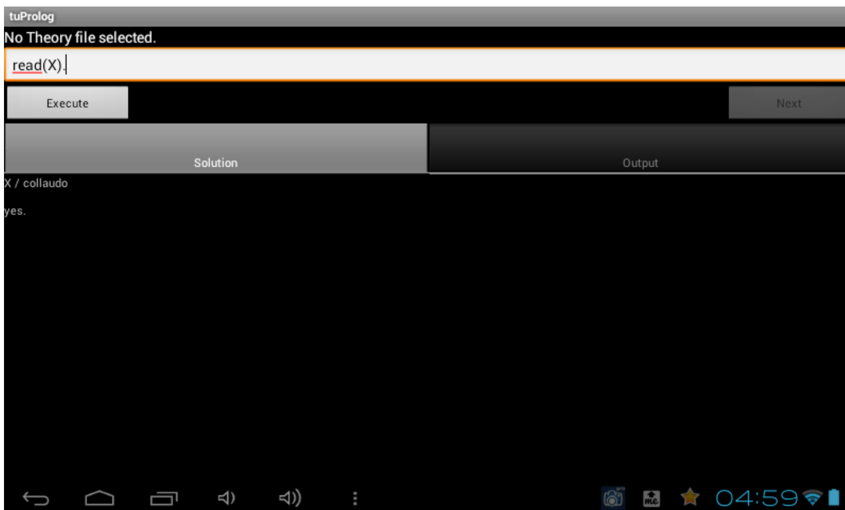


Figura 5.17: read con valore scorretto su Tablet

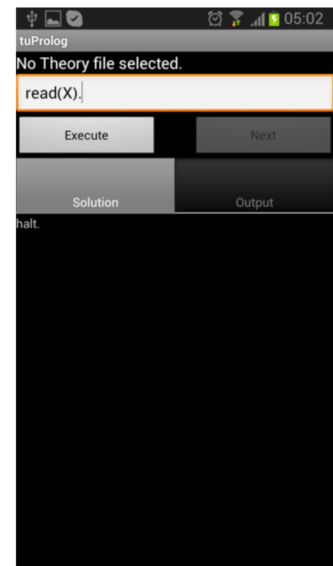


Figura 5.19: read con valore scorretto su Smartphone

Conclusioni

Lo scopo di questa tesi era l'estensione dell'interfaccia grafica dell'interprete tuProlog per Android: nello specifico la reingegnerizzazione delle librerie di input e output in modo da estendere le modalità di input, fino ad ora confinate alla versione avviabile da console.

Per ottenere questo risultato è stato necessario comprendere il funzionamento dell'applicazione a fronte dell'inserimento di una query. Al fine di rendere la modifica portabile anche agli altri ambienti di esecuzione, sono state studiate le strutture dei vari progetti, in particolare la versione per JVM e per Android.

Dopo aver compreso il funzionamento dell'applicazione e le differenze tra i vari ambienti di applicazione, è stato studiato il modo in cui inserirsi nel flusso di esecuzione per intercettare le richieste dei metodi che accedono all'input.

La soluzione scelta rispetta i requisiti e le specifiche imposti dal progetto e raggiunge gli obiettivi preposti. L'esecuzione con successo dei test è la conferma che i metodi delle librerie, seppur modificati, vengono eseguiti tuttora in modo corretto.

Lo studio sulla portabilità ha consentito di dare luogo ad un'implementazione della soluzione che getta le basi per l'estensione futura della modifica introdotta.

L'applicazione così ottenuta offre la possibilità di estendere le possibilità di input, finora confinato allo standard input, ad altri contesti come ad esempio la comunicazione di rete dato il supporto già presente nella versione Java.

Alla luce delle evoluzioni dei dispositivi mobile nel campo dei comandi vocali, un ulteriore sviluppo, prettamente per la versione Android quindi, potrebbe essere quello di introdurre un campionatore vocale che consenta l'inserimento vocale del carattere o della stringa richiesta vocalmente: ne semplificherebbe ulteriormente l'utilizzo in caso di display di dimensioni ridotte.

Appendice A

Aggiornamento IOLibrary

Per rispettare le specifiche di progetto e i requisiti di retro compatibilità è stato necessario introdurre delle modifiche alla libreria di IO.

Sono state inserite due variabili final di tipo string che consentono alla libreria di ottenere il corretto flusso di input in base alla modalità di esecuzione

```
@SuppressWarnings("serial")
public class IOLibrary extends Library {


    protected String inputStreamName = "stdin";
    protected InputStream inputStream = System.in;
    protected String outputStreamName = "stdout";
    protected OutputStream outputStream = System.out;
    private Random gen = new Random();
```

```
@SuppressWarnings("serial")
public class IOLibrary extends Library {

    public final String CONSOLE = "stdin";
    public final String GRAPHIC = "graphic";
    protected UserContextInputStream input;

    protected String inputStreamName = "stdin";
    protected InputStream inputStream = System.in;
    protected String outputStreamName = "stdout";
    protected OutputStream outputStream = System.out;
    private Random gen = new Random();
```

Figura A.1: Comparazione field IOLibrary

Sono poi stati inseriti tre metodi pubblici che consentono: al sistema di interagire con la libreria impostando l'input stream consono all'esecuzione in corso, e ai predicati della libreria di effettuare confronti coerenti con le modifiche strutturali apportate.

```
public UserContextInputStream getUserContextInputStream()
{
    return this.input;
}

public void setExecutionType(String input)
{
    this.inputStreamName = input;
    this.input = new UserContextInputStream(inputStreamName);
    inputStream = this.input;
}

public InputStream getInputStreamByContext(String inputStreamName)
{
    InputStream result = null;

    if(inputStreamName.compareTo(CONSOLE) == 0)
        result = System.in;
    else if (inputStreamName.compareTo(GRAPHIC) == 0)
        result = input;
    return result;
}
```

Figura A.2: Nuovi metodi introdotti

il terzo metodo introdotto trova applicazione nel corpo dei predicati di see e seen in quanto viene utilizzato dagli stessi per effettuare un controllo tra l'input corrente e quello impostato.

```
public boolean see_1(Term arg) throws PrologError {
    arg = arg.getTerm();
    if (arg instanceof Var)
        throw PrologError.instantiation_error(engine.getEngineManager(), 1);
    if (!arg.isAtom()) {
        throw PrologError.type_error(engine.getEngineManager(), 1, "atom",
                arg);
    }
    Struct arg0 = (Struct) arg.getTerm();
    if (inputStream != System.in)
        try {
            inputStream.close();
        } catch (IOException e) {
            return false;
        }
    if (arg0.getName().equals("stdin")) {
        inputStream = System.in;
    } else {
        try {
            inputStream = new FileInputStream(((Struct) arg0).getName());
        } catch (FileNotFoundException e) {
            throw PrologError.domain_error(engine.getEngineManager(), 1,
                    "stream", arg0);
        }
    }
    inputStreamName = ((Struct) arg0).getName();
    return true;
}
```

```
public boolean see_1(Term arg) throws PrologError {
    arg = arg.getTerm();
    if (arg instanceof Var)
        throw PrologError.instantiation_error(engine.getEngineManager(), 1);
    if (!arg.isAtom()) {
        throw PrologError.type_error(engine.getEngineManager(), 1, "atom",
                arg);
    }
    Struct arg0 = (Struct) arg.getTerm();
    if (inputStream != System.in)
        try {
            inputStream.close();
        } catch (IOException e) {
            return false;
        }
    if (arg0.getName().equals(inputStreamName)) {
        inputStream = getInputStreamByContext(inputStreamName);
    } else {
        try {
            inputStream = new FileInputStream(((Struct) arg0).getName());
        } catch (FileNotFoundException e) {
            throw PrologError.domain_error(engine.getEngineManager(), 1,
                    "stream", arg0);
        }
    }
    inputStreamName = ((Struct) arg0).getName();
    return true;
}
```

Figura A.3: Comparazione predicato see IOLibrary

```
public boolean seen_0() {
    if (inputStream != System.in) {
        try {
            inputStream.close();
        } catch (IOException e) {
            return false;
        }
        inputStream = System.in;
        inputStreamName = "stdin";
    }
    return true;
}
```

```
public boolean seen_0() {
    if (inputStream != getInputStreamByContext(inputStreamName)) {
        try {
            inputStream.close();
        } catch (IOException e) {
            return false;
        }
        inputStream = getInputStreamByContext(inputStreamName);

    }
    return true;
}
```

Figura A.4: Comparazione predicato seen IOLibrary

Stream: strato di separazione

Si tratta della classe fondamentale della modifica. Al momento della creazione, viene settata la variabile interna type che distingue tra i metodi di esecuzione (da linea di comando o grafica).

Il metodo read(), una volta invocato, in base al valore di type ha possibilità di gestire l'input dalla console standard oppure generare un evento che verrà poi gestito dalla classe che costruisce il dialog.

Nel caso si stia eseguendo in modalità grafica, il metodo read() genera il ReadEvent e si mette in attesa che la variabile input cambi il suo contenuto. È compito della classe che gestisce l'input grafico, poi, restituire un valore coerente e sbloccare il processo in attesa.

```java
package alice.tuprolog;

import java.io.IOException;
import java.io.InputStream;
import java.util.EventListener;

import alice.tuprolog.event.ReadEvent;
import alice.tuprolog.event.ReadListener;


public class UserContextInputStream extends InputStream implements Runnable{

	private boolean avalaible;
	private int i;
	private int count;
	private String type;
	private InputStream input;
	private EventListener listener;



	public UserContextInputStream(String type)
	{
		this.type = type;
		this.count = 0;
		this.input = null;
	}

	public synchronized InputStream getInput()
	{
		while (avalaible == false){
			try {
				wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		avalaible = false;
		notifyAll();
		return this.input;
	}

	public synchronized void putInput(InputStream input)
	{
		while (avalaible == true){
			try {
				wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		this.input = input;
		avalaible = true;
		notifyAll();
	}

	public void setCounter(){
		count = 0;
		input = null;
	}
```

```java
	public int read() throws IOException
	{
		if(type.compareTo("console") == 0)
		{

			try {
				while((i = System.in.read()) != -1)
				{
					return i;
				}
			} catch (IOException e) {
				e.printStackTrace();
			}

		}
		else if (type.compareTo("graphic") == 0)
		{
			if(count == 0)
			{

				fireReadCalled();
				getInput();
				count++;
			}

				try {
					do
					{
						i = input.read();
					}while (i < 0x20 && i >= 0);
				} catch (IOException e) {
					e.printStackTrace();
				}
		}
		return i;
	}

	private void fireReadCalled()
	{
		ReadEvent event = new ReadEvent(this);
		((ReadListener) listener).readCalled(event);
	}

	public void setReadListener(ReadListener r)
	{
		this.listener = r;
	}

	@Override
	public void run()
	{
	}
}
```

ReadEvent

Classe che implementa l'evento Read lanciato dalla classe Stream in caso di esecuzione grafica.

```
package alice.tuprolog.event;

import java.util.EventObject;
import alice.tuprolog.*;

public class ReadEvent extends EventObject {

	private static final long serialVersionUID = 1L;
	private alice.tuprolog.UserContextInputStream userContextInputStream;

	public ReadEvent(Stream str) {
		super(str);
		this. userContextInputStream = str;
	}

	public UserContextInputStream getUserContextInputStream()
	{
		return this. userContextInputStream;
	}

}
```

ReadListener

Rappresenta l'interfaccia che le classi che si vogliono mettere in ascolto dell'evento dovranno implementare.

```
package alice.tuprolog.event;

import java.util.EventListener;

public interface ReadListener extends EventListener{

	public void readCalled(ReadEvent event);
}
```

InputDialog

Creata e inizializzata dalla classe ConsoleManager, si registra presso la classe Stream come listener del ReadEvent.

Alla ricezione dell'evento, si avvia all'interno di un nuovo Thread che ha il compito di ottenere l'input dall'utente e restituirlo alla classe Stream sbloccando il Thread principale in attesa.

```
package alice.tuprologx.ide;

import java.awt.Dimension;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.io.ByteArrayInputStream;

import javax.swing.JFrame;
import javax.swing.JTextArea;

import alice.tuprolog.UserContextInputStream;
import alice.tuprolog.event.ReadEvent;
import alice.tuprolog.event.ReadListener;


public class InputDialog extends JFrame implements Runnable{

	private static final long serialVersionUID = 1L;
	private JTextArea testo;
	private UserContextInputStream userContextInputStream;

	public InputDialog(Stream str)
	{
		initComponent();
		userContextInputStream = str;
		userContextInputStream.setReadListener(new ReadListener(){
			@Override
			public void readCalled(ReadEvent event) {

				Thread t = new Thread();
				t.start();
				setVisible(true);
			}
		});
	}

	public void initComponent()
	{
		this.setTitle("Input Console");
		setSize(new Dimension(300,100));
		testo = new JTextArea();
		add(testo);
		testo.addKeyListener(new KeyListener()
		{

			public void keyPressed(KeyEvent arg0) {
				if(arg0.getKeyCode() == KeyEvent.VK_ENTER)
				{
					if(testo.getText().compareTo("") == 0)
						userContextInputStream.putInput(null);
```

```
                                else
                                userContextInputStream.putInput(new
ByteArrayInputStream(testo.getText().toString().getBytes()));
                                testo.setText(null);
                                setVisible(false);
                        }
                }
                @Override
                public void keyReleased(KeyEvent arg0) {}
                @Override
                public void keyTyped(KeyEvent arg0) {}
        });
    }

    @Override
    public void run() {
        initComponent();
        setVisible(true);
    }
}
```

CUIConsole Android

Per completezza si riportano anche le modifiche effettuate alla versione per piattaforma android.

Data l'entità del codice della classe in calce vengono riportati solo il costruttore della classe, in quanto ha subito modifiche, e il nuovo metodo di gestione ReadEvent introdotto.

Alla classe vengono aggiunti un campo Stream che viene inizializzato recuperando dal motore Prolog il riferimento alla IOLibrary caricata al momento dell'avvio.

Al costruttore della CUIConsole viene passato come argomento anche il contesto della tuPrologActivity (classe invocante) per avere un riferimento dal quale avviare il dialog per l'inserimento dell'input.

```
private UserContextInputStream userContextInputStream;
private alice.tuprologx.android.tuPrologActivity tuP;

public CUIConsole(TextView tv, AutoCompleteTextView et, Button btn,
    TextView sol, TextView out, Button next, Toast t, tuPrologActivity tu) {
  engine = new Prolog();

  engine.addWarningListener(this);
  engine.addOutputListener(this);
  engine.addSpyListener(this);
  engine.addExceptionListener(this);

  textView = tv;
  editText = et;
  button = btn;
  btnext = next;
  btnext.setEnabled(false);
  solution = sol;
```

```
    output = out;
    toast = t;
    tuP = tu;

    IOLibrary IO = (IOLibrary) engine.getLibrary("alice.tuprolog.lib.IOLibrary");
    IO.setInputType("graphic");
    userContextInputStream = IO.getUserContextInputStream();

    setReadEventListener();

    button.setOnClickListener(new OnClickListener() {
      public void onClick(View v)
      {
       userContextInputStream.setCounter();

        btnext.setEnabled(false);
        if (editText.getText().toString().equals("")) {
          toast.show();
        } else {
          ArrayAdapter<String> aa = new ArrayAdapter<String>(tuPrologActivity
              .getContext(), android.R.layout.simple_dropdown_item_1line,
              arrayList);
          if (!arrayList.contains(editText.getText().toString()))
            arrayList.add(editText.getText().toString());

          editText.setAdapter(aa);
          goalRequest();

          if ((engine.hasOpenAlternatives())) {
            btnext.setEnabled(true);
          }
        }
      }
    });

    btnext.setOnClickListener(new OnClickListener() {

      public void onClick(View v) {

        String choice = "";
        try {
          if ((info = engine.solveNext()) != null) {
            if (!info.isSuccess()) {
              solution.setText("no.\n");
              become("goalRequest");
            } else {
              choice = solveInfoToString(info) + "\n";
              solution.setText(choice);
            }
          }
        } catch (NoMoreSolutionException e) {

          Toast toast = Toast.makeText(tuPrologActivity.getContext(),
              "No more solutions", Toast.LENGTH_SHORT);
          toast.show();
          e.printStackTrace();
        }
      }

    });
  }
```

Il metodo setReadEventListener di seguito presentato è stato inserito per consentire alla CUIConsole di gestire l'evento lanciato dalla classe Stream.

Alla ricezione dell'evento, si procede alla creazione di un nuovo Thread che visualizzerà un AlertDialog tramite il quale l'utente potrà inserire il dato richiesto.

Alla chiusura del dialog il valore viene restituito al Thread principale sbloccandolo.

```
private void setReadEventListener(){

   userContextInputStream.setReadListener(new ReadListener(){

      public void readCalled(ReadEvent arg0) {

          new Thread(new Runnable(){

             public void run(){
                Looper.prepare();
                AlertDialog.Builder alert = new AlertDialog.Builder(tuP);
                alert.setTitle("Input Console");
                final EditText input = new EditText(tuP);
                alert.setView(input);

               alert.setPositiveButton("Ok", new DialogInterface.OnClickListener() {
                  public void onClick(DialogInterface dialog, int whichButton) {
                     if(input.getText().toString().compareTo("")==0)
                            userContextInputStream.putInput(null);
                     else
                            userContextInputStream.putInput(new
ByteArrayInputStream(input.getText()
                            .toString().getBytes()));
                                     }
                                   });
               alert.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
                   public void onClick(DialogInterface dialog, int whichButton) {
                      userContextInputStream.putInput(null);
                                     }
                                   });

               alert.show();
               Looper.loop();
               }
       }).start();

      }
   });
  }
```

Bibliografia

tuProlog
http://tuprolog.alice.unibo.it
http://tuprolog.googlecode.com

Eclipse + SDK
http://www.eclipse.org

Android APIs
http://www.android.com
http://developers.android.com

Tesi: Porting e testing di un interprete tuProlog su piattaforma Android
Pasquini Andrea, Anno Accademico 2010/11

Tesi: Reingegnerizzazione ed estensione dell'interprete tuProlog su piattaforma Android
Privitera Gianluca, Anno Accademico 2010/11