ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA, SCIENZE E INGEGNERIA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Fondamenti di Informatica T-2

Modularizzazione dell'interprete tuProlog su piattaforma Java 9

CANDIDATA
Maria Russo

RELATORE:
Chiar.mo Prof. Enrico Denti

Anno Accademico 2017/2018

Sessione II

Indice

Introduzione

L'aggiornamento di Java Platform Standard Edition alla versione 9 ha introdotto diverse novità nel campo dell'architettura software. In particolare, l'introduzione di un supporto per la produzione modulare attraverso il Java Platform Module System consente di migliorare, tramite l'utilizzo di moduli, il modello dell'applicazione strutturata su più pacchetti, favorendone così la scalabilità e la gestione a livello di sviluppo, efficienza, manutenzione e aggiornamento e permettendo di creare runtime personalizzate che comprendano solo i moduli necessari per ogni applicazione o dispositivo.

A tal proposito, obbiettivo della tesi è quello di realizzare una proposta di modularizzazione di tuProlog per la piattaforma Java.

tuProlog è un interprete Prolog scritto in Java, sviluppato dall'Università di Bologna, leggero ed open-source, disponibile per molte piattaforme e ambienti (Java, Android, Microsoft .NET, Eclipse, iOS) e integrabile con altri linguaggi, consentendo la programmazione molti-paradigma e multi-linguaggio.

In particolare, la tesi è strutturata in tre capitoli.
Nel primo capitolo viene esposto brevemente l'interprete tuProlog, la sua struttura e le caratteristiche.
Nel secondo capitolo vengono descritti nel dettaglio i moduli in Java 9 e come costruirli, compilarli ed eseguirli. Inoltre, si introducono gli strumenti `jdeps`, `jmod` e `jlink`.
Il terzo ed ultimo capitolo è suddivido in quattro fasi: la prima di analisi delle dipendenze all'interno del progetto tuProlog per Java e delle applicazioni sviluppate per le altre piattaforme; la seconda riguardante la scelta dell'IDE da utilizzare per realizzare la proposta di modularizzazione; la terza, nella quale vengono esposti i moduli che si è deciso di creare; la quarta, in cui viene descritta la proposta finale di modularizzazione e la creazione dei file JAR.

1. tuProlog

tuProlog è un framework Prolog leggero ed open-source per applicazioni ed infrastrutture distribuite, rilasciato sotto la licenza LGPL (GNU Lesser General Public License).
È stato sviluppato presso l'Università di Bologna ed è scaricabile al link `https://bitbucket.org/tuprologteam/tuprolog/downloads/`, tramite Bitbucket repository.

È disponibile per molte piattaforme e ambienti, come: JavaSE, Eclipse plugin e Android, i quali condividono lo stesso *core* e le stesse librerie; Microsoft .NET, il quale rappresenta una libreria apposita che estende l'approccio multi-paradigma a qualsiasi linguaggio disponibile sulla piattaforma .NET.
La piattaforma di riferimento principale, utilizzata per sviluppare tuProlog, è Java.
tuProlog è inoltre interoperabile sia con i modelli (*pattern*) standard di Internet (come TCP/IP, RMI, CORBA) che con i modelli e i linguaggi di coordinazione.

Per poter gestire tutte le piattaforme in modo omogeneo si ricorre ad uno schema di numerazione delle versioni di tuProlog: i primi due numeri rappresentano la versione del motore e, finché non cambiano, un'applicazione tuProlog è garantita a comportarsi in modo identico a qualsiasi piattaforma supportata; il terzo ed ultimo numero indica la specifica della piattaforma riguardante aggiornamenti minori (grafica, correzioni di problemi o bug specifici della piattaforma, etc.) che non hanno alcun impatto sul motore Prolog.

Le principali caratteristiche dell'interprete tuProlog sono:

- minimalità: il core contiene solo gli elementi utili al motore Prolog; mentre qualsiasi altra funzionalità è implementata all'interno delle librerie.
- configurazione dinamica: il sistema Prolog può essere personalizzato per soddisfare le esigenze dell'utente.
- integrazione diretta con Java e .NET: permette di supportare nativamente la programmazione multi-paradigma e multi-linguaggio.
- facilità di *deploy*: la procedura di installazione è semplice ed i requisiti minimali. Un'installazione basata su Java richiede unicamente una *Java*

Virtual Machine (*JVM*) adatta ed il singolo file JAR da eseguire. Se necessario, possono essere aggiunti successivamente altri componenti. Un processo di installazione simile è seguito dalla piattaforma .NET. L'installazione Android è invece eseguita in modo conforme ai requisiti Android. Allo stesso modo, l'installazione del plugin di Eclipse deve essere conforme ai requisiti di Eclipse plugin manager. In seguito all'installazione, viene creato un sito di aggiornamento dove il plugin di tuProlog è disponibile come funzionalità di Eclipse.

1.1. Struttura tuProlog

Ciascun motore tuProlog fornisce quattro livelli di configurabilità:

- librerie: forniscono un set di predicati, funtori e una *theory* correlata che consente di definire nuovi *flag* e operatori. Alcune librerie sono incluse nella distribuzione tuProlog e sono caricate di default nella sua configurazione standard; tuttavia, gli utenti possono facilmente sviluppare le proprie librerie in diversi modi. Queste ultime, infatti, possono essere caricate, per mezzo del predicato `load_library` (parte Prolog) o del metodo `loadLibrary` del motore tuProlog (parte Java o .NET), e scaricate per estendere dinamicamente un motore tuProlog.
- direttive: impartite tramite il predicato `:-/1`, nativamente supportato dal motore, possono essere impiegate per configurare e utilizzare un motore tuProlog (`set_prolog_flag/1`, `load_library/1`, `include/1`, `solve/1`), il formato e la sintassi dei *read-term* (termini Prolog definiti dallo standard ISO).
- flag: definiti dinamicamente, descrivono gli aspetti principali di librerie, predicati e funtori valutabili. Un flag è identificato da un nome, una lista di possibili valori, un valore di default e uno booleano, che indica se il valore del flag può essere cambiato. Un valore di flag, qualora fosse modificabile, può essere sostituito dinamicamente con uno nuovo incluso nella lista dei possibili valori.
- theory: possono essere caricate o scaricate per mezzo di appropriati predicati *library*. Una theory è un testo costituito da una sequenza di clausole e/o

direttive. Le clausole e le direttive sono terminate da un punto e separate da uno spazio.

In tuProlog sono presenti tre diversi tipi di predicati:

- predicati *built-in*: definiti a livello core di tuProlog, costituendo una parte essenziale del motore, non subiscono alcun effetto in seguito a modifiche fatte sul motore prima o durante l'esecuzione.
- predicati library: caricati in un motore tuProlog attraverso una libreria tuProlog possono variare da motore a motore e nello stesso motore nel tempo, poiché le librerie possono essere caricate e scaricate nei motori Prolog sia staticamente all'avvio del sistema che dinamicamente durante l'esecuzione. Possono essere sovrascritti dai predicati *theory* e per rimuovere un predicato library dal motore è necessario eliminare l'intera libreria.
- predicati theory: caricati in un motore tuProlog attraverso una theory tuProlog, proprio come i predicati library, possono cambiare da motore a motore e nello stesso motore nel tempo, in quanto le theory possono essere caricate e scaricate sia staticamente che dinamicamente.

1.2. Programmazione multi-paradigma e multi-linguaggio

La programmazione multi-paradigma e quella multi-linguaggio sono caratteristiche fondamentali di tuProlog poiché consentono agli utenti di:

- utilizzare qualsiasi classe, libreria e oggetto Java direttamente dal codice Prolog, lasciando i due linguaggi e i modelli computazionali totalmente separati in modo da preservare la loro semantica ed estendendo, pertanto, la piattaforma *object-oriented* a Prolog.
- usare qualsiasi motore Prolog direttamente dal codice Java o .NET, in un modo semplice e non intrusivo dal punto di vista semantico, rendendo possibile la programmazione logica nelle applicazioni Java o .NET.
- estendere Prolog, in modo semplice, definendo nuove librerie in Prolog e/o nel linguaggio object-oriented della piattaforma selezionata.
- estendere Java facilmente definendo nuovi metodi Java in Prolog (framework "P@J") (vedi *tuProlog Manual* [1], p. 8, cap. 1).

1.3. tuProlog in Java

La *release* di tuProlog per la piattaforma Java è rappresentata da un singolo file zip con all'interno i file specifici del progetto: `src`, che include tutti i file sorgente; `lib`, contenente le librerie usate dal progetto tuProlog; `doc`, che contiene la documentazione; `test`, che include i file sorgente dei test (*FIT test* e *JUnit test*); `ant`, che contiene alcuni script Ant per automatizzare il *build* di parti del progetto tuProlog; etc. Inoltre, è presente anche una versione ridotta priva della cartella `src`.

Sono anche presenti due file JAR:

- `tuprolog.jar`: contiene il core di tuProlog; cioè solo la parte che è necessario includere in un progetto Java per potere accedere alle classi tuProlog e scrivere applicazioni multi-paradigma Java o tuProlog.
- `2p.jar`: file JAR eseguibile che lancia l'IDE tuProlog e che contiene tutti gli elementi essenziali per il suo utilizzo (core API, Agent application, librerie, GUI, etc.).

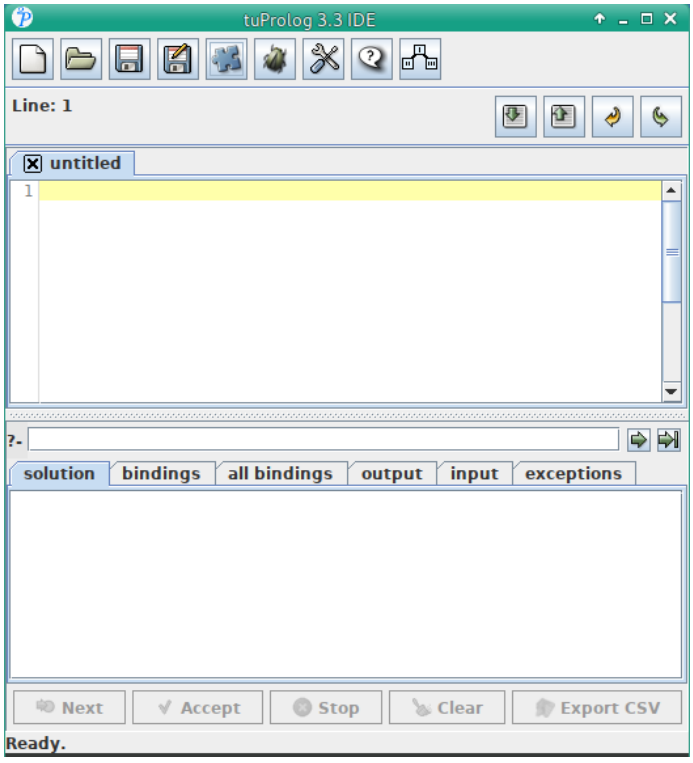

Figura 1.1 : tuProlog IDE

2. Java Platform Standard Edition 9

L'infrastruttura Java col passare degli anni è cresciuta molto ed è sempre in continua espansione. Essa è composta da:

- *Java Runtime Environment* (*JRE*): ambiente di esecuzione *runtime* che fornisce le librerie, la Java Virtual Machine e altri componenti necessari per l'esecuzione di applet e applicazioni scritte nel linguaggio di programmazione Java. Può essere anche ridistribuito con le applicazioni per renderle indipendenti.
- *Java Development Kit* (*JDK*): include il JRE e gli strumenti di sviluppo da riga di comando, come compilatori e *debuggers* necessari per lo sviluppo di applet e applicazioni.

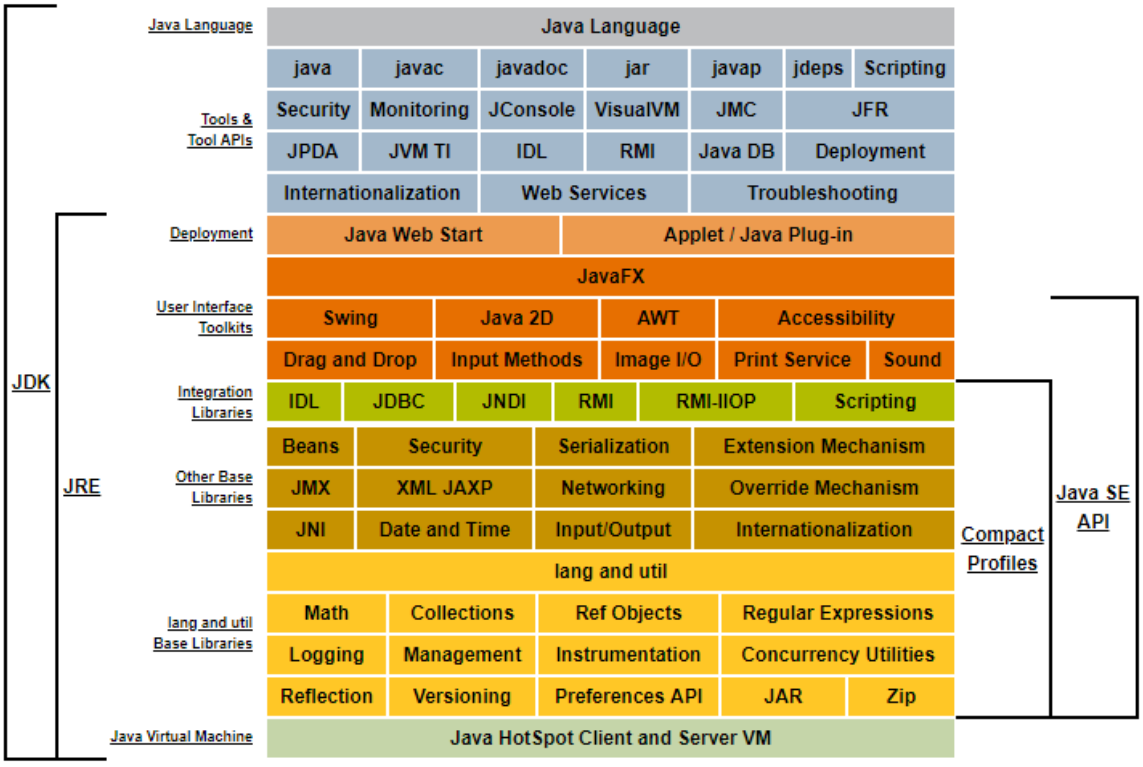

Figura 2.1 : Struttura Java SE (tratta da [7])

Il 21 Settembre 2017 è stato lanciato Java 9, il quale introduce importanti aggiornamenti e migliorie nelle librerie e nel linguaggio, specialmente nel campo dell'architettura software.
Una delle principali novità è rappresentata dall'introduzione di un supporto per la produzione modulare attraverso il *Java Platform Module System*, risultato del progetto Jigsaw.

2.1. Java Platform Module System

Mentre in precedenza, la piattaforma Java era costituita da una grande quantità di pacchetti (*package*), la cui gestione risultava difficile a livello di sviluppo, manutenzione e aggiornamento; Java Platform Module System permette di creare runtime personalizzate che comprendono solo i moduli necessari per ogni applicazione o dispositivo.

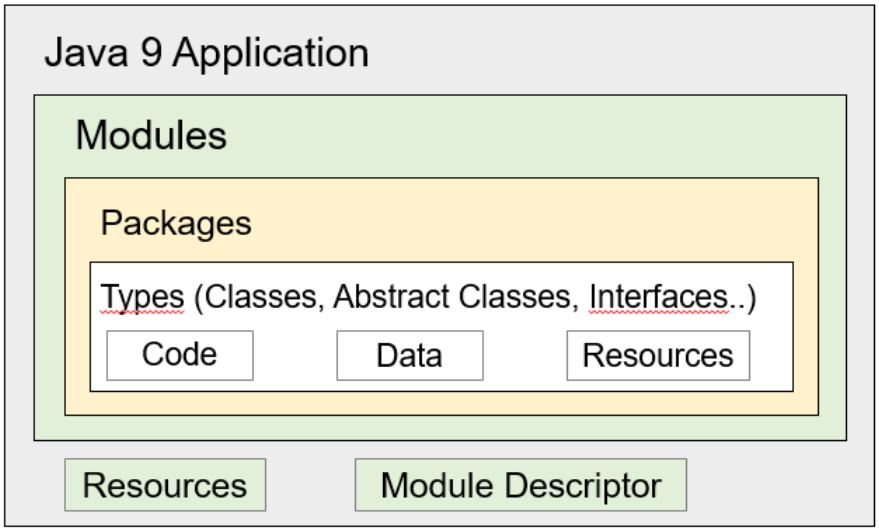

Figura 2.2 : Struttura Applicazione Java 9

In particolare, un aspetto fondamentale è stato la divisione del JDK in un set di moduli, a supporto di varie configurazioni, consentendo una maggiore comprensione delle loro dipendenze da parte degli utenti; il tutto apportando miglioramenti sulla scalabilità di Java, con la possibilità di adattarlo anche ai device di minore dimensione senza compromettere stabilità, performance e sicurezza.

Gli sviluppatori hanno così l'opportunità di realizzare e gestire contemporaneamente, in modo molto più efficace e senza perdite di tempo, librerie ed applicazioni complesse destinate sia alla *Java Standard Edition* (*Java SE*) che alla *Java Enterprise Edition* (*Java EE*).

Java 9 Platform Module System mette, pertanto, a disposizione dei programmatori novantacinque moduli all'interno del JDK, numero che potrebbe cambiare con l'evolversi di Java.

Eseguendo il comando `java --list-modules` all'interno della cartella *bin* del JDK, si ottiene, elencato, il set di moduli del JDK, il quale include:

- i moduli standard che implementano la specifica SE del linguaggio Java, i cui nomi cominciano con "`java.*`";
- i moduli JavaFX, i cui nomi iniziano con "`javafx.*`";
- i moduli specifici del JDK, i cui nomi cominciano con "`jdk.*`";
- i moduli specifici di Oracle, i cui nomi iniziano con "`oracle.*`".

Come è possibile vedere dall'output mostrato di seguito, ciascun nome di modulo è seguito da una stringa di versione `—@numeroVersione` che indica la versione di Java a cui appartiene il modulo (ad esempio, `—@9` denota che il modulo appartiene a Java 9).

```
java.activation@10.0.2
java.base@10.0.2
java.compiler@10.0.2
java.corba@10.0.2
java.datatransfer@10.0.2
java.desktop@10.0.2
java.instrument@10.0.2
java.jnlp@10.0.2
java.logging@10.0.2
java.management@10.0.2
java.management.rmi@10.0.2
java.naming@10.0.2
java.prefs@10.0.2
java.rmi@10.0.2
java.scripting@10.0.2
java.se@10.0.2
java.se.ee@10.0.2
java.security.jgss@10.0.2
java.security.sasl@10.0.2
java.smartcardio@10.0.2
java.sql@10.0.2
```

```
java.sql.rowset@10.0.2
java.transaction@10.0.2
java.xml@10.0.2
java.xml.bind@10.0.2
java.xml.crypto@10.0.2
java.xml.ws@10.0.2
java.xml.ws.annotation@10.0.2
javafx.base@10.0.2
javafx.controls@10.0.2
javafx.deploy@10.0.2
javafx.fxml@10.0.2
javafx.graphics@10.0.2
javafx.media@10.0.2
javafx.swing@10.0.2
javafx.web@10.0.2
jdk.accessibility@10.0.2
jdk.charsets@10.0.2
jdk.crypto.cryptoki@10.0.2
jdk.crypto.ec@10.0.2
jdk.crypto.mscapi@10.0.2
jdk.deploy@10.0.2
jdk.deploy.controlpanel@10.0.2
jdk.dynalink@10.0.2
jdk.httpserver@10.0.2
jdk.incubator.httpclient@10.0.2
jdk.internal.ed@10.0.2
jdk.internal.le@10.0.2
jdk.internal.vm.ci@10.0.2
jdk.internal.vm.compiler@10.0.2
jdk.internal.vm.compiler.management@10.0.2
jdk.javaws@10.0.2
jdk.jdwp.agent@10.0.2
jdk.jfr@10.0.2
jdk.jsobject@10.0.2
jdk.localedata@10.0.2
jdk.management@10.0.2
jdk.management.agent@10.0.2
jdk.management.cmm@10.0.2
jdk.management.jfr@10.0.2
jdk.management.resource@10.0.2
jdk.naming.dns@10.0.2
jdk.naming.rmi@10.0.2
jdk.net@10.0.2
jdk.pack@10.0.2
jdk.plugin@10.0.2
jdk.plugin.server@10.0.2
jdk.scripting.nashorn@10.0.2
jdk.scripting.nashorn.shell@10.0.2
jdk.sctp@10.0.2
jdk.security.auth@10.0.2
jdk.security.jgss@10.0.2
jdk.snmp@10.0.2
jdk.unsupported@10.0.2
jdk.xml.dom@10.0.2
```

```
jdk.zipfs@10.0.2
oracle.desktop@10.0.2
oracle.net@10.0.2
```

Il modulo `java.base` appartenente a Java 9 Platform Module System è indipendente; cioè non dipende da nessun altro modulo, ma di default tutti gli altri moduli dipendono da questo.

Con l'uscita di Java 9, viene introdotto, inoltre, un nuovo concetto in aggiunta al già esistente *class path*: il *module path*. In questo caso i JAR menzionati all'interno del module path sono trattati come moduli anziché come semplici JAR.

2.1.1. Modulo

Un modulo è costituito da un *Module Descriptor* e da un insieme di package correlati e risorse. In un'applicazione non sono ammessi package duplicati; pertanto un package può appartenere ad un solo modulo. È vero quindi che, per poter modularizzare un'applicazione, quest'ultima deve essere strutturata in più package; in caso contrario è necessario un *refactoring* dell'applicazione.

Il Module Descriptor contiene i metadati che specificano il nome del modulo, le sue dipendenze, i package che rende esplicitamente disponibili agli altri moduli e altro. È memorizzato in un file chiamato `module-info.class` all'interno della cartella root del modulo e viene creato quando si compila il *Module Declaration*, definito in un file chiamato `module-info.java`.
Ciascun Module Declaration inizia con la *keyword* `module` seguita dal nome univoco del modulo e dal corpo del modulo racchiuso tra parentesi graffe:

```
module moduleName {
}
```

Il corpo del Module Declaration può essere vuoto o può contenere varie *Module Directives*.

Importante per le direttive dei moduli è la *reflection*: di default, un modulo con accesso riflessivo a runtime a un package può vedere tutti i tipi `public` del pacchetto e i loro tipi annidati `public` e `protected`. Tuttavia, come accade nelle versioni precedenti di Java, la reflection può essere utilizzata per conoscere tutti i tipi in un package esposto e tutti gli elementi all'interno di questi tipi, inclusi gli elementi `private`, attraverso il metodo `setAccessible`, indipendentemente dal fatto che si voglia consentire o meno tale funzionalità. Pertanto, non vi è un vero incapsulamento a differenza del sistema modulare, dove il forte incapsulamento (*strong encapsulation*) rappresenta invece un risultato importante, in quanto migliora la sicurezza della piattaforma poiché meno classi sono accessibili a potenziali aggressori.
Di default, quindi, un tipo in un modulo non è accessibile ad altri moduli a meno che non sia di tipo `public` e non si esporti il package. Si rendono disponibili, perciò, solo i pacchetti che si vuole esporre e, con Java 9, ciò è applicato anche alla reflection.

Alcune *Module Directives* sono:

- **`exports:`** questa direttiva indica quali package sono accessibili dall'esterno. Pertanto, specifica uno o più package appartenenti al modulo corrente i cui tipi `public` e i loro tipi annidati `public` e `protected` devono essere accessibili al codice in tutti gli altri moduli.
  È possibile, inoltre, specificare selettivamente, tramite la direttiva **`exports…to`**, in un elenco separato da virgole, i moduli che possono accedere al package esportato. Questa operazione è conosciuta come *qualified export*.

```
module moduleName
{
   exports packageName1, ...;
   exports packageName to moduleName1, ...; //in caso di qualified export
}
```

La Java Virtual Machine (JVM) supporta nativamente il concetto di modulo: l'accesso a package non esplicitamente esportati viene impedito (*strong encapsulation*) in modo retrocompatibile.

- **`requires:`** questa direttiva specifica una relazione di dipendenza da altri moduli, chiamata *module dependency*. Indica, quindi, da quali altri moduli il modulo corrente dipende. Ciascun modulo deve esplicitamente dichiarare le sue dipendenze.
  È possibile specificare una dipendenza transitiva, tramite **`requires transitive`**, al fine di indicare una dipendenza da un altro modulo e di garantire che gli altri moduli, che leggono quello corrente, leggano anche questa dipendenza, conosciuta come *implied readability*.
  Si può, inoltre, utilizzare la direttiva **`requires static`** nel caso in cui il modulo sia richiesto a *compile time* ma non necessariamente a runtime. Questa tipologia di dipendenza opzionale è conosciuta come *optional dependency*.

```
module moduleName
{
   requires moduleName1, ...;           //in caso di module dependency
   requires transitive moduleName;      //in caso di implied readability
   requires static moduleName;          //in caso di optional dependency
}
```

  I moduli standard di Java SE devono permettere in tutti i casi implied readability, tranne che ai moduli non-standard, sebbene potrebbero dipendere da essi. Ciò garantisce che il codice, dipendente solo dai moduli standard di Java SE, sia portabile attraverso le implementazioni Java SE.

- **`opens:`** questa direttiva apre un package alla reflection permettendo, in modo selettivo, l'accesso ai campi non pubblici di una classe di un certo modulo, cosa che normalmente viene impedita. Indica, quindi, che i tipi `public` di uno specifico pacchetto e i loro tipi annidati `public` e `protected` sono accessibili al codice in altri moduli solo in fase di runtime.
  Si può anche determinare, tramite la direttiva **`opens…to`**, che i tipi `public` del pacchetto specificato e i loro tipi annidati `public` e `protected` siano accessibili, attraverso la reflection, al codice nei moduli dichiarati in un elenco separato da virgole solo in fase di runtime.

```
module moduleName
{
   opens packageName;
   opens packageName to moduleName1, ...;     //in caso di opens selettiva
}
```

Inoltre, attraverso la keyword **`open`**, è possibile stabilire che tutti i pacchetti in un dato modulo siano accessibili, a runtime e tramite la reflection, a tutti gli altri moduli.

```
open module moduleName
{
   //Module Directives
}
```

- **`uses:`** questa direttiva specifica un servizio usato dal modulo corrente, rendendo quest'ultimo un consumatore del servizio (*service consumer*). Un servizio (*service*) è un oggetto di una classe che implementa l'interfaccia o estende la classe astratta specificata nella direttiva `uses`.

```
module moduleName
{
   uses serviceInterfaceORAbstractClass;
}
```

- **`provides…with:`** questa direttiva indica che un modulo fornisce un'implementazione del servizio, facendo diventare il modulo un fornitore del servizio (*service provider*). La parte `provides` della direttiva specifica un'interfaccia o una classe astratta elencata nella direttiva `uses` del modulo; mentre la parte `with` della direttiva indica il nome della classe del service provider che implementa l'interfaccia o estende la classe astratta.

```
module moduleName
{
   provides serviceInterfaceORAbstractClass with serviceProviderClass;
}
```

2.1.2. Denominazione di un modulo

Il nome di un modulo è univoco, come lo sono i nomi dei package.
Tipicamente, per stabilire il nome di un package si prende il nome del dominio Internet dell'organizzazione e lo si scrive in ordine inverso. Ad esempio, se il nome del dominio è `name.com`, allora il nome del package sarà `com.name`.
Similarmente, i moduli utilizzano la convenzione *reverse-domain-name* per stabilire i propri nomi; pertanto, il nome di un modulo è univoco ed è strutturato a livelli (*`identifier.indentifier.indentifier`*).
Nel caso (errato) in cui più moduli abbiano lo stesso nome, a compile time si ottiene un errore di compilazione.
Solitamente il nome di uno modulo corrisponde al nome del più importante package al suo interno. Ad esempio, il modulo `java.sql` del JDK contiene il package `java.sql`. Ciò è possibile in quanto Java mantiene separati i nomi dei moduli dai nomi dei package.

2.1.3. Modulo di default e Modulo automatico

Non è necessario modularizzare un'applicazione affinché questa funzioni in Java 9; in quanto per retrocompatibilità esiste un modulo di default (*unnamed module*) nel quale vengono inseriti tutti i package non appartenenti ad un modulo esplicito.
Il modulo di default può accedere ad ogni altro modulo ed esporta tutti i sui package (`exports all`, `requires all`); non è, però, accessibile dagli altri moduli con nome, come accade per il al package di default.

È disponibile, inoltre, un efficace supporto alla migrazione di applicazioni già esistenti rappresentato dai *moduli automatici*: ogni JAR all'interno del module path è convertito automaticamente in un modulo con all'incirca lo stesso nome, che richiede tutto ed esporta tutto.
Più in generale, i JAR non modulari possono essere convertiti in JAR modulari. Innanzitutto, bisogna estrarre dal JAR tutte le dipendenze; dopodiché, si deve creare l'apposito file `module-info.java` e compilarlo con l'opzione `--patch-module`. Infine, tramite il comando `jar --update`, bisogna aggiornare il JAR aggiungendo la nuova dichiarazione di modulo.

2.1.4. Denominazione di un modulo automatico: Algoritmo di naming

Un modulo automatico si chiama all'incirca come il JAR ma con alcuni accorgimenti per garantire che il nome ottenuto sia un valido nome di package secondo la sintassi Java multi-livello e non sia legato a specifiche versioni; in tal caso esiste l'attributo *version*.

Il funzionamento dell'algoritmo di *naming* è il seguente: se il nome contiene un trattino seguito da caratteri numerici e quindi combacia con l'espressione regolare "`-(\\d+(\\.|$))`", allora il nome del modulo è rappresentato solo dalla parte che precede il trattino; mentre la parte successiva costituisce, se possibile, il numero di versione, altrimenti viene ignorata. In seguito, ciascun carattere non alfanumerico è sostituito da un punto e i gruppi di punti vengono collassati in un unico punto; dopodiché tutti i punti iniziali e finali vengono rimossi.
Nel caso in cui il nome del JAR cominci con caratteri numerici e quindi non sia un identificatore Java valido, dato che l'algoritmo di conversione elimina i numeri nei nomi dei JAR e quindi non è possibile estrarne un modulo automatico, alcune possibili soluzioni sono:

- cambiare nome al JAR, se si è utente;
- aggiungere al MANIFEST la riga `Automatic-Module-Name: `*`name`*, se si è utente evoluto o creatore del JAR;
- aggiungere al package un vero `module-info.java`, se si è il creatore del JAR.

2.1.5. Compilazione di un modulo

Per compilare un modulo di un'applicazione è necessario eseguire, posizionandosi all'interno della cartella dell'applicazione, il seguente comando:

```
javac -d mods/compiledModuleName src/moduleName/module-info.java
mainPath/main.java
```

L'opzione `-d` indica che `javac` posizionerà il codice compilato in una cartella specifica *`mods`* che conterrà una sottocartella *`compiledModuleName`* rappresentante il modulo compilato. Convenzionalmente, si utilizza il nome *`mods`* per indicare la cartella che contiene tutti i moduli compilati. Questa cartella è detta *exploded-module folder*, in quanto le cartelle e i file `.class`, situati al suo interno, non si trovano in un file JAR.

È possibile ottenere informazioni riguardanti uno specifico modulo compilato attraverso l'opzione `--describe-module` del comando `java` eseguito all'interno della cartella dell'applicazione.

```
java --module-path mods --describe-module moduleName
```

Per compilare automaticamente tutti i moduli si può utilizzare l'opzione `--module-source-path` del comando `javac`, indicando la cartella contenente tutti i moduli.

```
javac -d mods --module-source-path src mainPath/main.java
```

2.1.6. Esecuzione di un'applicazione modulare

Per eseguire un'applicazione modulare in Java 9 si può usufruire di due metodi distinti, da utilizzare all'interno della cartella dell'applicazione tramite il comando `java`:

- forma *`--optiontext`*: si utilizzano le opzioni `--module-path`, per indicare il module path (*`mods`*), e `--module`, per specificare il nome del modulo ed il nome completo della classe contenente il *`main`*.

```
java --module-path mods --module moduleName/completeClassMainName
```

- forma *`-option`*: si usano le specifiche `-p` e `-m` rispettivamente per sostituire `--module-path` e `--module`.

```
java -p mods -m moduleName/completeClassMainName
```

2.1.7. Trasformazione di un modulo in un file JAR modulare

Per trasformare una exploded-module folder in un file JAR modulare contenente tutti i file del modulo, incluso il file `module-info.class` posizionato nella cartella *root* del JAR, si utilizza il comando `jar`. La cartella che include tutti i file JAR creati deve esistere prima di eseguire tale comando.

```
jar --create -f jars/moduleName.jar -C mods/moduleName .
```

Se il modulo include un *entry-point* dell'applicazione, cioè una classe contenente il metodo *`main`*, allora si può specificare tale classe con l'opzione `--main-class` del comando `jar`.

```
jar --create -f jars/moduleName.jar --main-class completeClassMainName -C
mods/moduleName .
```

In particolare:

- `--create`: specifica che il comando `jar` potrebbe creare un nuovo file JAR;
- `-f`: indica il nome della cartella dove si trovano tutti i file JAR dei moduli, seguito dal nome del file JAR;
- `--main-class`: indica il nome completo dell'entry-point dell'applicazione;
- `-C`: specifica la cartella contenente i file da includere nel file JAR ed i nomi dei file da inserire. Si utilizza "`.`" per indicare che devono essere inseriti tutti i file all'interno della cartella.

2.1.8. Esecuzione di un'applicazione con file JAR modulari

Per eseguire un'applicazione, dove tutti i moduli sono stati trasformati in file JAR modulari:

- se è stato specificato un entry-point al momento della creazione del file JAR, si può utilizzare il seguente comando.

```
/* forma --otpion-text */
java --module-path jars moduleMainName

/* forma -option */
java -p jars -m moduleMainName
```

- se non è stato specificato un entry-point, è invece necessario indicare il nome del modulo seguito dal nome completo della classe contenente il *`main`*.

```
/* forma --otpion-text */
java --module-path jars --module moduleName/completeClassMainName

/* forma -option */
java -p jars -m moduleName/completeClassMainName
```

2.1.9. Strumento `jdeps`

Lo strumento `jdeps` [6] permette di analizzare le dipendenze di un file `.class`, di una directory o di un file di tipo JAR, indicando il loro path.

```
jdeps [options] path ...
```

Di default, il comando `jdeps` stampa le dipendenze sull'output di sistema.

Alcune delle opzioni che determinano l'output sono:

- `-summary`: consente di ottenere un output sintetico delle dipendenze. Può essere scritto anche nella sua forma abbreviata `-s`.

- `-verbose`: stampa tutti i livelli di dipendenza di un file `.class`. Esiste una forma abbreviata `-v` e due varianti:
  - `-verbose:package`, che stampa tutti i livelli di dipendenza di un package, escludendo di default le dipendenze con lo stesso package;
  - `-verbose:class`, che stampa tutti i livelli di dipendenza di un file `.class`, escludendo di default le dipendenze con lo stesso archivio.

  L'opzione `-v` è equivalente a:

```
-verbose:class -filter:none
```

- `--classpath`: specifica dove trovare i file `.class`. È possibile utilizzare anche la forma abbreviata `-cp` o `-classpath`.

```
--classpath path
```

- `--module-path`: indica il module path.

```
--module-path modulePath
```

- `--jdk-internals`: permette di conoscere i livelli di dipendenza di un file `.class` nelle API interne del JDK. Questa opzione può anche essere scritta nella forma `-jdkinternals`.

2.1.10. Strumento `jmod`

Lo strumento `jmod` [7] consente di creare un file di tipo JMOD o di ottenere informazioni da quest'ultimo.

Il formato JMOD è un formato di archivio, molto simile a quello JAR, ma con la possibilità di includere codice nativo e file di configurazione.

```
jmod (create|extract|list|describe|hash) [options] jmodFile
```

In particolare, *jmodFile* indica il file JMOD con cui si desidera interagire attraverso alcune operazioni del comando `jmod`:

- `create`, che crea un nuovo file di archivio JMOD;
- `extract`, che estrae tutti i file all'interno di un file ti tipo JMOD;
- `list`, che stampa i nomi di tutti gli elementi all'interno del file JMOD;
- `describe`, che stampa i dettagli di un modulo;
- `hash`, che determina i moduli *foglia* e registra gli *hash* delle dipendenze che li richiedono direttamente e indirettamente.

Alcune delle opzioni che offre il comando `jmod` sono:

- `--class-path`: specifica il percorso della cartella contenente i file JAR dell'applicazione o una cartella contenente le classi da copiare all'interno di *jmodFile*.

```
--class-path path
```

- `--exclude`: esclude i file corrispondenti ai pattern indicati in un elenco separato da virgole.

```
--exclude pattern-list
```

  Ogni elemento della lista di pattern può essere scritto in una delle tre seguenti forme:
  - *glob-pattern*;
  - `glob:`*glob-pattern*;
  - `regex:`*regex-pattern*, dove *regex-pattern* rappresenta una espressione regolare.

- `--hash-modules`: svolge la stessa funzione dell'operazione `hash` del comando `jmod`, basandosi sul modello grafico dei moduli corrispondenti al dato *`regex-pattern`*. Gli hash vengono registrati nel file JMOD che si desidera creare (*`jmodFile`*) o in un archivio, JMOD o JAR modulare, situato all'interno del module path indicato.

```
--hash-modules regex-pattern
```

- `--module-path`: specifica il module path. Questa opzione è richiesta quando si utilizza l'opzione `--hash-modules`.

```
/*forma --option-text */
--module-path path

/* forma -option */
-p path
```

- `--main-class`: indica la classe, contenente il metodo *`main`*, da registrare all'interno del file `module-info.class`.

```
--main-class className
```

2.1.11. Strumento `jlink`

Lo strumento `jlink` [8] permette in fase di *linktime*, fase intermedia tra compile time e runtime, di assemblare e ottimizzare un set di moduli e le loro dipendenze in una *runtime image* personalizzata che include solo le librerie della piattaforma Java necessarie all'applicazione.
Questo strumento è utile per risolvere il problema della pesantezza del JRE in applicazioni *embedded*; infatti consente di generare una versione ridotta e più leggera del JRE, contenente solo le classe e i JAR effettivamente utilizzati.
Quindi `jlink` genera applicazioni embedded analizzando quali classi dell'applicazione vengono realmente utilizzate e generando la runtime image

corrispondente: costruisce il minimo insieme di moduli necessari all'applicazione perché funzioni. L'applicazione così ottenuta può essere distribuita ed eseguita autonomamente, in quanto non è richiesta l'installazione del JRE.

Questo strumento non funziona con JAR non modularizzati, ma solo con JAR modulari, file JMOD o exploded-module folders.

```
jlink [options] --module-path modulepath --add-modules module [,module…]
```

In particolare:

- `--module-path`: indica il percorso della cartella dove si trovano i moduli che si vuole includere nella *runtime image*. Questa opzione può anche essere utilizzata nella sua forma abbreviata `-m`.

- `--add-modules`: specifica i nomi del modulo o dei moduli che si desidera aggiungere alla *runtime image* insieme alle loro dipendenze transitive.

Alcune delle altre opzioni del comando `jlink` disponibili sono:

- `--launcher`: specifica il nome del comando di lancio per il modulo, seguito o meno dal nome della classe contenente il metodo *`main`*.

```
/* prima forma */
--launcher command=moduleMainName

/* seconda forma */
--launcher command=moduleMainName/completeMainName
```

- `--limit-modules`: limita il comando `jlink` ai moduli indicati, al modulo contente il *`main`* e ai moduli specificati tramite l'opzione `--add-modules` se presente, escludendo i moduli aggiunti per dipendenza transitiva.

```
--limit-modules moduleName [,moduleName...]
```

- `--output`: specifica il percorso della cartella, non ancora esistente, in cui verrà generata la runtime image.

```
--output pathRuntimeImage
```

2.2. Java Platform Standard Edition 10

Dal mese di Settembre 2017, con l'uscita di Java SE 9, Oracle ha dichiarato di volere pubblicare un nuova release ogni sei mesi, con aggiornamenti trimestrali, ed una release a lungo termine ogni tre anni, a partire da Java SE 11 nel mese di Settembre 2018.

Il progetto di tesi è stato svolto utilizzando Java Platform Standard Edition 10, rilasciato il 20 Marzo 2018, il quale introduce alcune novità importanti, ma nessuna di rilevante inerente ai moduli.

3. Modularizzazione di tuProlog

In questo capitolo ci si pone l'obbiettivo di realizzare una soluzione di modularizzazione ottimale del progetto già esistente `2p` relativo all'interprete Prolog basato su Java e far in modo che chi ne usufruisce possa importare solo i moduli necessari al loro funzionamento.

3.1. Analisi delle dipendenze

Per progettare una proposta di modularizzazione di tuProlog è necessario effettuare un'analisi dettagliata di tutte le dipendenze all'interno del progetto `2p` e dei package appartenenti alla libreria `tuprolog.jar` utilizzati dalle applicazioni tuProlog sviluppate per le altre piattaforme (Android, Eclipse plugin, .NET, iOS).

Per analizzare le dipendenze, oltre ad uno studio accurato del progetto, è utile lo strumento `jdeps` offerto da Java Platform Standard Edition. Eseguendo il comando `jdeps` seguito dal nome del progetto, precedentemente compilato, è infatti possibile ottenere una lista dei package presenti all'interno della cartella `src` del progetto, indicanti tutti i pacchetti da cui ciascuno di questi dipende ed il modulo a cui questi ultimi appartengono.

Nei capitoli sottostanti sono mostrati all'interno di tabelle, per un risultato più chiaro e comprensibile, gli output relativi all'esecuzione del comando `jdeps` per ogni progetto, partendo dal progetto `2p` per Java e proseguendo con le applicazioni tuProlog sviluppate per le altre piattaforme.

3.1.1. Analisi del progetto `2p` per Java

Per la realizzazione della tesi e quindi per la fase di analisi è stata utilizzata la versione 3.3.0 del progetto tuProlog rilasciata il 22 Marzo 2018.

Nella tabella sottostante è possibile visualizzare nel dettaglio il risultato prodotto eseguendo il comando:

```
jdeps 2p
```

L'output ottenuto, mostrato di seguito all'interno di una tabella, mostra le relazioni tra i package di `2p`. In particolare, è possibile notare che:

- i package `alice.tuprologx.pj.annotations.parser`, `alice.util`, `alice.util.proxyGenerator`, `alice.tuprolog.lib.annotations`, `alice.util.jedit` e `alice.tuprologx.pj.annotations` sono "indipendenti", cioè possono essere utilizzati da altri package, ma non ne importano nessuno.
- i package delle coppie `alice.tuprolog` e `alice.tuprolog.event`, `alice.tuprolog` e `alice.tuprolog.interfaces`, `alice.tuprolog` e `alice.tuprolog.json`, `alice.tuprologx.pj.engine` e `alice.tuprologx.pj.meta`, `alice.tuprologx.pj.engine` e `alice.tuprologx.pj.model`, `alice.tuprolog.event` e `alice.tuprolog.lib` potrebbero causare cicli di dipendenza se inseriti in moduli separati, in quanto si richiedono l'un l'altro.
- i package `alice.tuprologx.runtime.rmi`, `alice.tuprologx.ide`, `alice.tuprologx.runtime.tcp`, `alice.tuprolog.scriptengine` e `alice.tuprologx.pj.annotations.processing` importano altri pacchetti ma non vengono richiesti da nessuno.

<table>
<tr><td colspan="3">2p -> java.base
2p -> java.compiler
2p -> java.datatransfer
2p -> java.desktop
2p -> java.rmi
2p -> java.scripting
2p -> not found</td></tr>
<tr><td rowspan="3">alice.tuprolog</td><td>-> alice.tuprolog.event
-> alice.tuprolog.interfaces
-> alice.tuprolog.json
-> alice.tuprolog.lib
-> alice.util</td><td>2p</td></tr>
<tr><td>-> cli.System.Reflection</td><td>not found</td></tr>
<tr><td>-> java.io
-> java.lang
-> java.lang.reflect</td><td>java.base</td></tr>
</table>

	-> java.net -> java.util -> java.util.concurrent.locks -> java.util.regex	
	-> junit.framework -> org.concordion.api .extension -> org.concordion.ext -> org.junit -> org.junit.runner -> org.junit.runners	not found
alice.tuprolog.event	-> alice.tuprolog -> alice.tuprolog.lib	2p
	-> java.lang -> java.util	java.base
alice.tuprolog .interfaces	-> alice.tuprolog -> alice.tuprolog.event	2p
	-> java.lang -> java.util	java.base
alice.tuprolog.json	-> alice.tuprolog	2p
	-> com.google.gson	not found
	-> java.lang -> java.lang.reflect -> java.util	java.base
alice.tuprolog.lib	-> alice.tuprolog -> alice.tuprolog.event -> alice.tuprolog.interfaces -> alice.tuprolog.lib .annotations -> alice.util -> alice.util.proxyGenerator	2p
	-> java.io -> java.lang -> java.lang.annotation -> java.lang.reflect -> java.net -> java.util -> java.util.concurrent -> java.util.regex	java.base
	-> org.apache.commons.lang3	not found
alice.tuprolog.lib .annotations	-> java.lang -> java.lang.annotation	java.base
alice.tuprolog .scriptengine	-> alice.tuprolog -> alice.tuprolog.event -> alice.tuprolog.lib -> alice.util	2p
	-> java.io -> java.lang -> java.util	java.base
	-> javax.script	java.scripting
alice.tuprologx.ide	-> alice.tuprolog -> alice.tuprolog.event -> alice.tuprolog.lib -> alice.tuprologx.spyframe -> alice.util -> alice.util.jedit	2p

	-> cli.System.Reflection	not found
	-> java.awt -> java.awt.event -> java.beans	java.desktop
	-> java.io -> java.lang -> java.net -> java.util	java.base
	-> javax.swing -> javax.swing.border -> javax.swing.event -> javax.swing.filechooser -> javax.swing.plaf -> javax.swing.table -> javax.swing.text -> javax.swing.tree -> javax.swing.undo	java.desktop
	-> org.fife.ui.autocomplete -> org.fife.ui.rsyntaxtextarea -> org.fife.ui.rtextarea	not found
alice.tuprologx.pj .annotations	-> java.lang -> java.lang.annotation	java.base
alice.tuprologx.pj .annotations.parser	-> java.io -> java.lang -> java.util	java.base
alice.tuprologx.pj .annotations.processing	-> alice.tuprolog -> alice.tuprologx.pj .annotations -> alice.tuprologx.pj .annotations.parser -> alice.tuprologx.pj.model	2p
	-> java.lang -> java.lang.annotation -> java.util	java.base
	-> javax.annotation.processing -> javax.lang.model -> javax.lang.model.element -> javax.lang.model.type -> javax.lang.model.util -> javax.tools	java.compiler
alice.tuprologx.pj .engine	-> alice.tuprolog -> alice.tuprolog.lib -> alice.tuprologx.pj .annotations -> alice.tuprologx.pj .annotations.parser -> alice.tuprologx.pj.lib -> alice.tuprologx.pj.meta -> alice.tuprologx.pj.model	2p
	-> java.io -> java.lang -> java.lang.annotation -> java.lang.ref -> java.lang.reflect -> java.security -> java.util	java.base

	-> javassist -> javassist.bytecode -> javassist.util.proxy	not found
alice.tuprologx.pj .lib	-> alice.tuprolog -> alice.tuprolog.lib -> alice.tuprologx.pj .annotations -> alice.tuprologx.pj.engine -> alice.tuprologx.pj.model -> alice.util	2p
	-> java.io -> java.lang -> java.lang.reflect -> java.util	java.base
alice.tuprologx.pj .meta	-> alice.tuprolog -> alice.tuprologx.pj .annotations -> alice.tuprologx.pj .annotations.parser -> alice.tuprologx.pj.engine -> alice.tuprologx.pj.model	2p
	-> java.io -> java.lang -> java.lang.annotation -> java.lang.reflect -> java.util	java.base
alice.tuprologx.pj.model	-> alice.tuprolog -> alice.tuprologx.pj .annotations -> alice.tuprologx.pj.engine	2p
	-> java.beans	java.desktop
	-> java.io -> java.lang -> java.lang.annotation -> java.lang.reflect -> java.util	java.base
alice.tuprologx .runtime.rmi	-> alice.tuprolog	2p
	-> java.io -> java.lang -> java.net	java.base
	-> java.rmi -> java.rmi.registry -> java.rmi.server	java.rmi
alice.tuprologx .runtime.tcp	-> alice.tuprolog	2p
	-> java.io -> java.lang -> java.lang.reflect -> java.net	java.base
alice.tuprologx.spyframe	-> alice.tuprolog -> alice.tuprolog.event	2p
	-> java.awt -> java.awt.event -> java.awt.font -> java.awt.geom	java.desktop
	-> java.io -> java.lang	java.base

	-> java.util	
	-> javax.swing	java.desktop
alice.util	-> cli.System.Reflection -> ikvm.runtime	not found
	-> java.io -> java.lang -> java.lang.reflect -> java.net -> java.security -> java.util	java.base
alice.util.jedit	-> java.awt	java.desktop
	-> java.awt.datatransfer	java.datatransfer
	-> java.awt.event	java.desktop
	-> java.io -> java.lang -> java.lang.reflect -> java.util	java.base
	-> javax.swing -> javax.swing.event -> javax.swing.text -> javax.swing.undo	java.desktop
alice.util .proxyGenerator	-> cli.System.Reflection -> ikvm.runtime	not found
	-> java.io -> java.lang -> java.lang.reflect -> java.net -> java.nio.charset -> java.util	java.base
	-> javax.tools	java.compiler

3.1.2. Analisi del progetto `2p-ios` per piattaforma iOS

Per la piattaforma iOS esiste il progetto `2p-ios` che importa la libreria `tuprolog.jar`. Per conoscere i pacchetti che questo progetto utilizza si può eseguire, all'interno della cartella in cui quest'ultimo è situato, il comando:

```
jdeps 2p-ios
```

Dall'output ottenuto, visualizzato nella tabella riportata di seguito, è possibile notare che il package `application` all'interno della cartella `src` dell'applicazione sviluppata per iOS utilizza unicamente il package `alice.tuprolog` appartenente alla libreria `tuprolog.jar`.

```
2p-ios -> java.base
```

2p-ios -> not found		
application	-> alice.tuprolog	not found
	-> java.lang -> java.util	java.base
	-> org.robovm.apple.coreanimation -> org.robovm.apple.coregraphics -> org.robovm.apple.foundation -> org.robovm.apple.uikit -> org.robovm.objc -> org.robovm.objc.annotation -> org.robovm.rt.bro.annotation	not found

3.1.3. Analisi del progetto `2p-net` per Microsoft .NET

Riguardo alla piattaforma .NET è presente solo una demo, `TicTacToeServer`, che importa la libreria `tuprolog.jar`. Per conoscere i pacchetti utilizzati da questo progetto è possibile eseguire, all'interno della cartella in cui si trova quest'ultimo, il comando:

```
jdeps TicTacToeServer
```

Dall'output ottenuto, visualizzato di seguito all'interno di una tabella, è possibile notare che l'applicazione sviluppata per Microsoft .NET utilizza solo i package `alice.tuprolog` e `alice.tuprolog.event` appartenenti alla libreria `tuprolog.jar`.

TicTacToeServer -> java.base TicTacToeServer -> java.rmi TicTacToeServer -> not found		
<unnamed>	-> alice.tuprolog -> alice.tuprolog.event	not found
	-> java.io -> java.lang -> java.net	java.base
	-> java.rmi -> java.rmi.registry -> java.rmi.server	java.rmi
	-> java.util	java.base

3.1.4. Analisi del progetto `2p-android` per piattaforma Android

Per la piattaforma Android esiste il progetto `2p-android` che importa la libreria `tuprolog.jar`. Per conoscere i pacchetti che questo progetto utilizza si può eseguire, all'interno della cartella in cui quest'ultimo è situato, il comando:

```
jdeps 2p-android
```

Dall'output ottenuto, visualizzato nella tabella riportata di seguito, è possibile notare che il package `alice.tuprologx.android.tuprologmobile` all'interno della cartella `src` dell'applicazione sviluppata per Android utilizza i package `alice.tuprolog`, `alice.tuprolog.event`, `alice.tuprolog.lib` e `alice.util` appartenenti alla libreria `tuprolog.jar`.

2p-android -> java.base 2p-android -> not found		
alice.tuprologx.android .tuprologmobile	-> <unnamed>	not found
	-> SharedPreferences	not found
	-> alice.tuprolog -> alice.tuprolog.event -> alice.tuprolog.lib	not found
	-> alice.tuprologx.android .tuprologmobile.models	2p-android
	-> alice.util	not found
	-> java.io -> java.lang -> java.net -> java.util	java.base
alice.tuprologx.android .tuprologmobile.adapters	-> <unnamed>	not found
	-> alice.tuprologx.android .tuprologmobile.models	2p-android
	-> java.lang -> java.util	java.base
alice.tuprologx.android .tuprologmobile.fragments	-> <unnamed>	not found
	-> alice.tuprologx.android .tuprologmobile	2p-android
	-> android.support.v4.app	not found
	-> java.lang	java.base
alice.tuprologx.android .tuprologmobile.models	-> java.lang	java.base

3.1.5. Analisi del progetto `2p-plugin` per Eclipse plugin

Per quanto riguarda il plugin relativo alla piattaforma Eclipse sono presenti il progetto `tuPrologPlugin` (versione 3.2.2), che importa la libreria `tuprolog.jar`, e i progetti `tuPrologUpdateSite` e `tuPrologFeature`. Per conoscere i pacchetti che `tuPrologPlugin` utilizza è possibile eseguire, all'interno della cartella in cui è presente il progetto, il comando:

```
jdeps tuPrologPlugin
```

Dall'output ottenuto, visualizzato nella tabella riportata di seguito, è possibile notare che i package `alice.tuprologx.eclipse.toolbar`, `alice.tuprologx.eclipse.core`, `alice.tuprologx.eclipse.views` e `alice.tuprologx.eclipse.util` situati all'interno della cartella `src` del plugin sviluppato per Eclipse utilizzano i pacchetti `alice.tuprolog`, `alice.tuprolog.event`, `alice.tuprolog.interfaces` e `alice.tuprolog.lib` appartenenti alla libreria `tuprolog.jar`.

<table>
<tr><td colspan="3">tuPrologPlugin -> java.base
tuPrologPlugin -> java.desktop
tuPrologPlugin -> not found</td></tr>
<tr><td rowspan="3">alice.tuprologx
.eclipse</td><td>-> alice.tuprologx.eclipse.core
-> alice.tuprologx.eclipse.perspective
-> alice.tuprologx.eclipse.properties
-> alice.tuprologx.eclipse.util</td><td>tuPrologPlugin</td></tr>
<tr><td>-> java.lang
-> java.net</td><td>java.base</td></tr>
<tr><td>-> org.eclipse.core.resources
-> org.eclipse.core.runtime
-> org.eclipse.jface.preference
-> org.eclipse.jface.resource
-> org.eclipse.swt.graphics
-> org.eclipse.ui
-> org.eclipse.ui.plugin
-> org.osgi.framework</td><td>not found</td></tr>
<tr><td>alice.tuprologx
.eclipse.core</td><td>-> alice.tuprolog
-> alice.tuprolog.event
-> alice.tuprolog.interfaces
-> alice.tuprolog.lib</td><td>not found</td></tr>
</table>

	-> alice.tuprologx.eclipse -> alice.tuprologx.eclipse.editors -> alice.tuprologx.eclipse.properties -> alice.tuprologx.eclipse.util -> alice.tuprologx.eclipse.views	tuPrologPlugin
	-> java.io -> java.lang -> java.util	java.base
	-> org.eclipse.core.resources -> org.eclipse.core.runtime -> org.eclipse.swt.widgets -> org.eclipse.ui	not found
alice.tuprologx .eclipse.editors	-> alice.tuprologx.eclipse -> alice.tuprologx.eclipse.core -> alice.tuprologx.eclipse.scanners -> alice.tuprologx.eclipse.util	tuPrologPlugin
	-> java.lang -> java.util	java.base
	-> org.eclipse.core.resources -> org.eclipse.core.runtime -> org.eclipse.jface.preference -> org.eclipse.jface.resource -> org.eclipse.jface.text -> org.eclipse.jface.text.presentation -> org.eclipse.jface.text.rules -> org.eclipse.jface.text.source -> org.eclipse.jface.util -> org.eclipse.swt.custom -> org.eclipse.swt.graphics -> org.eclipse.swt.widgets -> org.eclipse.ui -> org.eclipse.ui.editors.text -> org.eclipse.ui.texteditor	not found
alice.tuprologx .eclipse .importWizards	-> java.io -> java.lang	java.base
	-> org.eclipse.core.resources -> org.eclipse.core.runtime -> org.eclipse.jface.preference -> org.eclipse.jface.viewers -> org.eclipse.jface.wizard -> org.eclipse.swt.events -> org.eclipse.swt.layout -> org.eclipse.swt.widgets -> org.eclipse.ui -> org.eclipse.ui.dialogs	not found
alice.tuprologx .eclipse .perspective	-> java.lang	java.base
	-> org.eclipse.ui	not found
alice.tuprologx .eclipse .preferences	-> alice.tuprologx.eclipse	tuPrologPlugin
	-> java.lang	java.base
	-> org.eclipse.core.runtime .preferences -> org.eclipse.jface.preference -> org.eclipse.jface.resource -> org.eclipse.jface.util -> org.eclipse.swt.graphics	not found

	-> org.eclipse.swt.layout -> org.eclipse.swt.widgets -> org.eclipse.ui	
alice.tuprologx.eclipse.properties	-> alice.tuprologx.eclipse.core	tuPrologPlugin
	-> java.lang -> java.util	java.base
	-> org.eclipse.core.resources -> org.eclipse.core.runtime -> org.eclipse.jface.dialogs -> org.eclipse.swt.events -> org.eclipse.swt.graphics -> org.eclipse.swt.layout -> org.eclipse.swt.widgets -> org.eclipse.ui.dialogs	not found
alice.tuprologx.eclipse.scanners	-> alice.tuprologx.eclipse.util	tuPrologPlugin
	-> java.lang -> java.util	java.base
	-> org.eclipse.core.runtime -> org.eclipse.jface.text -> org.eclipse.jface.text.presentation -> org.eclipse.jface.text.rules	not found
alice.tuprologx.eclipse.toolbar	-> alice.tuprolog	not found
	-> alice.tuprologx.eclipse -> alice.tuprologx.eclipse.core -> alice.tuprologx.eclipse.wizards	tuPrologPlugin
	-> java.io -> java.lang	java.base
	-> org.eclipse.core.commands -> org.eclipse.core.resources -> org.eclipse.core.runtime -> org.eclipse.jface.viewers -> org.eclipse.jface.wizard -> org.eclipse.swt.widgets -> org.eclipse.ui -> org.eclipse.ui.part	not found
alice.tuprologx.eclipse.util	-> alice.tuprolog -> alice.tuprolog.event	not found
	-> alice.tuprologx.eclipse -> alice.tuprologx.eclipse.core	tuPrologPlugin
	-> java.awt -> java.awt.event -> java.awt.geom	java.desktop
	-> java.io -> java.lang -> java.util	java.base
	-> javax.swing -> javax.swing.border	java.desktop
	-> org.eclipse.core.resources -> org.eclipse.core.runtime -> org.eclipse.jface.dialogs -> org.eclipse.jface.preference -> org.eclipse.jface.text -> org.eclipse.jface.text.rules -> org.eclipse.jface.text.source -> org.eclipse.jface.util -> org.eclipse.swt.events	not found

	-> org.eclipse.swt.graphics -> org.eclipse.swt.layout -> org.eclipse.swt.widgets -> org.eclipse.ui.dialogs -> prefuse -> prefuse.action -> prefuse.action.animate -> prefuse.action.assignment -> prefuse.action.filter -> prefuse.action.layout -> prefuse.action.layout.graph -> prefuse.activity -> prefuse.controls -> prefuse.data -> prefuse.data.event -> prefuse.data.expression -> prefuse.data.search -> prefuse.data.tuple -> prefuse.render -> prefuse.util -> prefuse.util.ui -> prefuse.visual -> prefuse.visual.expression -> prefuse.visual.sort	
alice.tuprologx .eclipse.views	-> alice.tuprolog -> alice.tuprolog.event -> alice.tuprolog.interfaces -> alice.tuprolog.lib	not found
	-> alice.tuprologx.eclipse -> alice.tuprologx.eclipse.core -> alice.tuprologx.eclipse.editors -> alice.tuprologx.eclipse.util	tuPrologPlugin
	-> java.awt	java.desktop
	-> java.io -> java.lang -> java.util	java.base
	-> javax.swing	java.desktop
	-> org.eclipse.core.resources -> org.eclipse.core.runtime -> org.eclipse.jface.action -> org.eclipse.jface.dialogs -> org.eclipse.jface.resource -> org.eclipse.jface.viewers -> org.eclipse.swt.custom -> org.eclipse.swt.events -> org.eclipse.swt.graphics -> org.eclipse.swt.layout -> org.eclipse.swt.widgets -> org.eclipse.ui -> org.eclipse.ui.part -> prefuse.data	not found
alice.tuprologx .eclipse.wizards	-> alice.tuprologx.eclipse -> alice.tuprologx.eclipse.core -> alice.tuprologx.eclipse.properties	tuPrologPlugin
	-> java.io -> java.lang	java.base

	-> java.lang.reflect	
	-> org.eclipse.core.resources -> org.eclipse.core.runtime -> org.eclipse.jface.dialogs -> org.eclipse.jface.operation -> org.eclipse.jface.viewers -> org.eclipse.jface.wizard -> org.eclipse.swt.events -> org.eclipse.swt.layout -> org.eclipse.swt.widgets -> org.eclipse.ui -> org.eclipse.ui.dialogs -> org.eclipse.ui.ide	not found

3.2. Scelta dell'IDE: Eclipse Photon

L'ambiente di sviluppo scelto per modularizzare tuProlog è Eclipse Photon, in quanto si è preferito mantenere lo stesso IDE utilizzato precedentemente nelle altre fasi di sviluppo.
Sfortunatamente, però, Eclipse, a differenza di altri IDE, non permette ancora di realizzare un progetto che contenga più moduli. Pertanto, l'unica soluzione possibile è quella di creare un progetto per ciascun modulo.

La procedura per realizzare un modulo in Eclipse implica, quindi, la creazione di un nuovo progetto a cui poi si associa il Module Declaration.
I passi da seguire sono i seguenti:

- selezionare `File > New > Other… > Java Project > Next` dal menu di Eclipse.
- inserire, all'interno della nuova finestra, il nome del progetto, di norma corrispondere al nome del modulo, che si desidera creare e selezionare `Next`.
- selezionare l'opzione `Create module-info.java file` e successivamente `Finish` (*Figura 3.1*) per creare il progetto con all'interno il Module Declaration.

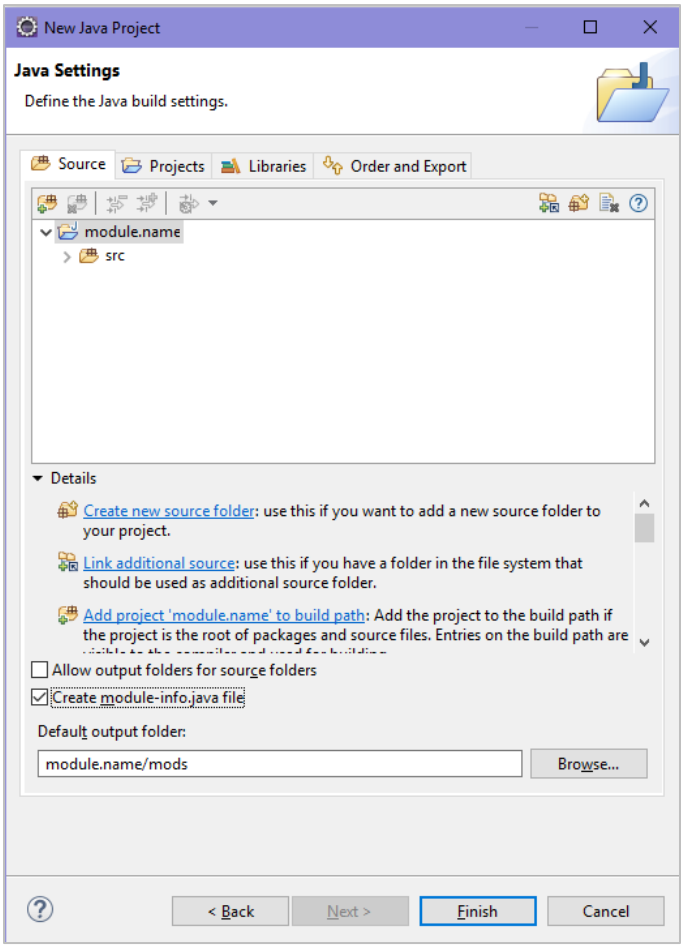

Figura 3.1 : Creazione progetto con Module Declaration

È anche possibile rimandare la creazione del file `module-info.java` in una fase successiva alla realizzazione del progetto non selezionando l'opzione `Create module-info.java file` (*Figura 3.1*); ma:

- selezionando l'opzione `Configure > Create module-info.java` tramite il menu a tendina, aperto cliccando con il tasto destro sul progetto (*Figura 3.2*);
- inserendo nella nuova finestra (*Figura 3.3*) il nome del modulo, di norma lo stesso del progetto, e selezionando `Create`.

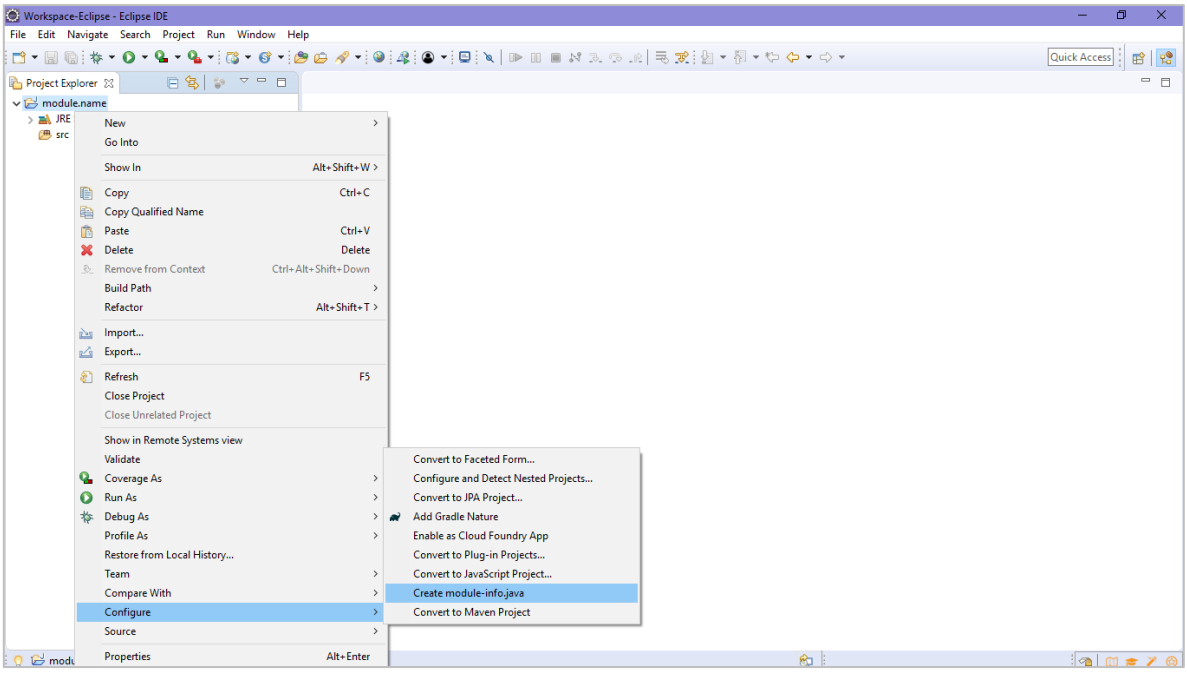

Figura 3.2 : Procedimento creazione Module Declaration

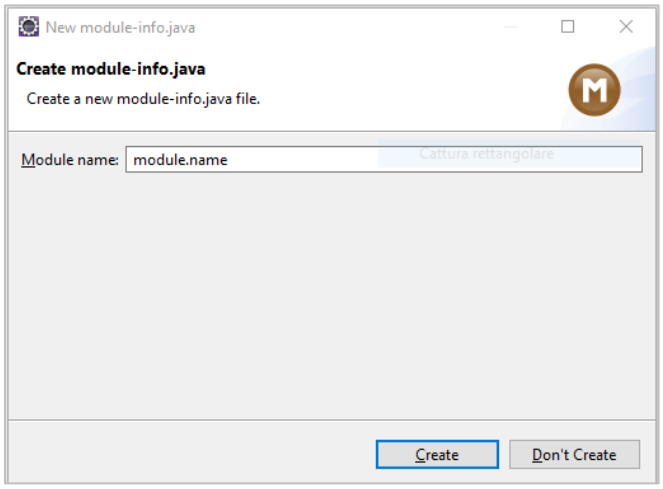

Figura 3.3 : Creazione Module Declaration

Nel caso in cui il modulo creato ne richieda un altro, quest'ultimo deve essere aggiunto all'interno del module path del progetto corrente aprendo il menu a tendina (cliccando col tasto destro sul progetto) e selezionando `Build Path > Configure Build Path…` (*Figura 3.4*). Dopodiché all'interno della scheda `Projects` bisogna selezionare il module path e l'opzione `Add…` (*Figura 3.5*), la quale apre una finestra che permette di indicare i moduli che si desidera inserire nel module path e di aggiungerli selezionando `OK` (*Figura 3.6*).

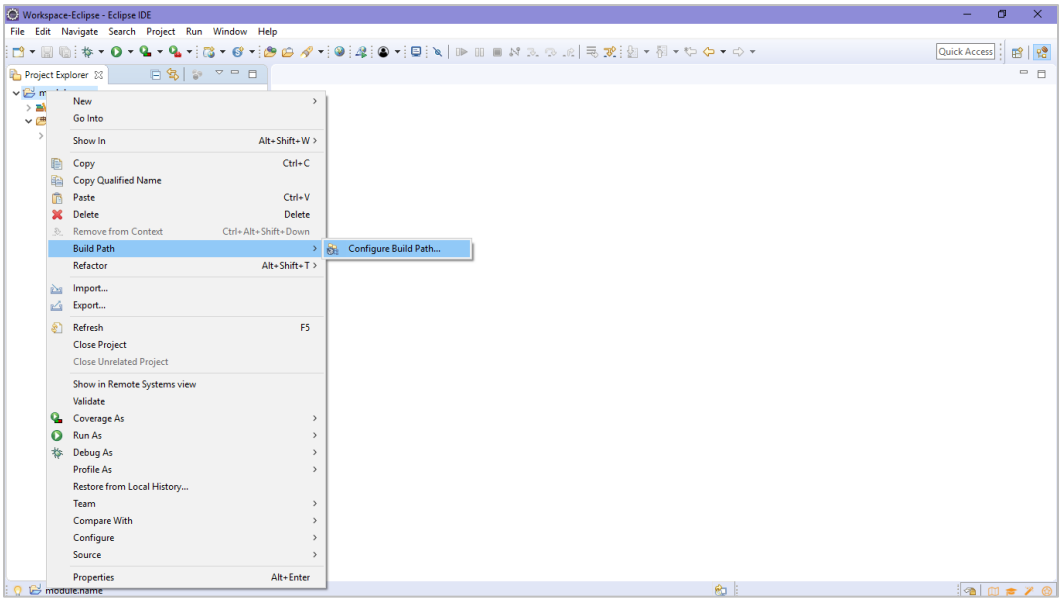

Figura 3.4 : Procedimento configurazione module path

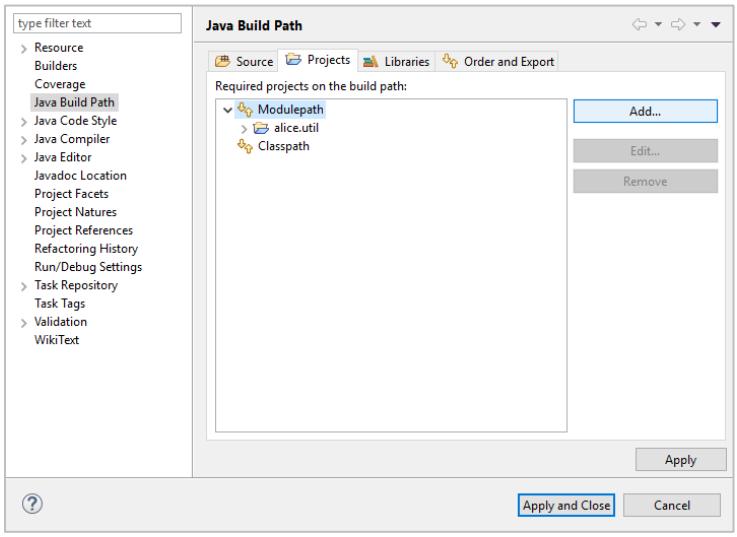

Figura 3.5 : Configurazione module path

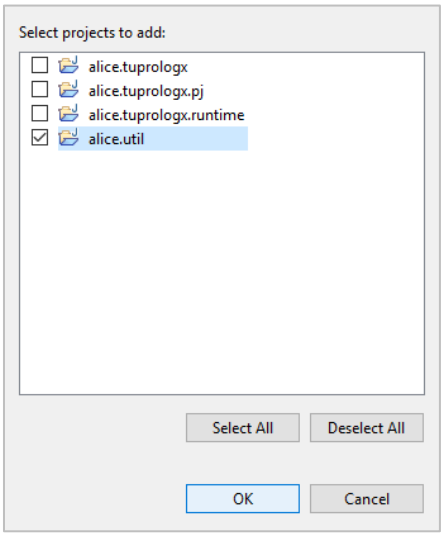

Figura 3.6 : Finestra di inserimento moduli all'interno del module path

Un procedimento simile deve essere effettuato per le librerie esterne, le quali devono essere spostate dal class path al module path all'interno della scheda `Libraries` (*Figura 3.7*). Questo procedimento è svolto in automatico nel caso in cui il Module Declaration sia creato successivamente al progetto e le librerie siano già importate da quest'ultimo. Inoltre, le librerie esterne, se non già modularizzate, quando spostate all'interno del module path vengono trasformate di default in moduli automatici.

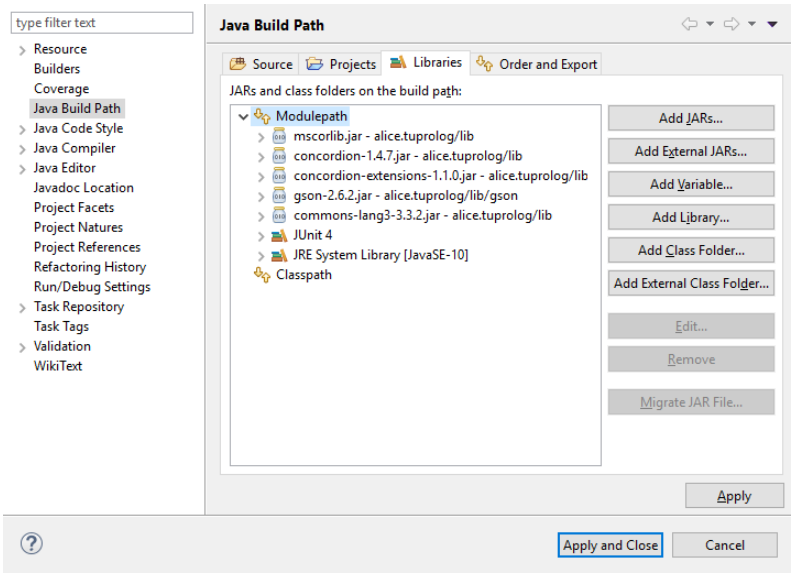

Figura 3.7 : Configurazione module path per le librerie

3.3. Proposta di modularizzazione

Utilizzando il materiale raccolto durante la fase di analisi delle dipendenze, dopo aver scelto l'ambiente di sviluppo, è stato possibile ipotizzare una proposta di modularizzazione di tuProlog.

La situazione iniziale è quella mostrata di seguito in *Figura 3.8*.

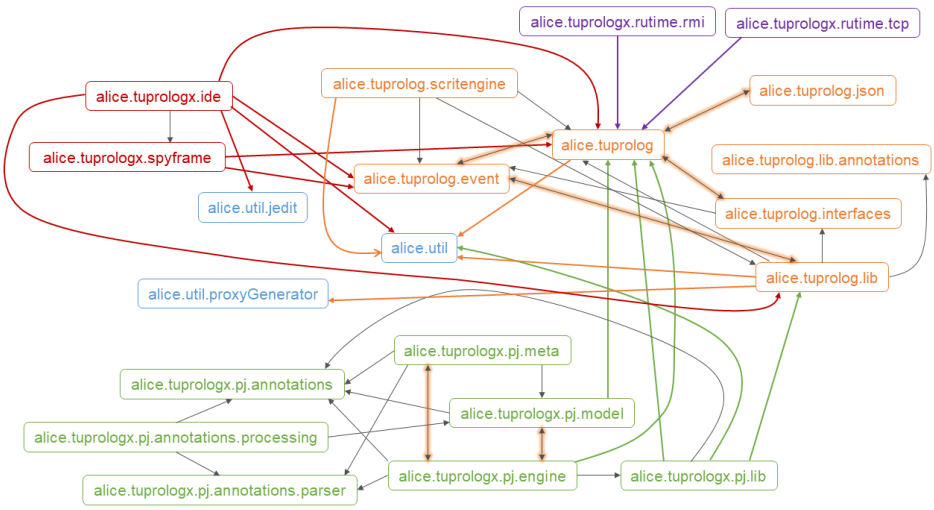

Figura 3.8 : Rappresentazione iniziale delle dipendenze tra package

In fase di modularizzazione, sono stati fissati i seguenti obbietti:

- rispettare le dipendenze tra i package;
- evitare cicli di dipendenza;
- rispettare la struttura logica del progetto `2p`.

In seguito a varie ipotesi di modularizzazione, è stato possibile infine realizzare cinque moduli, le cui dipendenze sono illustrate di seguito in *Figura 3.9*.

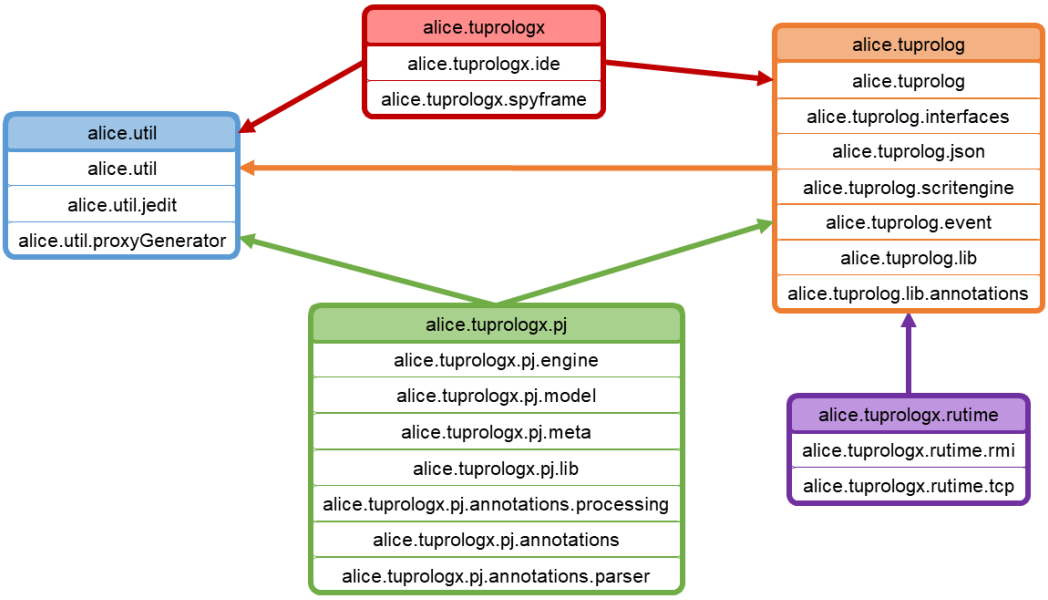

Figura 3.9 : Rappresentazione finale delle dipendenze tra moduli

Nei capitolo sottostanti sono illustrati uno ad uno i moduli ed il procedimento che ha portato alla loro realizzazione.

3.3.1. Module `alice.tuprolog`

Per evitare i cicli di dipendenza si è optato per unire all'interno dello stesso modulo i package `alice.tuprolog.interfaces`, `alice.tuprolog.json` e `alice.tuprolog.event` con `alice.tuprolog`, e in seguito questi con `alice.tuprolog.lib`. Non è stato quindi possibile creare un modulo separato per le librerie, problema forse risolvibile in futuro con un refactoring dei package all'interno dell'applicazione `2p`.

Inoltre, sono stati inseriti all'interno di questo modulo i package `alice.tuprolog.lib.annotations`, in quanto richiesto da `alice.tuprolog.lib`, e `alice.tuprolog.scriptengine`, per evitare moduli isolati con un solo package al loro interno, in quanto questo pacchetto non è richiesto da nessuno, ma importa i package `alice.tuprolog`, `alice.tuprolog.event` e `alice.tuprolog.lib`.

Il modulo `alice.tuprolog` è stato realizzato seguendo il procedimento spiegato al capitolo 3.2., inserendovi i package indicati e le librerie utili a questi ultimi.
Nelle figure 3.10 e 3.11 sono illustrate rispettivamente la struttura del progetto su Eclipse e quella ad albero (ottenuta eseguendo il comando `tree` seguito dal nome del progetto) del modulo compilato in automatico da Eclipse (se attivo il build automatico) con le cartelle: `lib`, contenente le librerie utilizzate; `src`, contenente i file sorgente del progetto; `mods`, contentente i file `.class` ed il Module Descriptor.

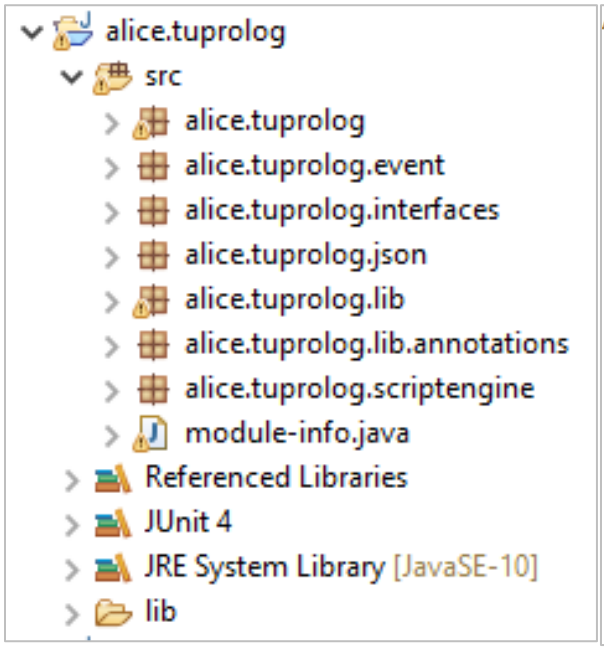

Figura 3.10 : Module alice.tuprolog package explorer

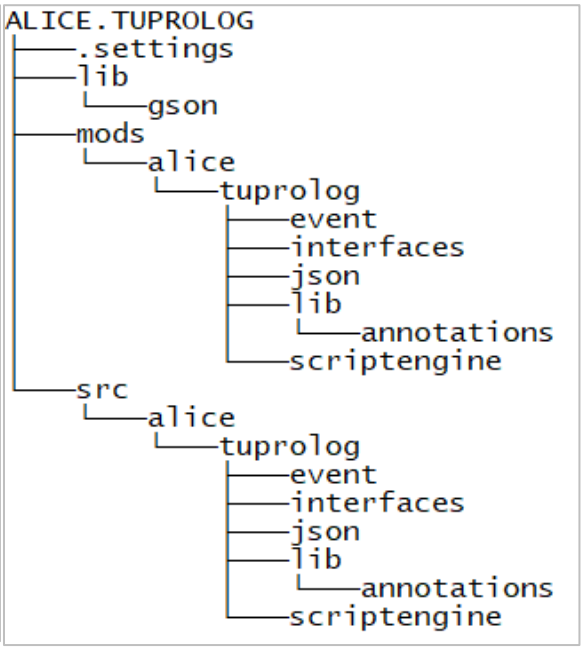

Figura 3.11 : Module alice.tuprolog tree

Di seguito è riportato il file `module-info.java` del modulo `alice.tuprolog`. È possibile notare come vengano esportati tramite `export` selettivo unicamente i package utilizzati dagli altri moduli. Inoltre, essendo il modulo corrente e quasi tutti gli altri moduli che lo richiedono dipendenti da `alice.util`, si è deciso di utilizzare

la direttiva `requires transitive` per esprimere tale dipendenza, in modo che chi richiede `alice.tuprolog` non debba richiedere anche `alice.util`, in quanto questa dipendenza risulta transitiva. Al contrario, il modulo automatico `mscorlib`, corrispondente alla libreria esterna omonima, non deve essere richiesto da `alice.tuprolog` in quanto lo è già tramite la direttiva `requires transitive` da `alice.util`, ma deve essere comunque inserito all'interno del module path delle librerie.

```
module alice.tuprolog
{
   /* export selettivo dei package appartenenti al modulo */
   exports alice.tuprolog.event to alice.tuprologx;
   exports alice.tuprolog to alice.tuprologx.pj, alice.tuprologx.runtime,
alice.tuprologx;
   exports alice.tuprolog.lib to alice.tuprologx.pj, alice.tuprologx;

   /* si utilizza requires transitive in modo che chi richiede questo
      modulo possa utilizzare anche il modulo indicato dalla direttiva */
   requires transitive alice.util;

   /* richiesta librerie utilizzate all'interno del modulo */
   requires java.scripting;
   requires commons.lang3;
   requires gson;
   /* la libreria mscorlib è richiesta tramite requires transitive
      dal modulo alice.util */
}
```

3.3.2. Module `alice.util`

Si è deciso di inserire in uno stesso modulo i package `alice.util`, `alice.util.jedit` e `alice.util.proxyGenerator`, in quanto non dipendono da nessun altro package.

Il modulo corrente è stato realizzato seguendo il procedimento spiegato al capitolo 3.2., inserendovi i package indicati e le librerie utili a questi ultimi.

Nelle figure 3.12 e 3.13 sono illustrate rispettivamente la struttura del progetto su Eclipse e quella ad albero del modulo, compilato in automatico da Eclipse, con le cartelle: `lib`, contenente le librerie utilizzate; `src`, contenente i file sorgente del progetto; `mods`, contentente i file `.class` ed il Module Descriptor.

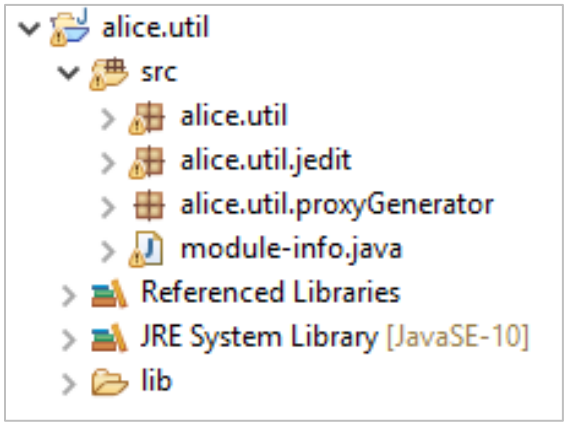

Figura 3.12 : Module alice.util package explorer

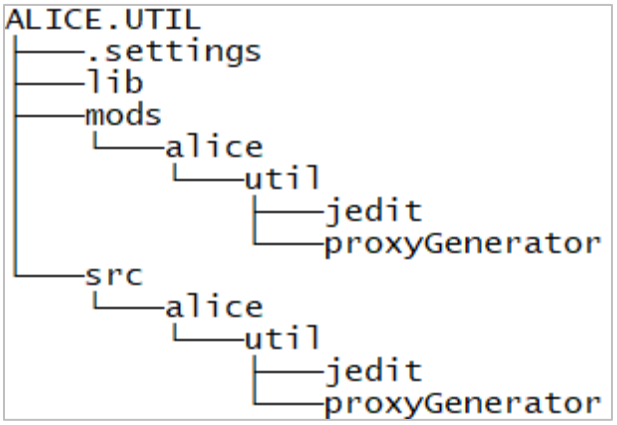

Figura 3.13 : Module alice.util tree

Di seguito è riportato il file `module-info.java` del modulo `alice.util`, dal quale è possibile notare l'utilizzo della direttiva `exports…to` per importare unicamente i package richiesti dagli altri moduli. Inoltre, essendo la libreria `mscorlib` utilizzata anche da altri moduli dipendenti da `alice.util`, si è deciso di utilizzare la direttiva `requires transitive` per esprimere tale relazione, in modo che chi richiede `alice.util` non debba richiedere anche `mscorlib`, in quanto questa dipendenza risulta transitiva.

```
module alice.util
{
   /* export selettivo dei package appartenenti al modulo */
   exports alice.util to alice.tuprologx, alice.tuprolog,
alice.tuprologx.pj;
   exports alice.util.jedit to alice.tuprologx;
   exports alice.util.proxyGenerator to alice.tuprolog;

   /* richiesta librerie utilizzate all'interno del modulo */
   requires java.compiler;
   requires java.datatransfer;
   requires java.desktop;
```

```
    requires ikvm.api;
    /* si usa requires transitive in modo che chi richiede questo modulo
       possa utilizzare anche la libreria indicata dalla direttiva */
    requires transitive mscorlib;
}
```

3.3.3. Module `alice.tuprologx`

Essendo il package `alice.tuprologx.ide` dipendente da `alice.tuprologx.spyframe`, si è deciso di inserire entrambi all'interno di un unico modulo `alice.tuprologx`.

Tale modulo è stato creato seguendo il procedimento spiegato al capitolo 3.2., inserendovi i package indicati e le librerie utili a questi ultimi.
Nelle figure 3.14 e 3.15 vengono illustrate rispettivamente la struttura del progetto su Eclipse e quella ad albero del modulo, compilato in automatico da Eclipse, con le cartelle: `lib`, contenente le librerie utilizzate; `src`, contenente i file sorgente del progetto; `mods`, contentente i file `.class` ed il Module Descriptor.

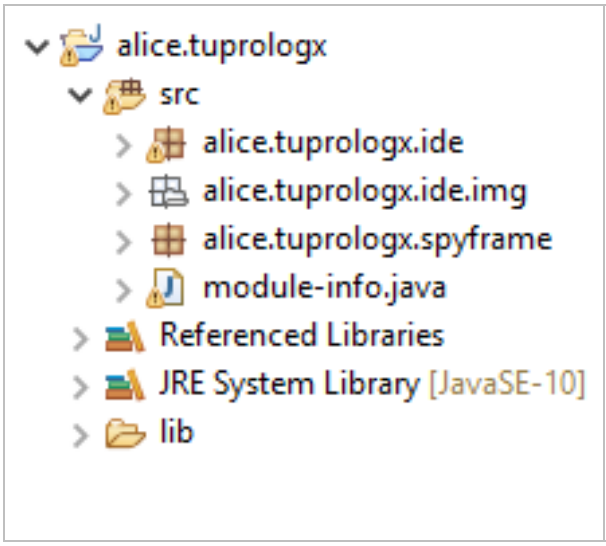

Figura 3.14 : Module alice.tuprologx package explorer

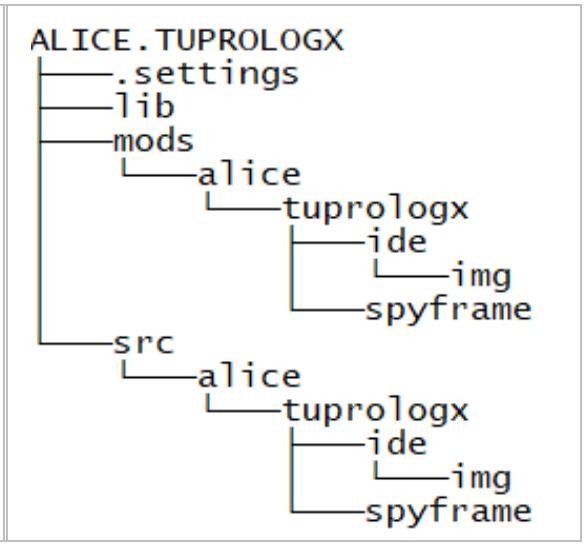

Figura 3.15 : Module alice.tuprologx tree

Di seguito è riportato il file `module-info.java` di `alice.tuprologx`, nel quale è possibile notare che questo modulo, non essendo richiesto da nessuno altro, non

esporta alcun package, ma dipende dai moduli `alice.tuprolog` e `alice.util`. Quest'ultimo però è già richiesto tramite la direttiva `requires transitive` da `alice.tuprolog`; perciò non è necessario richiederlo nuovamente, ma deve essere comunque inserito all'interno del module path del progetto. Allo stesso modo, il modulo automatico `mscorlib`, corrispondente alla libreria esterna omonima, non deve essere richiesto da `alice.tuprologx` in quanto lo è già tramite la direttiva `requires transitive` da `alice.util`, ma deve essere comunque inserito all'interno del module path delle librerie.

```
module alice.tuprologx
{
   /* richiesta moduli utilizzati da questo modulo */
   requires alice.tuprolog;
   /* il modulo alice.util è richiesto tramite requires transitive
      dal modulo alice.tuprolog */

   /* richiesta librerie utilizzate all'interno del modulo */
   requires java.desktop;
   requires autocomplete;
   requires rsyntaxtextarea;
   /* la libreria mscorlib è richiesta tramite requires transitive
      dal modulo alice.util */
}
```

3.3.4. Module `alice.tuprologx.pj`

Per evitare i cicli di dipendenza tra i package `alice.tuprologx.pj.meta`, `alice.tuprologx.pj.model` e `alice.tuprologx.pj.engine`, questi sono stati inseriti all'interno del modulo `alice.tuprologx.pj`.
L'idea iniziale era quella di creare un altro modulo, in aggiunta a questo, che comprendesse i package `alice.tuprologx.pj.annotations`, `alice.tuprologx.pj.annotations.processing` e `alice.tuprologx.pj.annotations.parser`. La realizzazione di questi due moduli però avrebbe comportato un altro ciclo di dipendenza; pertanto, si è ritenuto opportuno unire i due moduli.

Infine, per evitare un altro ciclo di dipendenza con il package `alice.tuprologx.lib` è stato necessario inserire anche quest'ultimo all'interno del modulo ottenuto (`alice.tuprologx.pj`).

Il modulo `alice.tuprologx.pj` è stato creato seguendo il procedimento spiegato al capitolo 3.2. per la realizzazione di un progetto modulare al quale sono stati aggiunti i package indicati e sono state importate le librerie esterne utilizzate.
Nelle figure 3.16 e 3.17 vengono illustrate rispettivamente la struttura del progetto su Eclipse e quella ad albero del modulo, compilato in automatico da Eclipse, con le cartelle: `lib`, contenente le librerie utilizzate; `src`, contenente i file sorgente del progetto; `mods`, contentente i file `.class` ed il Module Descriptor.

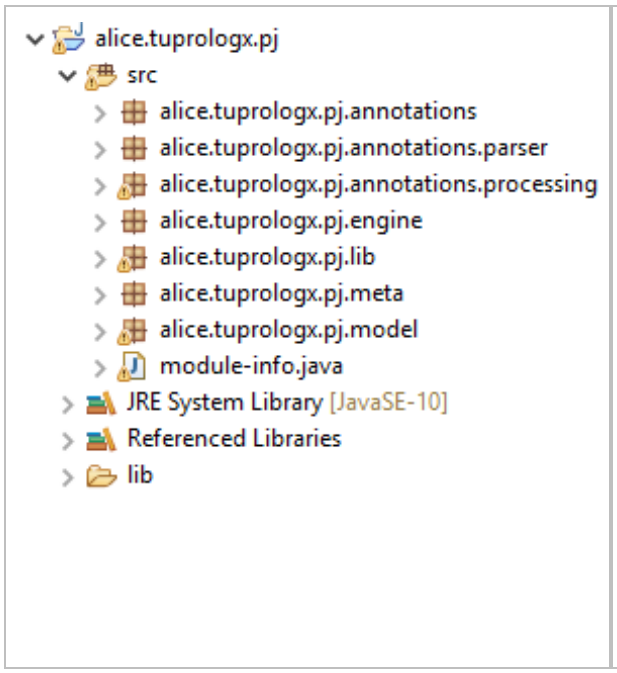

Figura 3.16 : Module alice.tuprologx.pj package explorer

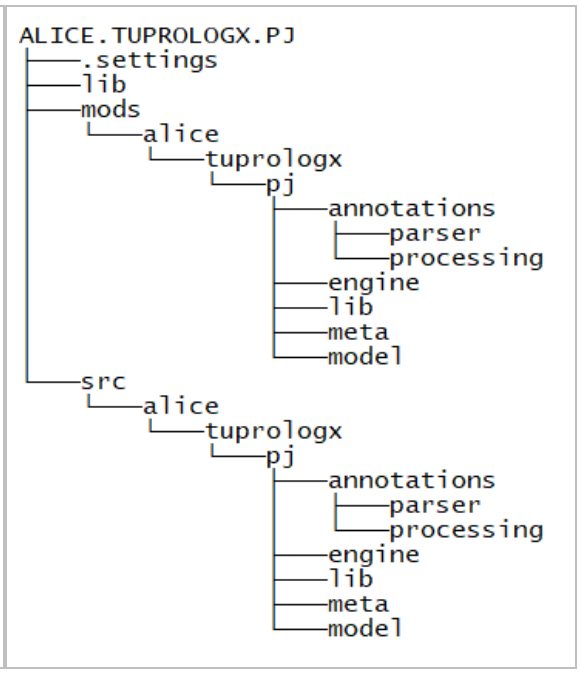

Figura 3.17 : Module alice.tuprologx.pj tree

Di seguito è riportato il file `module-info.java` di `alice.tuprologx.pj`, nel quale è possibile notare che, esattamente come `alice.tuprologx`, questo modulo, non essendo richiesto da nessuno altro, non esporta alcun package, ma dipende dai moduli `alice.tuprolog` e `alice.util`. Quest'ultimo però, essendo già richiesto tramite la direttiva `requires transitive` da `alice.tuprolog`, non è necessario richiederlo nuovamente; bisogna, invece, inserirlo comunque all'interno del module path del progetto.

```
module alice.tuprologx.pj
{
   /* richiesta moduli utilizzati da questo modulo */
   requires alice.tuprolog;
   /* il modulo alice.util è richiesto tramite requires transitive dal
      modulo alice.tuprolog */

   /* richiesta librerie utilizzate all'interno del modulo */
   requires java.compiler;
   requires java.desktop;
   requires javassist;
}
```

3.3.5. Module `alice.tuprologx.runtime`

Dall'analisi svolta in precedenza si è potuto constatare che i package `alice.tuprologx.runtime.rmi` e `alice.tuprologx.runtime.tcp` non vengono importati da nessun altro e che dipendono unicamente dal package `alice.tuprolog`.
Si è deciso quindi di inserire questi due package all'interno del modulo `alice.tuprologx.runtime` creato seguendo il procedimento spiegato al capitolo 3.2.
Nelle figure 3.18 e 3.19 vengono illustrate rispettivamente la struttura del progetto su Eclipse e quella ad albero del modulo, compilato in automatico da Eclipse, con le cartelle: `src`, contenente i file sorgente del progetto; `mods`, contentente i file `.class` ed il Module Descriptor.

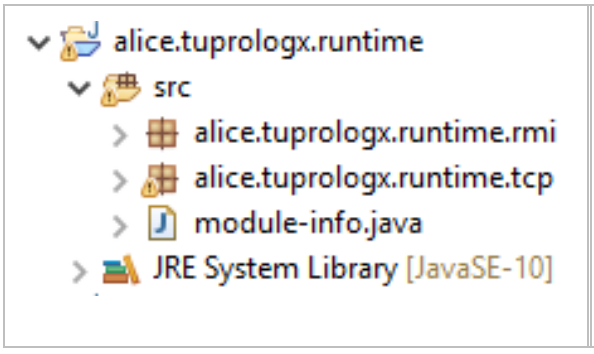

Figura 3.18 : Module alice.tuprologx.runtime package explorer

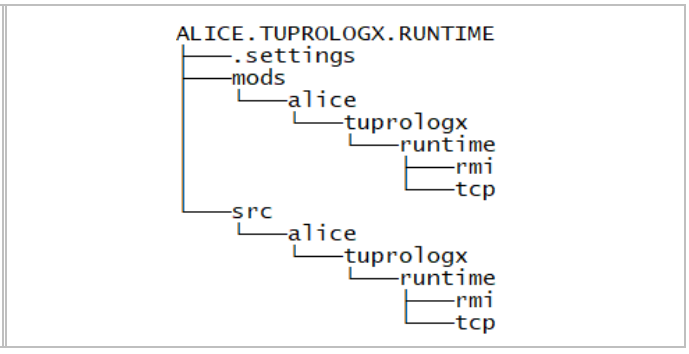

Figura 3.19 : Module alice.tuprologx.runtime tree

Di seguito è riportato il file `module-info.java` del modulo `alice.tuprologx.runtime`, il quale richiede unicamente il modulo `alice.tuprolog`.

```
module alice.tuprologx.runtime
{
   /* richiesta moduli utilizzati da questo modulo */
   requires alice.tuprolog;

   /* richiesta librerie utilizzate all'interno del modulo */
   requires java.rmi;
}
```

3.4. Applicazione modulare tuProlog per Java e Creazione file JAR

In seguito alla realizzazione dei singoli moduli, è stato possibile creare un'applicazione modulare per tuProlog in Java.

Non essendo però ancora possibile su Eclipse creare un progetto unico che contenga più moduli, si è deciso di inserire le altre risorse (tra cui le cartelle `test`, `doc` e `ant` ed i file di licenza), presenti nel progetto originale `2p`, all'interno del modulo `alice.tuprolog`, in quanto è utilizzato dai test (*Figura 3.20*).

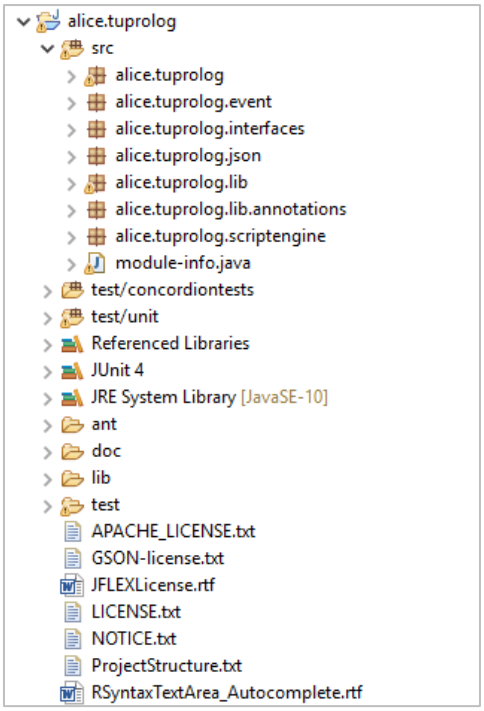

Figura 3.20 : Package explorer di alice.tuprolog con test e risorse

Con l'aggiunta delle risorse all'interno del progetto `alice.tuprolog` è stato necessario inserire all'interno del file `module-info.java`, riferito a questo modulo, anche le librerie richieste dai test:

```
requires concordion;
requires concordion.extensions;
```

La proposta finale di modularizzazione di tuProlog per Java è pertanto costituita dai moduli `alice.tuprolog`, `alice.tuprologx`, `alice.util`, `alice.tuprologx.pj` e `alice.tuprologx.runtime`.

Per poter creare il file JAR eseguibile di tale applicazione si può usufruire di due metodi differenti: uno da linea di comando, come spiegato precedentemente all'interno del capitolo 2.1.7.; uno direttamente da Eclipse. Il procedimento da seguire per quest'ultimo, è il seguente:

- selezionare `File > Export… > Java > Runnable JAR file > Next` dal menu di Eclipse. Bisogna essere all'interno del *Workspace* contenente

tutti i moduli e questi devono essere già compilati (operazione effettuata automaticamente da Eclipse se attivo il build automatico).

- indicare, in `Launch configuration`, il main ed il modulo a cui questo appartiene e, in `Export destination`, il percorso con il nome del file JAR da creare. Dopodiché, selezionare l'opzione che si preferisce per le librerie ed infine `Finish` per creare il file JAR (*Figura 3.21*).

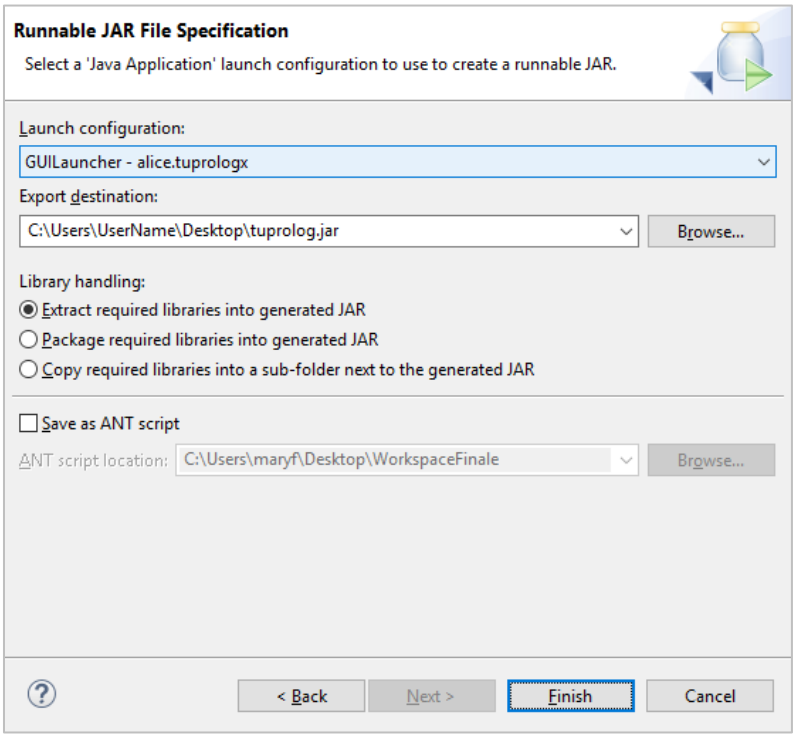

Figura 3.21 : Configurazione e creazione file JAR eseguibile

Dai moduli creati nel capitolo 3.2 e dall'analisi delle dipendenze effettuata nel capitolo 3.1, è stato inoltre possibile realizzare i file JAR personalizzati dei moduli richiesti da ciascuna delle applicazioni tuProlog disponibili per le altre piattaforme, i quali potrebbero sostituire la libreria `tuprolog.jar` già importata.

Per creare il file JAR di un modulo è possibile seguire il metodo da linea di comando, come spiegato nel capitolo 2.1.7., o tramite Eclipse:

- selezionare `File > Export… > Java > JAR file > Next` dal menu di Eclipse;

- indicare il modulo precompilato (deselezionando eventualmente i package indesiderati) da cui si desidera ricavare il file JAR ed il percorso con il nome del file da creare e terminare selezionando `Finish` (*Figura 3.22*).

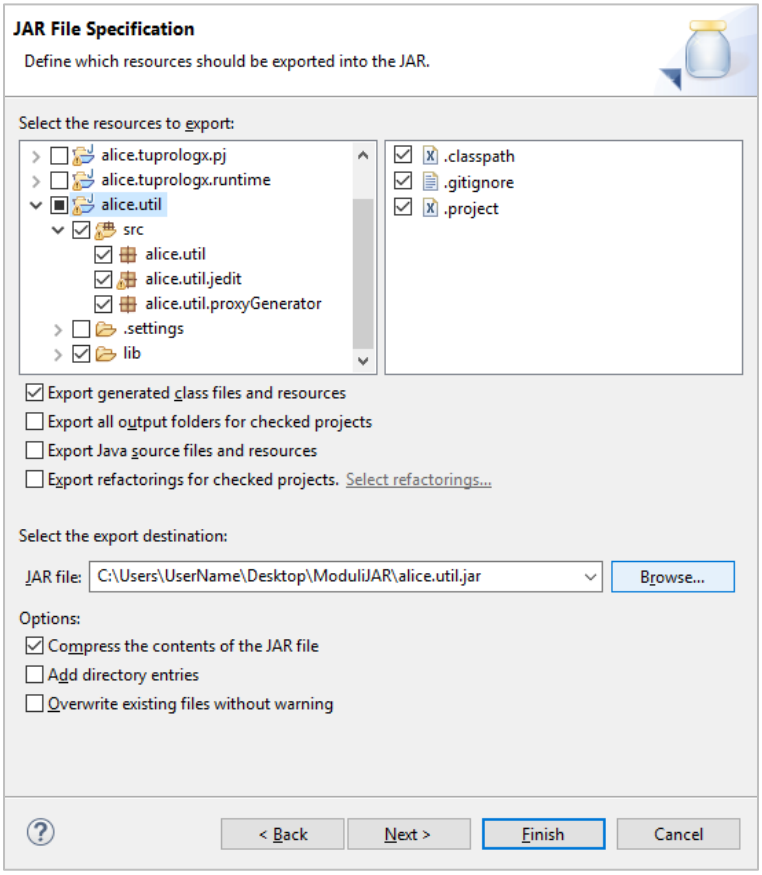

Figura 3.22 : Configurazione e creazione file JAR

In particolare, nei capitoli sottostanti vengono illustrati singolarmente i procedimenti seguiti per ciascuna applicazione nella scelta dei moduli da importare in sostituzione alla libreria `tuprolog.jar`.

3.4.1. Moduli tuProlog per piattaforma iOS

Dall'analisi delle dipendenze del progetto `2p-ios` sviluppato per la piattaforma iOS, effettuata in precedenza nel capitolo 3.1.2., è risultato che quest'applicazione utilizza unicamente il package `alice.tuprolog`, il quale è situato all'interno del modulo omonimo. Quest'ultimo però risulta dipendente da `alice.util`.

Pertanto, per questa applicazione, bisogna creare i file JAR dei moduli `alice.tuprolog` e `alice.util`.

3.4.2. Moduli tuProlog per Microsoft .NET

Dall'analisi delle dipendenze del progetto sviluppato per Microsoft .NET, effettuata in precedenza nel capitolo 3.1.3., si è potuto notare che vengono utilizzati i package `alice.tuprolog` e `alice.tuprolog.event`.
Questi si trovano all'interno del modulo `alice.tuprolog`, il quale richiede a sua volta `alice.util`.

Pertanto, per questa applicazione è necessario creare i file JAR dei moduli `alice.tuprolog` e `alice.util`.

3.4.3. Moduli tuProlog per piattaforma Android

Dall'analisi delle dipendenze del progetto `2p-android` sviluppato per la piattaforma Android, effettuata in precedenza nel capitolo 3.1.4., è risultato che quest'applicazione utilizza i package `alice.tuprolog.event`, `alice.tuprolog` e `alice.tuprolog.lib`, situati all'interno del modulo `alice.tuprolog`, e `alice.util`, situato all'interno del modulo omonimo.

Pertanto, si devono creare i file JAR dei moduli `alice.tuprolog` e `alice.util`.

3.4.4. Moduli tuProlog per Eclipse plugin

Dall'analisi delle dipendenze del plugin sviluppato per Eclipse, effettuata in precedenza nel capitolo 3.1.5., è risultato che i package utilizzati sono `alice.tuprolog`, `alice.tuprolog.event`, `alice.tuprolog.interfaces` e `alice.tuprolog.lib`.

Questi si trovano all'interno del modulo `alice.tuprolog`, il quale richiede a sua volta `alice.util`.

Pertanto, per quest'applicazione è necessario creare i file JAR dei moduli `alice.tuprolog` e `alice.util`.

Conclusioni

La tesi è stata svolta con l'obbiettivo di valutare e in seguito formulare una proposta di modularizzazione dell'interprete tuProlog per la piattaforma Java che rispetti le relazioni tra gli elementi del progetto.

In seguito allo studio dell'applicazione tuProlog per Java e ad un'analisi accurata delle dipendenze all'interno del progetto da modularizzare e delle applicazioni tuProlog sviluppate per le altre piattaforme (Android, Microsoft .NET, Eclipse plugin, iOS), è stato possibile progettare cinque moduli.
Questo ha permesso di realizzare librerie personalizzate per ciascuna delle applicazioni tuProlog presenti per le altre piattaforme.
Inoltre, un'applicazione organizzata in moduli può risultare vantaggiosa per quanto riguarda la fase di sviluppo e manutenzione delle future versioni di tuProlog, permettendo, ad esempio, di modificare determinate caratteristiche o funzionalità semplicemente intervenendo sui moduli.

La scelta di Eclipse come IDE per la progettazione della proposta modulare ha comportato particolari difficoltà, in quanto quest'ambiente di sviluppo, a differenza di altri, non permette di creare un progetto unico contenente più moduli, ma obbliga la creazione di un progetto per ciascun modulo. Un miglioramento futuro pertanto potrebbe avvenire in seguito ad un aggiornamento di Eclipse che risolva tale problema o all'eventuale scelta di un diverso ambiente di sviluppo, quale IntelliJ, che supporti nativamente i moduli senza costringere a partizionare artificiosamente il progetto in progetti mutuamente dipendenti.

Riferimenti Bibliografici

[1] Denti Enrico, *tuProlog Manual*. Disponibile all'indirizzo: `https://bitbucket.org/tuprologteam/tuprolog/downloads/` (14 Febbraio 2017)

[2] Bitbucket. Documentazione e download tuProlog disponibili all'indirizzo: `https://bitbucket.org/tuprologteam/tuprolog`

[3] Apice, *tuProlog Home*. Disponibile all'indirizzo: `http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome`

[4] Denti Enrico, *Il concetto di modulo in Java 9*. (Anno Accademico 2017-2018)

[5] Oracle, *Understanding Java 9 Modules*. Disponibile all'indirizzo: `https://www.oracle.com/corporate/features/understanding-java-9-modules.html` (Settembre/Ottobre 2017)

[6] Oracle Java Magazine, *More Java 9*. Disponibile all'indirizzo: `http://www.javamagazine.mozaicreader.com/SeptOct2017#&pageSet=17&page=0` (Settembre/Ottobre 2017)

[7] Oracle, *Java SE Technologies*. Disponibile all'indirizzo: `http://www.oracle.com/technetwork/java/javase/tech/index.html`

[8] Oracle Help Center, *Java Platform, Standard Edition Tools Reference – jdeps*. Disponibile all'indirizzo: `https://docs.oracle.com/javase/9/tools/jdeps.htm#JSWOR690` (2017)

[9] Oracle Help Center, *Java Platform, Standard Edition Tools Reference – jmod*. Disponibile all'indirizzo: `https://docs.oracle.com/javase/9/tools/jmod.htm#JSWOR-GUID-0A0BDFF6-BE34-461B-86EF-AAC9A555E2AE` (2017)

[10] Oracle Help Center, *Java Platform, Standard Edition Tools Reference – jlink*. Disponibile all'indirizzo: `https://docs.oracle.com/javase/9/tools/jlink.htm#JSWOR-GUID-CECAC52B-CFEE-46CB-8166-F17A8E9280E9` (2017)

[11] Oracle Help Center, *JDK 10 Documentation*. Disponibile all'indirizzo: `https://docs.oracle.com/javase/10/` (2018)

[12] Oracle, *JDK 10 Release Note*. Disponibile all'indirizzo:
http://www.oracle.com/technetwork/java/javase/10-relnote-issues-4108729.html#NewFeature