ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Dipartimento di Informatica - Scienza e Ingegneria DISI

TESI DI LAUREA

in

Fondamenti di Informatica T-2

Revisione architetturale di plug-in multilinguaggio su piattaforma Eclipse

CANDIDATO
Federico Stella

RELATORE
Prof. Enrico Denti

CORRELATORE
Dott. Ing. Roberta Calegari

Anno Accademico 2016/17

Sessione I

Prefazione

Premettendo che non sono mai stato una persona particolarmente incline a manifestazioni di affetto, compresi i ringraziamenti, mi sento comunque di scrivere qualche parola a conclusione di questo primo, piccolo, percorso.

Una Laurea è certamente un traguardo importante, conseguito dopo tre anni di impegno e diverse soddisfazioni, ma nella mia mente non è che, ancora, un piccolo tassello. Il mio obiettivo, forse apparentemente banale, è il continuo miglioramento e accrescimento culturale, e questi due concetti non si valutano certo nel breve termine. Perciò le motivazioni al "piccolo tassello" sono che da un lato non è possibile valutare con un titolo l'esperienza, l'intelligenza e la competenza di una persona, così come la sua volontà di mettersi in gioco; dall'altro, anche riconoscendolo come traguardo, rimane un semplice inizio, per quanto positivo, di ciò che sarà il mio percorso futuro.
La mia forte speranza è quella di poter giungere nel luogo che mi sono prefisso, un luogo in senso figurato nel quale sarò soddisfatto di tutto ciò che ho appreso e, nonostante questo, vorrò continuare a imparare. Il percorso si è annunciato lungo sin dall'inizio, ed è quindi cominciato con questo Corso di Laurea. Seguiranno sicuramente altri percorsi di apprendimento, finalizzati, secondo la via che ho provato a delineare, ad accrescere in primo luogo la mia esperienza nel settore che ho scelto, quello informatico, e successivamente a soddisfare la mia necessità di conoscenza "totale", perciò anche in altri settori.

Quando ho iniziato questo Corso il mio pensiero era molto differente, soprattutto poiché venivo da un periodo caratterizzato da una chiara percezione della mia identità, ma anche da una forte insoddisfazione in ciò che stavo facendo, nonostante gli ottimi risultati. Durante questi tre anni ho perso progressivamente la chiara identità che percepivo prima, e questo certamente è stato un male, ma ho poi acquisito un'idea molto chiara di ciò che voglio fare. Nei prossimi mesi e anni dovrò quindi concentrarmi sul ritrovare del tutto la mia identità e il fatto di voler proseguire a lungo con l'apprendimento mi aiuterà certamente.

In questi anni il supporto più forte è arrivato dalla mia famiglia, sia da parte dei miei genitori e di mia sorella, sia da parte dei miei nonni, e non posso che ringraziare tutti perché l'appoggio di chi mi è sempre stato più vicino è la cosa più importante per me.
In tutto ciò non dimentico chi, tra i miei amici, ha sopportato i miei monologhi di sfogo e chi mi è sinceramente rimasto vicino, nel modo di pensare, scavalcando l'invidia che facilmente sorge tra persone.

Sommario

Introduzione

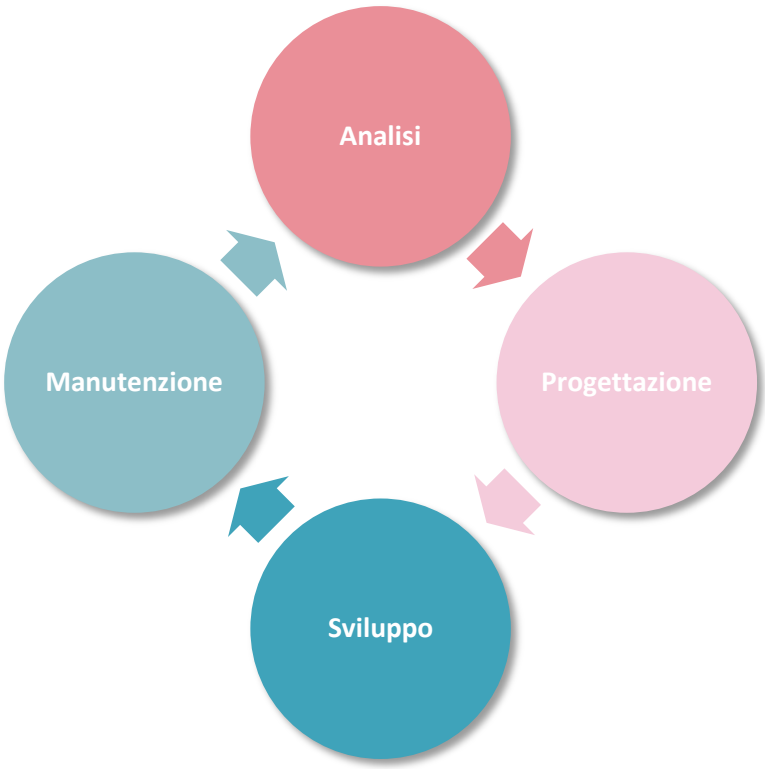
L'obiettivo di questa tesi è quello di revisionare l'architettura del plug-in tuProlog per Eclipse in modo da renderlo compatibile con le versioni attuali della piattaforma, aggiornando anche il plug-in stesso per supportare le versioni più recenti di tuProlog.
Attualmente il plug-in risulta compatibile con le versioni di Eclipse anteriori a Juno, compreso, e fa uso della libreria tuProlog 2.9.1, versione ora superata.

Per poterne analizzare l'architettura e per poter individuare i problemi è necessario disporre di strumenti adeguati e, inoltre, è altrettanto necessario mettere a punto un processo per il collaudo che sia il più possibile semplice da utilizzare ed efficace nel suo funzionamento, in modo da garantire risultati attendibili. Per poter giungere a questo risultato è necessario analizzare sia la piattaforma di riferimento, ovvero Eclipse, sia gli strumenti che essa offre.
Fra gli strumenti utilizzati, e pertanto presentati, vi sono tool interni a Eclipse, come il Plug-in Development Environment, ed anche software esterni, come VirtualBox.

Il percorso della tesi partirà quindi da considerazioni generali sul significato dello sviluppo software e della manutenzione, passando poi ad analizzare la piattaforma e gli strumenti fondamentali che questa offre, osservando come il discorso generale su sviluppo e manutenzione si applichi ad Eclipse ed ai suoi strumenti.
Dopodiché si passerà a introdurre tuProlog e il suo plug-in, spiegandone il funzionamento.
Infine si procederà ad analizzare precisamente i problemi che causano la mancata compatibilità ed a risolverli, mettendo a punto un opportuno sistema per il collaudo.

1 Il processo di sviluppo software

Il processo di sviluppo software si articola oggi in 4 fasi principali: l'analisi, la progettazione, lo sviluppo e la manutenzione. La fase di analisi mira a fornire uno studio preliminare del dominio del sistema, facilitando il lavoro di progettazione e rendendo il più possibile chiare le specifiche del progetto. La fase di progettazione ha lo scopo di fornire un quadro completo e dettagliato su come sarà realizzata la struttura del software e su quali approcci o pattern saranno impiegati in fase di sviluppo, fase che fornirà l'implementazione effettiva del software.
Dopo queste prime 3 fasi, e loro successive iterazioni, un software può considerarsi pronto per la pubblicazione e per l'utilizzo, tuttavia il suo ciclo di sviluppo non è affatto terminato: la fase più lunga e importante del ciclo è certamente la manutenzione.
La fase di manutenzione ricopre oggi un ruolo fondamentale a causa della continua evoluzione dei sistemi modellati e delle funzionalità richieste, nonché a causa dell'importanza del bugfixing e delle patch di sicurezza. Durante la manutenzione si individuano problemi critici o nuove funzionalità da implementare e si ricomincia il ciclo di analisi-progettazione-sviluppo per poterle applicare al sistema: è ovvia, in tutto ciò, la fondamentale importanza del versioning e della documentazione in ogni passo del processo di sviluppo.



Il discorso sulla manutenzione va incontro a una profonda estensione nel momento in cui si parla di sviluppo software all'interno di una piattaforma specifica, la quale quindi subirà un proprio processo di manutenzione e una propria evoluzione. In questi casi è necessario adattare il proprio software ai cambiamenti che avvengono nella piattaforma di supporto.
È sicuramente vero che una piattaforma ben progettata nelle sue fasi iniziali non dovrebbe andare incontro a modifiche architetturali tali da rendere incompatibili i cambiamenti tra le sue diverse versioni, tuttavia la realtà dei fatti è che alla nascita di un software o di una piattaforma non è facile prevedere quali saranno gli sviluppi futuri.

Per questi motivi, sviluppare software all'interno di una piattaforma specifica presenta dei rischi e necessita di una fase di manutenzione più ampia, per potersi adattare ad eventuali cambiamenti non dipendenti dal proprio sistema. Nelle sezioni seguenti sarà analizzata la piattaforma Eclipse, sulla quale è stato sviluppato il plug-in oggetto di questa tesi, spiegando la sua struttura e i tool che offre per lo sviluppo del software.

1.1 La piattaforma Eclipse

L'Eclipse Project è stato creato da IBM a partire dal 2001, mentre il gruppo no-profit di programmatori noto come Eclipse Foundation ne cura lo sviluppo dal 2004. La release attuale è Neon (Eclipse 4.6) dal giugno 2016 [1].
Eclipse è uno strumento ampiamente utilizzato dalla comunità informatica ed è conosciuto soprattutto per le sue funzionalità di ambiente di sviluppo integrato (Integrated Development Environment, o IDE) per il linguaggio Java. Tuttavia, nonostante venga impiegato principalmente per lo sviluppo di applicazioni Java, Eclipse è in realtà uno strumento ben più potente. È una piattaforma completa costituita da un *core* e da un insieme di plug-in che ne estendono le funzionalità, e consente al suo interno sia di scrivere applicazioni tradizionali sia di scrivere plug-in in grado di estendere la piattaforma stessa. Tutto ciò è fatto astraendo dal sistema operativo sottostante, sgravando così dagli sviluppatori l'onere di curarsi obbligatoriamente degli aspetti platform-specific dei propri plug-in.
In virtù di questa struttura, lo stesso IDE Java è, in realtà, un insieme di plug-in che estende le funzionalità della piattaforma, così come lo è anche il PDE (Plug-in Development Environment), ossia l'insieme dei tool forniti da Eclipse per sviluppare plug-in per la propria piattaforma.

Un plug-in non è altro che un modulo software in grado di estendere le funzionalità della piattaforma collegandosi ad opportuni *extension points.* A sua volta un plug-in può esporre dei punti di estensione che potranno essere usati da altri plug-in per estenderne ulteriormente le caratteristiche, andando quindi a costituire un sistema modulare in cui anche i componenti fondamentali sono caratterizzati dalla stessa struttura dei moduli aggiuntivi.
La piattaforma, per poter essere in grado di utilizzare le estensioni, deve catalogarle in modo opportuno. Eclipse è per questo dotato di un *plug-in registry* che memorizza le informazioni di ogni plug-in installato. Tramite un'interrogazione al registry è possibile trovare i plug-in correntemente presenti e visualizzare le rispettive estensioni, extension points, servizi utilizzati e dipendenze.
Oltre al plug-in, sono di fondamentale importanza i concetti di *fragment*, *feature* e *update site*.

- Un *fragment* può essere visto come un "frammento" di plug-in, del quale offre alcune funzionalità aggiuntive. I *fragment* sono molto utili per la distribuzione di language pack o di funzionalità specifiche per una piattaforma sottostante. Eclipse rileva i fragment e unisce le loro dichiarazioni a quelle del plug-in a cui fanno riferimento.
- Una *feature* è un'unità di funzionalità e, come tale, raggruppa insieme diversi plug-in che fanno riferimento allo stesso prodotto software. Per esempio una *feature* può contenere diversi plug-in che sono stati pensati per essere installati ed utilizzati insieme. Chiaramente nel caso più semplice una *feature* conterrà un solo plug-in.
- Un *update site* è fondamentalmente costituito da una directory contenente plug-in e feature e da un file (site.xml) che ne descrive il contenuto. È utilizzato dall'Update Manager di Eclipse per scaricare, installare e aggiornare le feature e i plug-in, ed è suddiviso in *categorie* di feature [2].

L'Update Manager di Eclipse può essere aperto da *Help -> Install New Software.*

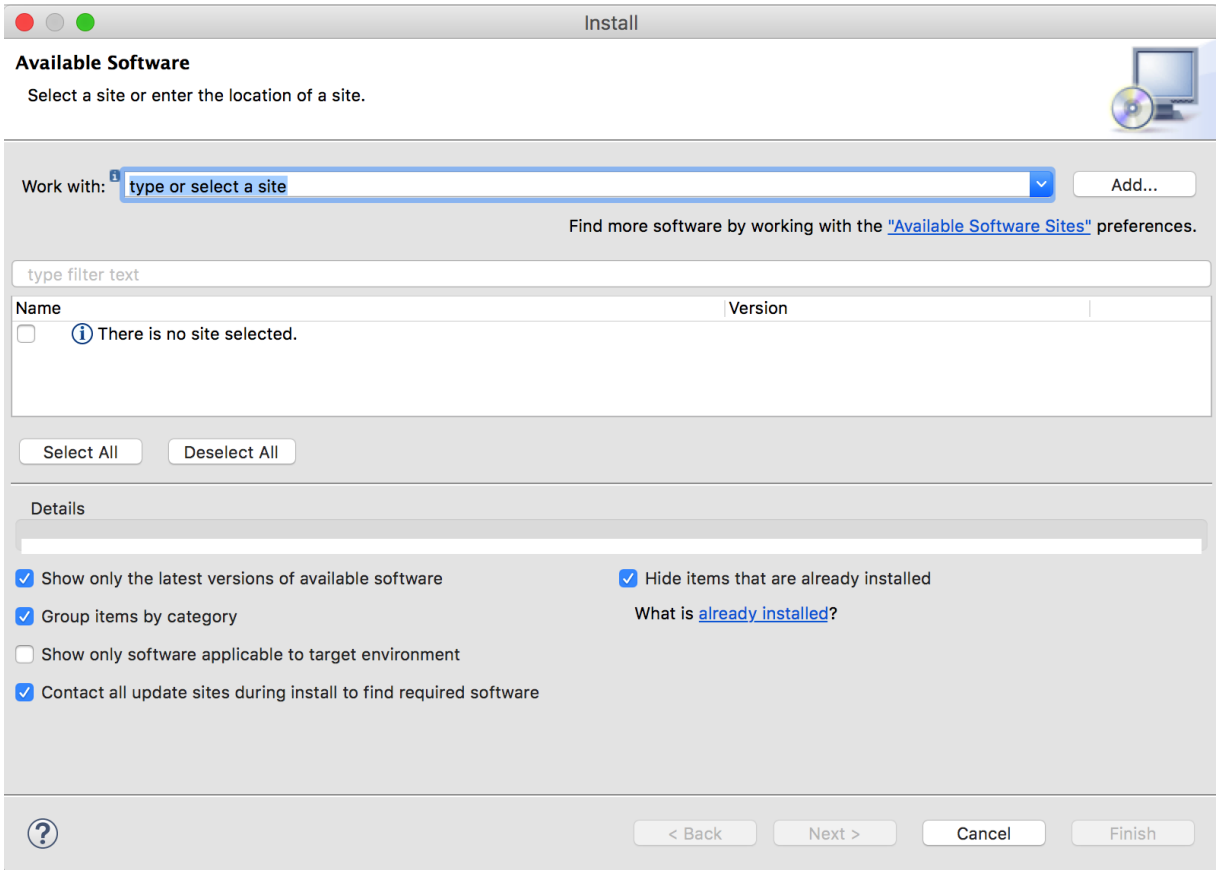


Figura 1. Update Manager in Eclipse Neon per macOS. Sono presenti gli appositi bottoni per inserire un nuovo *update site* e installare le feature desiderate

1.2 Struttura di Eclipse

La struttura fondamentale di Eclipse è riassunta dalla figura seguente:

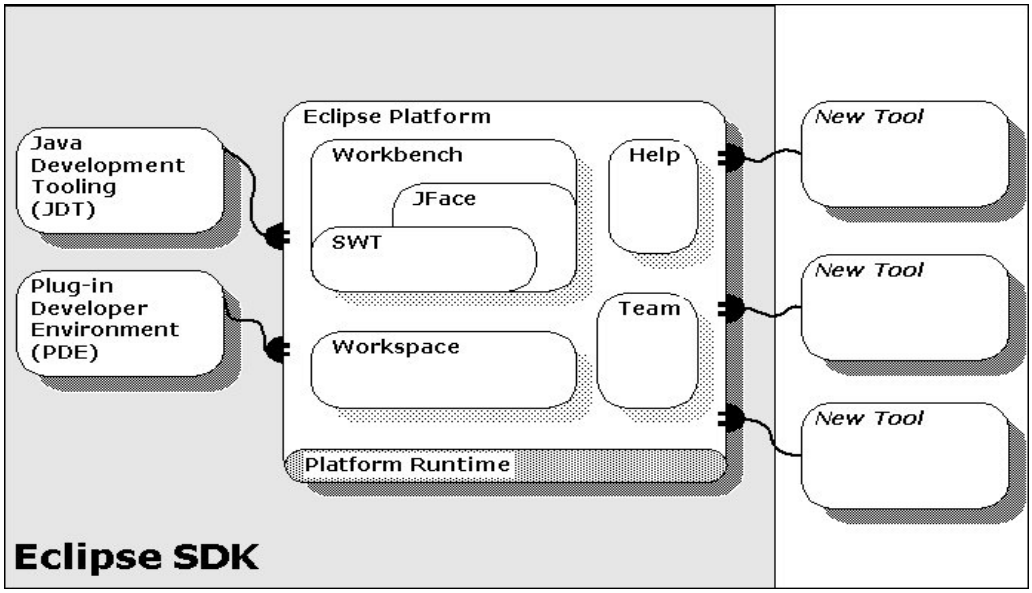


Figura 2. Infografica sulla struttura dell'Eclipse SDK

I principali componenti sono:

- *Platform Runtime*: è il cuore della piattaforma e, in quanto tale, definisce il modello a plug-in. Il suo compito è quello di trovare dinamicamente i plug-in e mantenere le informazioni nel *registry*. Lo scopo principale del Runtime Core è quello di garantire all'utente finale il minor consumo possibile di risorse, caricando i plug-in soltanto al momento del loro effettivo utilizzo, e non all'avvio della piattaforma. Il Platform Runtime è oggi rappresentato da Equinox, del quale si parlerà ulteriormente in fondo a questa sezione.
- *Workspace* (Resource management): definisce le API per gestire le risorse del file system e supporta creazione e gestione dei file, delle directory e dei progetti.
- *Workbench* (UI): è l'interfaccia utente di Eclipse. Il suo scopo è quello di offrire agli sviluppatori un paradigma comune per lo sviluppo della GUI delle proprie applicazioni, fornendo strumenti indipendenti dal sistema operativo per visualizzare, modificare e navigare le risorse gestite dal *Workspace*.
  Il *Workbench* fornisce 3 concetti fondamentali e relativi punti di estensione: *view*, *editor* e *perspective*. Gli *editors* sono strumenti con i quali è possibile aprire e modificare file, associando i vari tipi di file agli *editors* appropriati, ognuno dei quali può avere le proprie personalizzazioni e il proprio set di strumenti. Le *views* consentono invece di visualizzare risorse, per esempio i progetti del Workspace.

Infine le *perspectives* sono insiemi di *view*s ed *editors* che vanno a formare un set specifico di funzionalità, un esempio comune è rappresentato dall'IDE Java.
Il *Workbench* offre anche ulteriori funzionalità, come toolbar, marker (warning ed errori) e bookmark.
I tool per lo sviluppo di un interfaccia utente comune, efficiente e portabile sono *SWT* (Standard Widget Toolkit) e *JFace*, di più alto livello.

- *Help*: è il sistema di aiuto di Eclipse. Fornisce i punti di estensione per definire tool di aiuto e di documentazione.
- *Team*: fornisce supporto al versioning ed allo sviluppo in team.

Tra gli altri supporti forniti da Eclipse va citato anche il *Debug*, per implementare debugger specifici di un linguaggio, e i due tool principali del *Software Development Kit*: *JDT* (Java Development Tools, ossia l'insieme di plug-in che costituisce l'IDE Java) e *PDE* (Plug-in Development Environment, del quale si discuterà nel relativo capitolo). Questi due kit forniscono non solo degli strumenti utilissimi per lo sviluppo di applicazioni Java e di estensioni per la piattaforma, ma rappresentano anche degli ottimi esempi di come costruire un'estensione ben fatta per la piattaforma stessa [2].

Discorrendo più specificamente del *Core*, nel 2004, con l'uscita di Eclipse 3.0, il sistema di gestione dei plug-in è stato sostituito con quello attuale, utilizzando le specifiche OSGi. OSGi (Open Service Gateway Initiative) è un framework Java per lo sviluppo e il deployment di moduli software, e fornisce alla piattaforma Java un modello per il caricamento dinamico di *bundle* in grado di fornire servizi. Le specifiche OSGi definiscono concetti fondamentali per la gestione di un sistema i cui moduli possono essere cercati e caricati a runtime, senza bisogno di un riavvio e gestendo correttamente le dipendenze e il ciclo di vita del codice. I moduli OSGi sono definiti *bundle* e sono collezioni di classi in grado di fornire specifici servizi al sistema, opportunamente descritti dal proprio *manifest* [3] [4].
L'implementazione delle specifiche OSGi presente in Eclipse è chiamata Equinox. Essa non si limita a essere un implementazione certificata delle specifiche del framework OSGi core, ma fornisce anche diversi servizi opzionali. In virtù del suo grande utilizzo, Equinox è oggi una delle principali implementazioni OSGi ed è uno dei modelli di riferimento per l'architettura a plug-in in Java [5].
Le specifiche OSGi trovano oggi una grande diffusione nel mondo Java grazie alla semplicità con cui consentono di gestire i singoli componenti software all'interno di una

piattaforma più grande, e rappresentano la soluzione al problema della modularità, attiva e runtime, nel mondo Java. [1]

Inoltre, data la struttura modulare e il core molto ridotto, Eclipse fornisce agli sviluppatori la possibilità di creare applicazioni basate sulla piattaforma, ma senza il vincolo di dover scaricare (e avviare) l'intero IDE di Eclipse. La piattaforma minima dotata delle funzionalità fondamentali è detta Rich Client Platform (*RCP*), e comprende soltanto il *Core*, l'environment grafico (rappresentato dal *Workbench*, *SWT* e *JFace*) e il supporto all'*Help* [2].

Terminata questa digressione sulla struttura di Eclipse, risulta evidente il discorso iniziale sulla manutenzione del software, in particolar modo all'interno di una piattaforma. Non solo è necessaria una fase di manutenzione ampia e attenta, ma si rende anche indispensabile lo studio costante della piattaforma cui ci si appoggia, per capire come si evolve nel tempo e quali cambiamenti si rendono necessari nel proprio software a causa di questa evoluzione. Basti pensare a una dipendenza da un plug-in o da una funzionalità interna: un refactoring o un cambiamento nella funzionalità interna può invalidare il proprio codice.
Per questi motivi si andranno ad esaminare quelli che sono gli strumenti offerti da Eclipse allo sviluppatore per scrivere e gestire software all'interno della piattaforma.

1.3 Tool di sviluppo: il PDE

Per supportare la propria piattaforma modulare, Eclipse fornisce un apposito strumento per lo sviluppo di plug-in, *features* e *update sites*: il Plug-in Development Environment. Il PDE può essere scaricato a installato come una qualsiasi altra feature mediante l'*Update Manager*, utilizzando l'apposito update site[2], già presente di default nell'UM.
Una volta installato il PDE è possibile creare e importare progetti di plug-in, features e update sites.

In particolare, creando un nuovo *Plug-in project* si apre la prospettiva del PDE e vengono automaticamente generati i file necessari alla piattaforma per riconoscere il plug-in. In particolare il file *MANIFEST.MF*, che descrive il plug-in seguendo le specifiche dei *bundle*

[1] Si invita a visitare *https://www.osgi.org/developer/architecture* per comprendere più a fondo il problema e la soluzione OSGi.

[2] *http://download.eclipse.org/eclipse/updates/4.6*

OSGi. Aprendo questo file all'interno di Eclipse si apre un apposito editor del PDE che mostra informazioni come il nome del plug-in, la sua versione, l'ambiente di esecuzione richiesto, ecc.

Oltre a queste informazioni basiche, un bundle OSGi deve comprendere anche un insieme di informazioni legate alla piattaforma in cui si trova, come per esempio le dipendenze che richiede, le estensioni a cui si aggancia e le funzionalità che offre. L'editor del PDE dovrà quindi essere dotato delle apposite funzionalità in grado di gestire questi aspetti, ed infatti si presenta come segue.

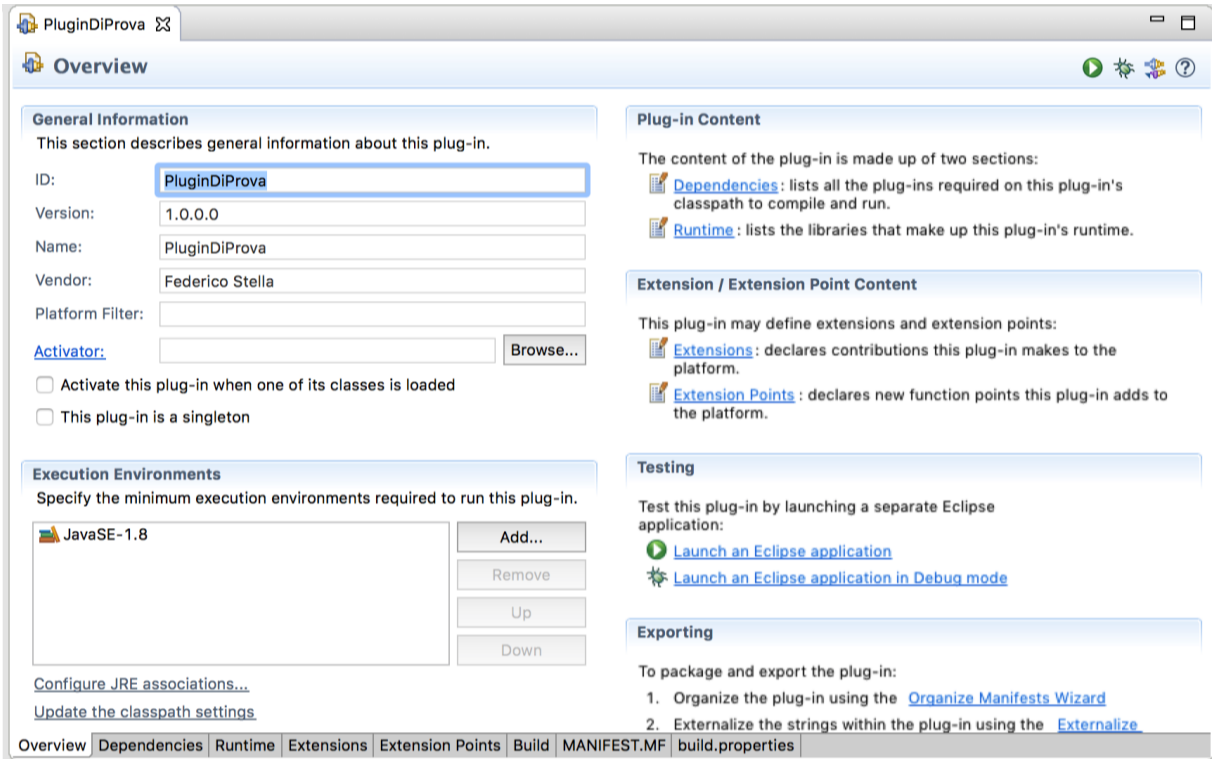


Figura 3. Editor dei file manifest del plug-in

In basso sono presenti diversi tab oltre a quello generico di *Overview*:

- Dependencies: ovvero i plug-in e i pacchetti dai quali dipende il plug-in aperto, e senza i quali questo non può essere installato. A tempo di installazione sarà visualizzato un avviso qualora le dipendenze richieste non siano presenti nel sistema e l'Update Manager non sia stato in grado di risolverle utilizzando gli update sites di cui dispone.

  Le dipendenze possono essere di tipo *required,* che indica i plug-in necessari, e *imported*, che indica i pacchetti necessari, indipendentemente dal plug-in che li offre. È possibile indicare versioni specifiche e algoritmi di match (perfetto, versione minima, versione massima), oltre che l'opzionalità di una determinata dipendenza.

- Runtime: consente di specificare quali pacchetti e funzionalità del plug-in debbano essere esposti all'esterno, ossia quali funzioni saranno salvate nel *registry* per essere disponibili nella piattaforma stessa. Solo funzioni e pacchetti esplicitamente esportati saranno visibili dall'esterno.
- Extensions: permette di aggiungere i *punti di estensione* utilizzati dal plug-in, punti dai quali esso sarà visibile alla piattaforma. Per esempio una *view* dovrà dichiarare la relativa estensione per poter essere inserita nell'elenco delle *views* disponibili in Eclipse.
- Extension Points: sono i punti di estensione dichiarati dal plug-in, sui quali altri plug-in potranno agganciarsi per estenderne le funzionalità.
- Build: contiene le indicazioni di quali file, directory e librerie debbano essere inclusi nelle build binarie e nelle build sorgenti.

L'editor del PDE consente anche di visualizzare direttamente i file che contengono tutte queste informazioni:

- MANIFEST.MF: è il manifest file dei bundle OSGi, contiene le informazioni attraverso le quali il plug-in sarà catalogato nel sistema, tra cui l'ID, la versione, le dipendenze e le funzionalità esposte.
- plugin.xml: file XML che contiene la dichiarazione degli extension points e delle estensioni, con le relative classi che ne fanno uso e le informazioni necessarie per la specifica estensione.
- build.properties: contiene le informazioni di build.

Il PDE fornisce ulteriori viste molto utili, come per esempio il Plug-in Registry, che consente di visualizzare le informazioni immagazzinate nel registry, vedere i plug-in presenti nel sistema, le loro dipendenze, estensioni, punti di estensione, pacchetti esportati, e di sapere se sono attualmente attivi.
Un'altra vista interessante e molto potente è la Console OSGi, visualizzabile come in figura a partire dalla normale vista Console di Eclipse.

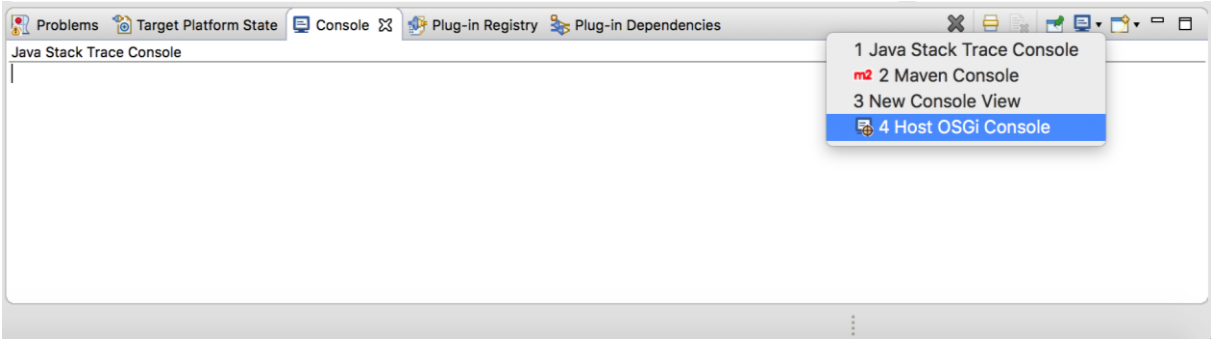


Figura 4. Console di Eclipse, dalla quale è possibile mostrare la Console OSGi.

All'interno della Console OSGi è possibile invocare comandi molto utili, come per esempio 'ss' (short status), che mostra lo stato dei bundle nel sistema e accetta come eventuale argomento un nome con cui filtrare l'output. Un esempio di questo comando è mostrato in figura.

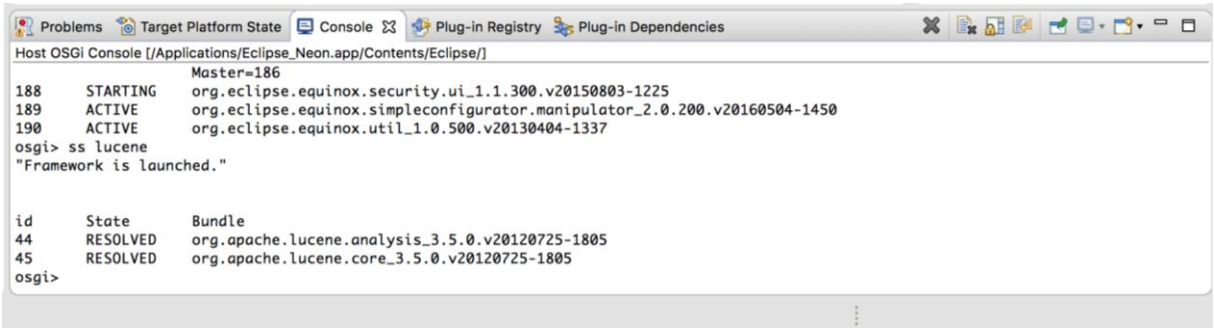


Figura 5. Comando OSGi ss, lanciato sul pacchetto Lucene. Lo stato RESOLVED indica che il plug-in è installato e le sue dipendenze sono soddisfatte, tuttavia non è attualmente in uso.

Gli stati dei plug-in possono essere riassunti come segue (figura 6):

- Installed: il *platform runtime* ha trovato il plug-in.
- Resolved: il plug-in è stato trovato e le sue dipendenze sono state correttamente risolte. Può essere avviato.
- Starting: è in esecuzione il metodo *start* della classe incaricata di gestire il ciclo di vita del plug-in, ossia il *BundleActivator*.
- Active: il plug-in è in esecuzione.
- Stopping: è in esecuzione il metodo *stop* del *BundleActivator*.
- Uninstalled: il plug-in è stato rimosso dal sistema [6].

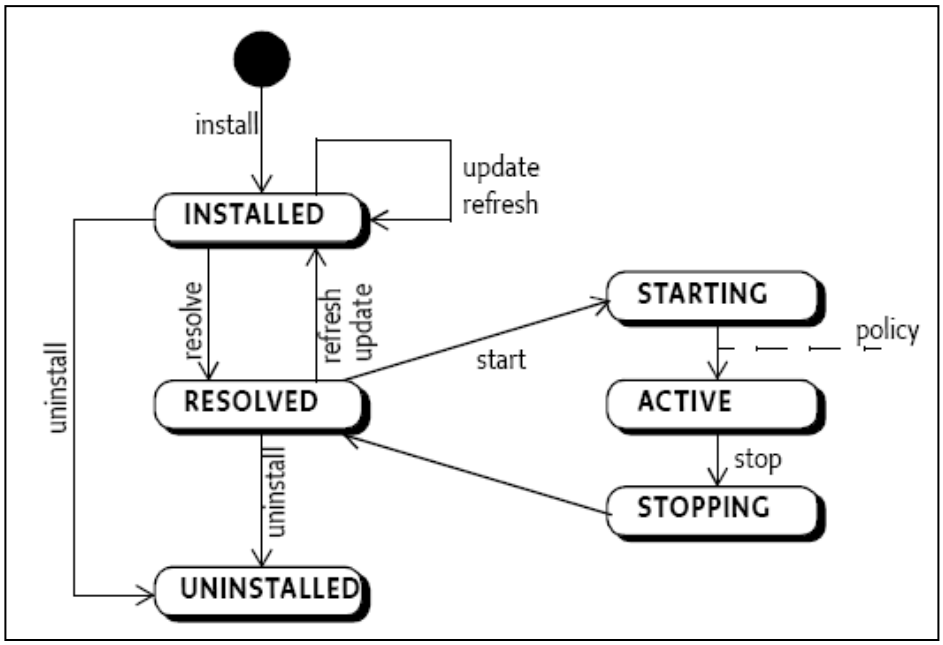


Figura 6. Diagramma di stato del ciclo di vita di un *bundle* OSGi [6]

Introdotta la piattaforma e descritti gli strumenti di sviluppo, si può ora procedere alla presentazione del progetto tuProlog e del suo plug-in, che rappresenta il punto centrale di questa tesi.

2 tuProlog

tuProlog (2p) è un progetto Open Source dell'Università di Bologna rilasciato sotto licenza LGPL. È un motore Prolog leggero, basato su Java e progettato per essere alla base di applicazioni e infrastrutture internet. A questo scopo, le caratteristiche principali di tuProlog sono la facilità di distribuzione, la leggerezza, la configurabilità, l'integrazione con Java e l'interoperabilità.

La facilità di distribuzione di tuProlog è ottenuta grazie a Java e al suo supporto cross-platform. Per questo motivo i requisiti per l'installazione di tuProlog sono costituiti unicamente dalla presenza di una JVM standard. La distribuzione di tuProlog prevede un singolo file JAR autocontenuto, in grado di eseguire il motore e l'interfaccia grafica.

tuProlog è stato progettato con l'obiettivo dell'essenzialità. Il cuore di tuProlog è un piccolo oggetto Java che contiene soltanto le proprietà più essenziali di un motore Prolog. Le caratteristiche richieste saranno aggiunte o rimosse dal motore secondo le necessità contingenti della specifica applicazione.

Controparte all'essenzialità è certamente la configurabilità, infatti per supportare il caricamento statico e dinamico di predicati, funtori e operatori in un motore tuProlog è necessario un meccanismo tanto semplice quanto potente. Tutto ciò è realizzato tramite il concetto di *libreria* tuProlog, la quale può essere definita all'interno della distribuzione standard di tuProlog, oppure essere creata ad hoc dall'utente o sviluppatore tuProlog. Una libreria tuProlog può essere scritta usando il linguaggio Prolog, oppure Java, o anche entrambi, e può essere impiegata staticamente per configurare un motore tuProlog al suo avvio, oppure essere caricata e scaricata dinamicamente durante l'esecuzione.

L'integrazione con Java mira a dare agli sviluppatori la possibilità di creare i componenti di un'applicazione tuProlog usando ad ogni fase il paradigma di programmazione ritenuto più opportuno (quello logico/dichiarativo rappresentato da Prolog o quello imperativo e orientato agli oggetti). Dal lato Prolog, infatti, ogni entità Java può essere rappresentata come un termine Prolog, e come tale può essere utilizzata. Per esempio i pacchetti Java Swing o JDBC possono essere usati direttamente da Prolog, espandendone le caratteristiche con funzionalità grafiche e di accesso ai database. Tutto ciò è possibile grazie alla JavaLibrary o, più generalmente, grazie alla OOLibrary. Nei suoi sviluppi più

recenti, infatti, tuProlog consente l'integrazione anche con qualsiasi linguaggio del framework Microsoft .NET.
Dall'altro lato, ossia Java, è possibile istanziare un motore tuProlog ed usarlo come un comune oggetto Java, integrandolo in un Bean o utilizzandolo in un contesto multi-threaded, secondo le esigenze dell'applicazione. Da Java è inoltre possibile invocare contemporaneamente più motori tuProlog, ognuno configurato con le proprie librerie e basi di conoscenza.

Infine, per quanto riguarda l'interoperabilità, tuProlog supporta l'interazione via TCP/IP e Java RMI, e può essere utilizzato come servizio CORBA. Inoltre tuProlog supporta la coordinazione tuple-based [7].

2.1 Plug-in tuProlog

Visto il grado di integrazione con i linguaggi a oggetti e, nello specifico, con Java, si è pensato di progettare una distribuzione di tuProlog in grado di eseguire all'interno dell'IDE Java più diffuso, ovvero Eclipse. Lo sviluppo è partito nel 2010 con il progetto tuClipse, il quale si è poi evoluto nell'attuale plug-in e non in una versione RCP.
I principali vantaggi offerti dalla piattaforma Eclipse a tuProlog rispetto alla distribuzione stand-alone sono i seguenti:

- Un'interfaccia grafica standard, già molto diffusa e conosciuta, rappresentata dal Workbench di Eclipse.
- Wizard guidati per la creazione di nuovi progetti e teorie tuProlog.
- Notifica degli errori sintattici durante la scrittura, sotto forma di warning ed errori.
- Colorazione della sintassi, personalizzabile dall'apposita estensione alle impostazioni Eclipse.
- View specifiche come la ASTView, che consente di visualizzare la teoria sotto forma di albero navigabile e interattivo.

Per questi motivi si è scelto di procedere con lo sviluppo all'interno di Eclipse. L'idea di un'applicazione RCP è meno convincente rispetto a quella di un plug-in, poiché una versione di tuProlog portabile e leggera esiste già, ed è la distribuzione stand-alone. Perciò il progetto è andato avanti sotto forma di plug-in, dando l'opportunità agli sviluppatori di avere, in un unico ambiente, sia la possibilità di scrivere Prolog, sia la possibilità di scrivere Java, sia la possibilità di interoperare con i due mondi attraverso il plug-in tuProlog e l'apposita libreria tuProlog per Java.

Ora che sono chiari i passi che hanno portato allo sviluppo di un plug-in, si procederà a descrivere le caratteristiche fondamentali di questo, focalizzando i punti necessari per la comprensione dei capitoli successivi.

2.2 Funzionalità del plug-in 2p

Per installare il plug-in tuProlog è necessario inserire nell'Update Manager di Eclipse l'update site corretto, reperibile sul sito di APICe[3]. Una volta inserito e selezionato, l'Update Manager mostrerà le versioni di 2p disponibili. Terminata la procedura, il plug-in risulterà installato nel sistema [8].

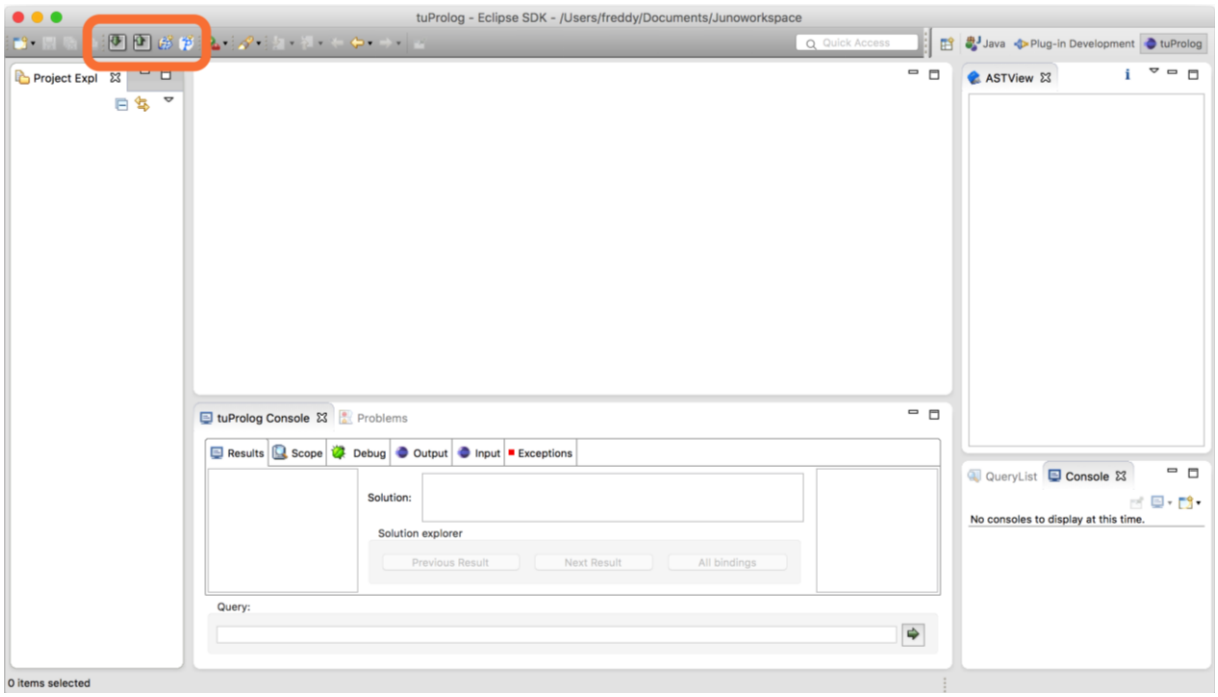


Figura 7. Prospettiva del plug-in tuProlog in Eclipse Juno. Evidenziata la toolbar del plug-in.

Figura 8. Ingrandimento della toolbar tuProlog

In alto, nella toolbar, sono visibili 4 bottoni. Di questi, in particolare, il terzo serve per aprire il wizard per la creazione di un nuovo progetto; il quarto per il wizard per la creazione di una nuova teoria. In basso è possibile vedere la tuProlog Console, view che

[3] *http://apice.unibo.it*

consente di eseguire query Prolog e di mostrarne le informazioni. A destra sono visibili le view QueryList, che mostra le query passate, e ASTView, già discussa.
Una volta creati il progetto e la teoria, quest'ultima viene aperta tramite l'editor del plug-in e diventa possibile modificarla. Qui sotto un semplice esempio di Hello World in Prolog.

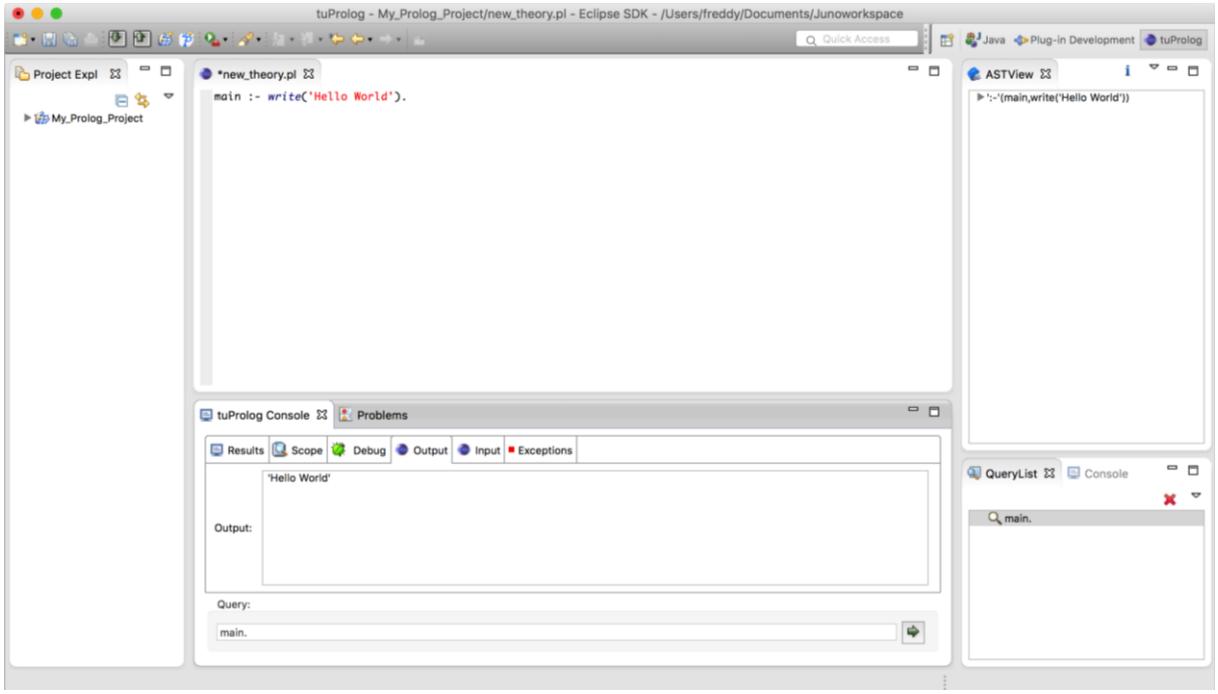


Figura 9. Esempio di Hello World nel plug-in tuProlog

Il tab Output della tuProlog Console mostra l'output delle write del programma. Negli altri tab è possibile visualizzare informazioni più specifiche per il debug, per l'esito della query e per le eccezioni.
Inoltre, grazie all'apposito extension point, è possibile cambiare le impostazioni dell'editor andando nel menù impostazioni di Eclipse. Da qui è possibile indicare il font e modificare i parametri del meccanismo di Syntax Coloring (figura 10 e 11).

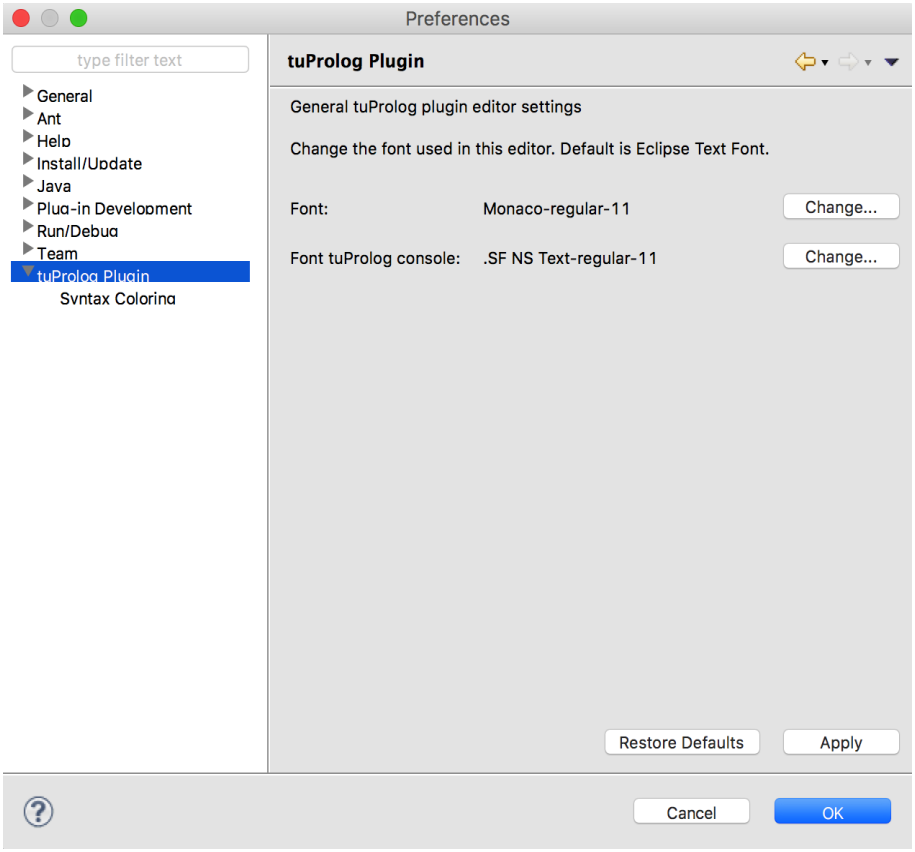


Figura 10. Impostazioni del plug-in.

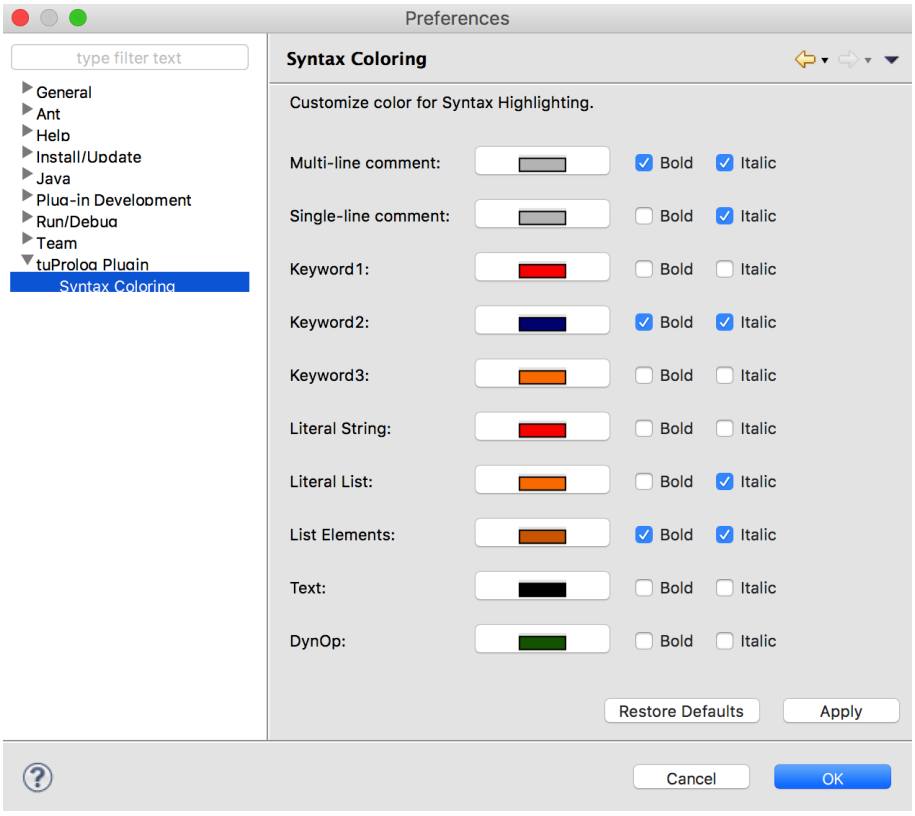


Figura 11. Impostazioni di Syntax Coloring

Di seguito, invece, un esempio più complesso di teoria Prolog per mostrare la vista ASTView.

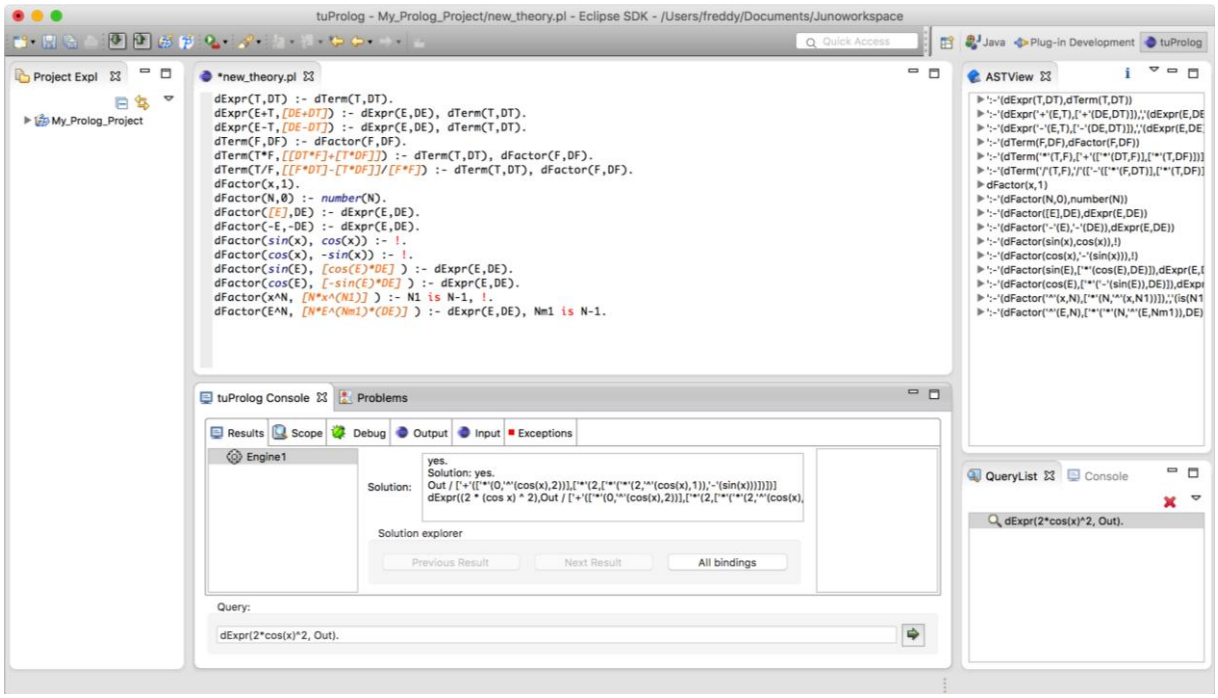


Figura 12. Esempio di teoria Prolog per il calcolo simbolo delle derivate. A destra la ASTView, in basso la query e la sua soluzione.

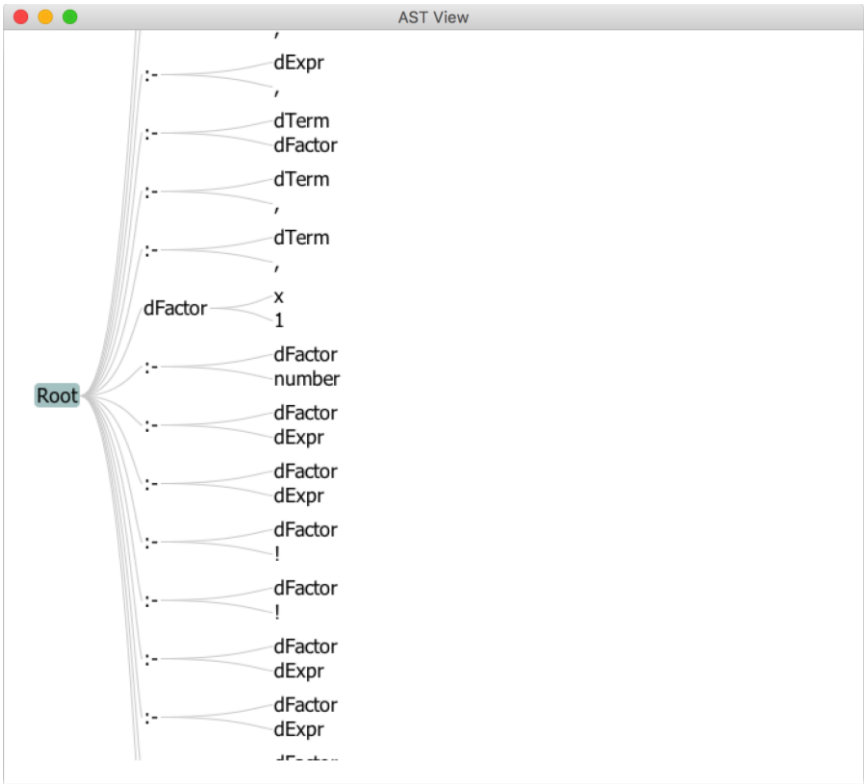


Figura 13. Visualizzazione grafica interattiva della teoria per il calcolo simbolico delle derivate.

2.3 Plug-in tuProlog: situazione attuale

La situazione attuale del plug-in di tuProlog vede la versione 2.7 come ultima versione stabile disponibile, con la 2.9 come feature nascosta nell'update site, quando invece la distribuzione principale ha raggiunto la versione 3.2.1 stabile ed è in continua evoluzione. Tutto ciò è dovuto al fatto che a partire da Eclipse Indigo (v3.7, 2011) ci sono stati dei refactoring della piattaforma Eclipse che hanno portato il plug-in a smettere di funzionare negli Eclipse successivi a Juno (v3.8 e 4.2, 2012).
Nel prossimo capitolo si analizzerà la gestione delle dipendenze in Eclipse per arrivare a determinare quali sono stati i cambiamenti che hanno reso incompatibile il plug-in con le nuove versioni, giungendo poi a risolvere questi problemi e a ripristinare il corretto funzionamento.

3 Analisi

Per comprendere quale sia il problema alla base del mancato funzionamento del plug-in nelle nuove versioni di Eclipse è anzitutto opportuno verificare se l'errore è presente anche a deployment-time. Perciò si rende necessario un esperimento di installazione con l'ultima release stabile disponibile, ovvero Eclipse Neon, e l'ultima versione del plug-in, la 2.9.0.

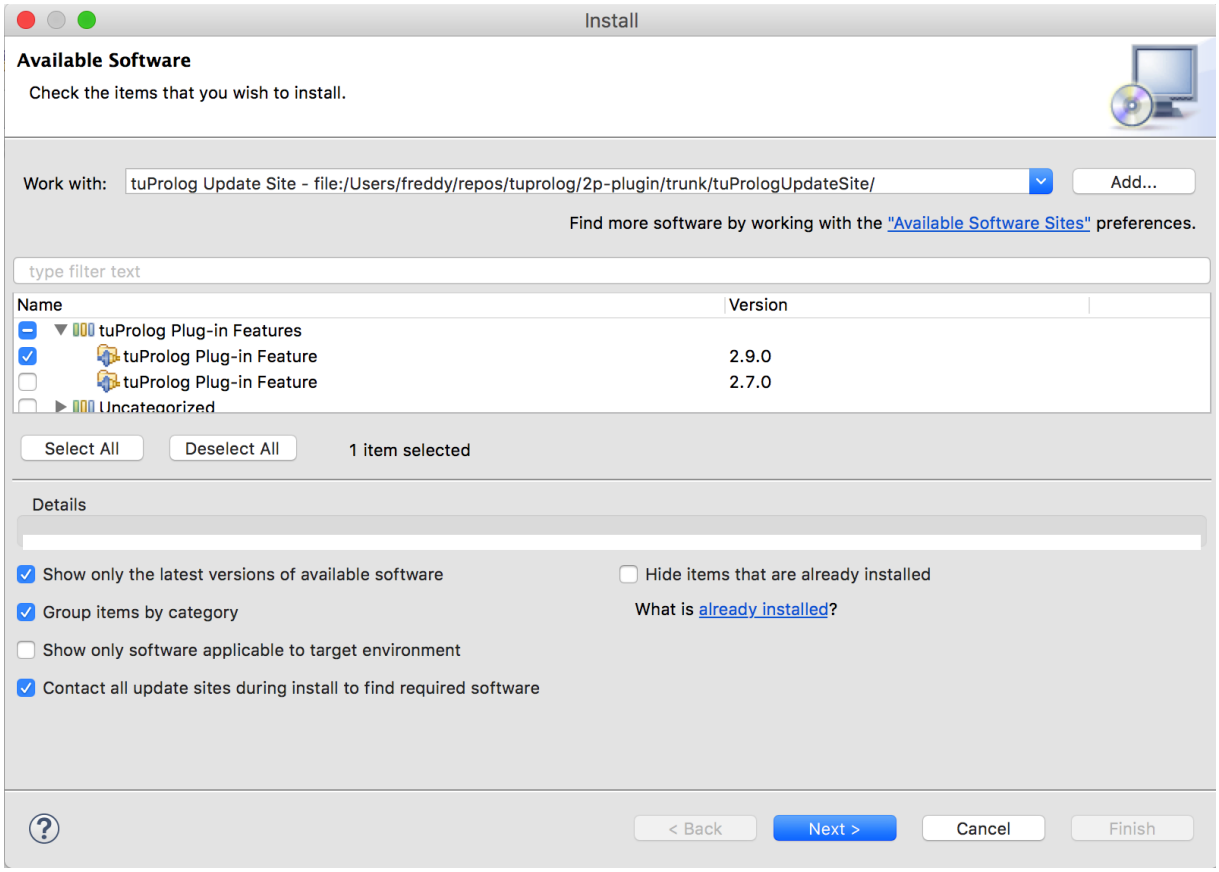


Figura 14. Finestra di selezione della feature da installare: *Help -> Install New Software*

Completando i passi necessari per l'installazione, Eclipse tenta di risolvere le dipendenze e, infine, produce l'errore in figura 15.

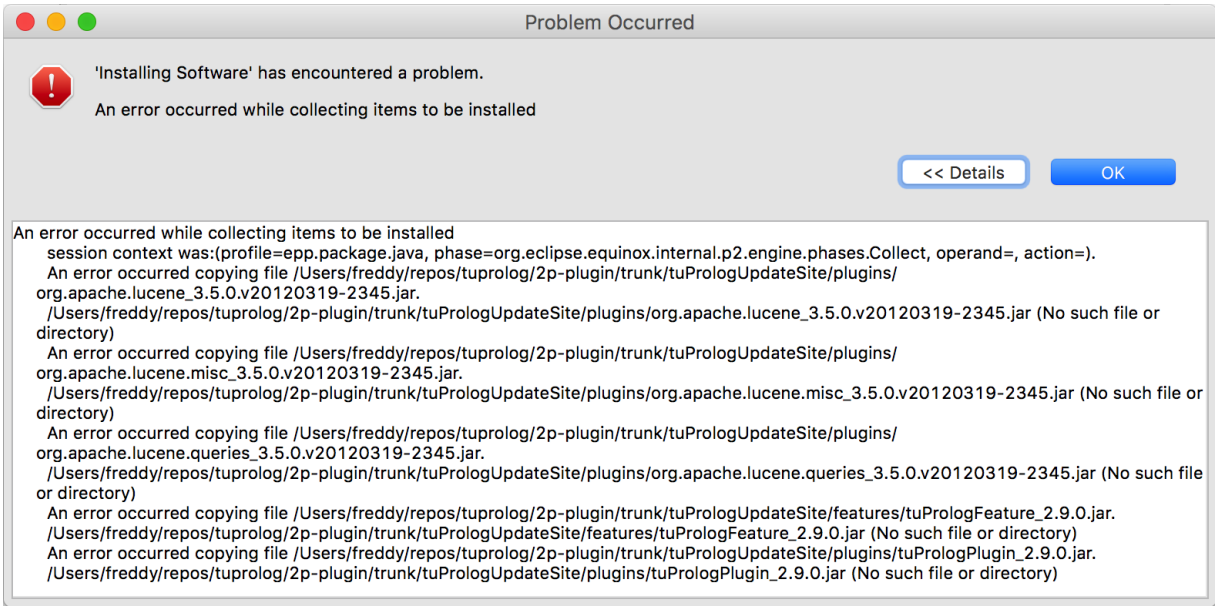


Figura 15. Errore durante l'installazione del plug-in tuProlog in Eclipse Neon

Come si nota dal messaggio di errore, il problema è generato dai JAR del plug-in Lucene, che Eclipse tenta di istallare dalla directory (o dall'update site) contenente il plug-in, ma che non sono presenti.
Il problema, quindi, si presenta già a deployment-time: per poterlo risolvere si procederà ad analizzare Lucene e il suo ruolo all'interno del plug-in e di Eclipse.

3.1 Lucene: funzioni, struttura e impiego all'interno di Eclipse

Lucene è una suite software prodotta da Apache, gratuita e Open Source. Le sue funzionalità principali sono quelle di motore di ricerca testuale, dotato di algoritmi di ricerca molto potenti ed efficienti, scritto completamente in Java e sviluppato a partire dal 1999. Oggi è giunto alla versione 6.5.1 (marzo 2017) [9] [10].
Eclipse fa ampio uso di Lucene nel suo sistema di Help, in particolare con il pacchetto org.eclipse.help.base, che è il prerequisito per la maggior parte delle funzioni dell'Help di Eclipse [2].
Fino ad Eclipse Helios (v3.6, 2010) la versione di Lucene in uso era la 1.9.1. Da Indigo (v3.7, 2011) si è passati alla versione 2.9.1, che porta diverse incompatibilità con la precedente versione, nonché la rimozione di diverse classi da quasi tutti i pacchetti Lucene [2]. Esso è infatti diviso in diversi pacchetti, tra cui *org.apache.lucene*, *org.apache.lucene.core*, *org.apache.lucene.analysis*, *org.apache.lucene.misc* e *org.apache.lucene.queries* [10].

Con Eclipse Kepler (v.4.3, 2013), inoltre, la versione di Lucene è cambiata nuovamente, passando alla 3.5, versione tutt'ora in uso [2]. Questo cambiamento ha portato con sé ulteriori incompatibilità.

Il caso Lucene rappresenta un esempio pratico di come un refactoring della piattaforma, o anche solo di un suo componente, possa invalidare l'organizzazione di un plug-in, e di come un'attenta e continuativa fase di manutenzione sia di fondamentale importanza per garantire che il proprio software continui a funzionare anche in seguito a un refactoring della piattaforma su cui si appoggia.

3.2 Il caso tuProlog

Per capire il problema specifico del plug-in tuProlog è opportuno analizzare il codice e i file manifest e, per farlo, è molto utile importare il progetto del plug-in all'interno di Eclipse, in modo da sfruttare gli editor appositi.
Per importare un *plug-in project* nel PDE di Eclipse, è sufficiente procedere alla normale importazione di progetti già esistenti.
Una volta importato il progetto e impostata la versione corretta del JDK, il plug-in mostra 1 errore di compilazione, situato nel file MANIFEST.MF.

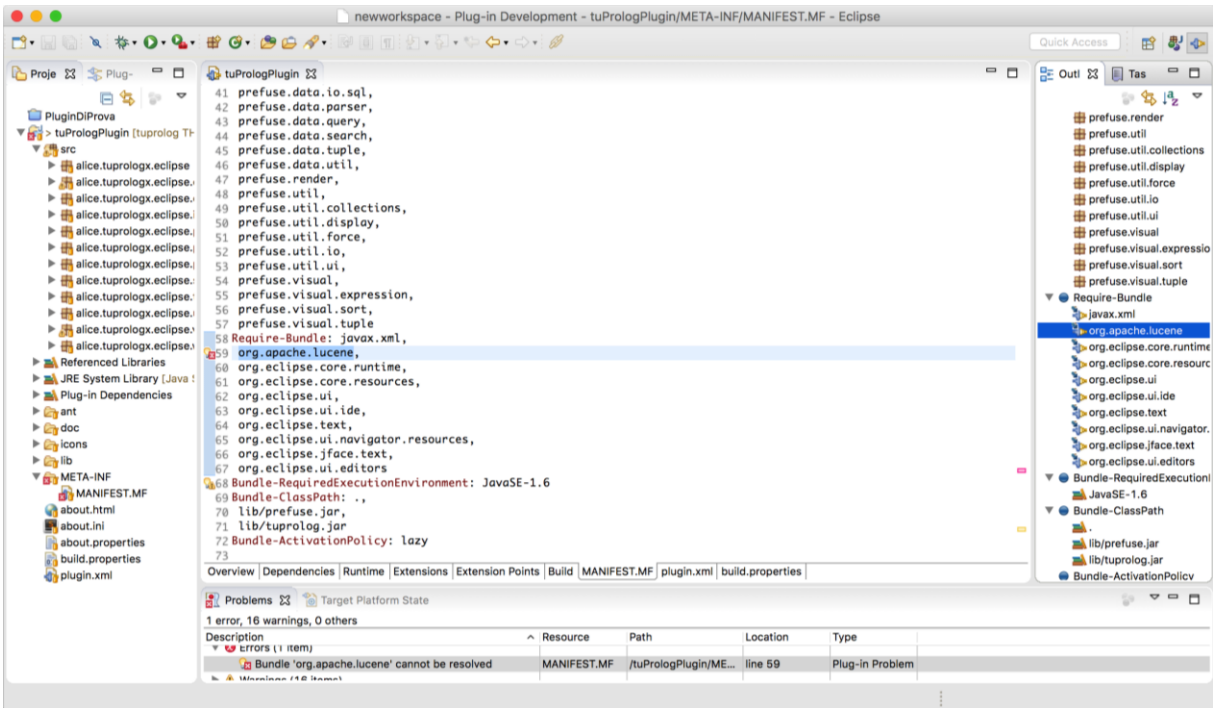


Figura 16. Errore di compilazione dovuto alla mancata risoluzione della dipendenza da Lucene.

Eclipse Neon non riesce a risolvere la dipendenza. Importando lo stesso progetto in release più vecchie di Eclipse, come Indigo e Juno, questo problema non si presenta, indice del fatto che è stato proprio un refactoring della piattaforma a causare il mancato funzionamento del plug-in.

Controllando la documentazione di Lucene e cercando nelle cartelle contenenti i plug-in di Eclipse è possibile notare quali sono i cambiamenti nei pacchetti tra una versione e l'altra della piattaforma. Infatti, nella directory "plugins" di Eclipse (su macOS, *Eclipse.app -> Show Package Contents -> Contents -> Eclipse -> plugins*) sono presenti tutti i plug-in installati, ognuno dei quali è rappresentato da un file JAR o da una subdirectory, a seconda delle impostazioni di deployment. In Eclipse Indigo e Juno sono presenti i plug-in *org.apache.lucene*, *org.apache.lucene.core* e *org.apache.lucene.analysis*, con i relativi pacchetti sorgente. Nelle versioni successive di Eclipse, per esempio Neon, sono presenti soltanto *org.apache.lucene.core* e *org.apache.lucene.analysis*. Per questo motivo il progetto del plug-in mostra errore di compilazione.

Inoltre, analizzando il codice e il funzionamento del plug-in possono essere individuati altri problemi da risolvere:

- La libreria tuProlog in uso nel progetto risulta essere la 3.0, quando invece l'ultima versione stabile disponibile è la 3.2.1. Successive versioni sono attualmente in fase di sviluppo, perciò occorre aggiornare la libreria a quella corrente e, non appena disponibili, alle versioni successive.

- Il plug-in imposta di default la JavaLibrary tra le librerie da caricare *nell'engine* Prolog, tuttavia non supporta la OOLibrary, che è l'attuale libreria impiegata per l'utilizzo di oggetti Java all'interno di tuProlog, mentre la JavaLibrary è deprecata. È necessario estendere il supporto del plug-in per includere anche la OOLibrary e cambiare l'impostazione di default.

Per risolvere questi problemi è necessario disporre di strumenti adeguati, in modo da poter collaudare correttamente il funzionamento del plug-in. Nel prossimo capitolo saranno analizzati gli strumenti utilizzati ai fini di questa tesi.

4 Approcci risolutivi

Per quanto riguarda la risoluzione degli aspetti implementativi vi sono diverse possibilità:

- Esportazione del plug-in e installazione in Eclipse.
  È un processo sicuramente funzionante ma molto dispersivo e macchinoso, risultando in una eccessiva perdita di tempo. Inoltre installare il plug-in nella stessa istanza di Eclipse usata per lo sviluppo è un vincolo troppo forte e poco utile.

- Esportazione del plug-in e copia manuale nella directory "*dropins*" di un'installazione separata di Eclipse.
  La directory *dropins* è presente in Eclipse appositamente per l'installazione rapida e "manuale" dei plug-in, in modo da poterli sottoporre velocemente a test. È una directory definita "*watched*", ossia controllata continuamente dal core di Eclipse per verificare la presenza di nuovi plug-in e inserirli nel sistema. Questo approccio è certamente migliore del precedente, tuttavia presenta ancora dei passi che possono essere evitati, come il processo di esportazione e la copia dei file.

- Utilizzare l'istanza runtime di Eclipse.
  Questo approccio è certamente il migliore ed il più versatile, per questo merita una spiegazione più approfondita nella prossima sezione.

4.1 Istanza runtime di Eclipse

Eclipse mette a disposizione degli sviluppatori la possibilità di avviare, da dentro l'IDE stesso, un'istanza separata di Eclipse, definendo i parametri di configurazione desiderati. Per avviarla è sufficiente selezionare *Run -> Run As -> Eclipse Application*. L'istanza runtime è dotata dei propri argomenti di avvio, del proprio set di plug-in e del proprio Workspace dedicato, mantenuto persistente anche tra un'invocazione e l'altra [2].
È possibile configurare l'istanza runtime andando su *Run -> Run Configurations…* . Da qui è necessario, la prima volta, creare una nuova configurazione di avvio mediante l'apposito menù contestuale sulla voce "*Eclipse Application*" del menù laterale.

Il pannello a destra mostra tutti i parametri di configurazione: particolarmente interessanti sono il tab *Arguments* e il tab *Plug-ins*, che consentono di specificare argomenti come "-clean", per avviare in modo pulito Eclipse, e il set di plug-in da includere nell'istanza, permettendo quindi una gestione semplificata delle dipendenze.
Altra grande comodità dell'istanza runtime di Eclipse è la possibilità di essere avviata in modalità Debug, in modo da consentire l'esecuzione passo passo del codice del plug-in.

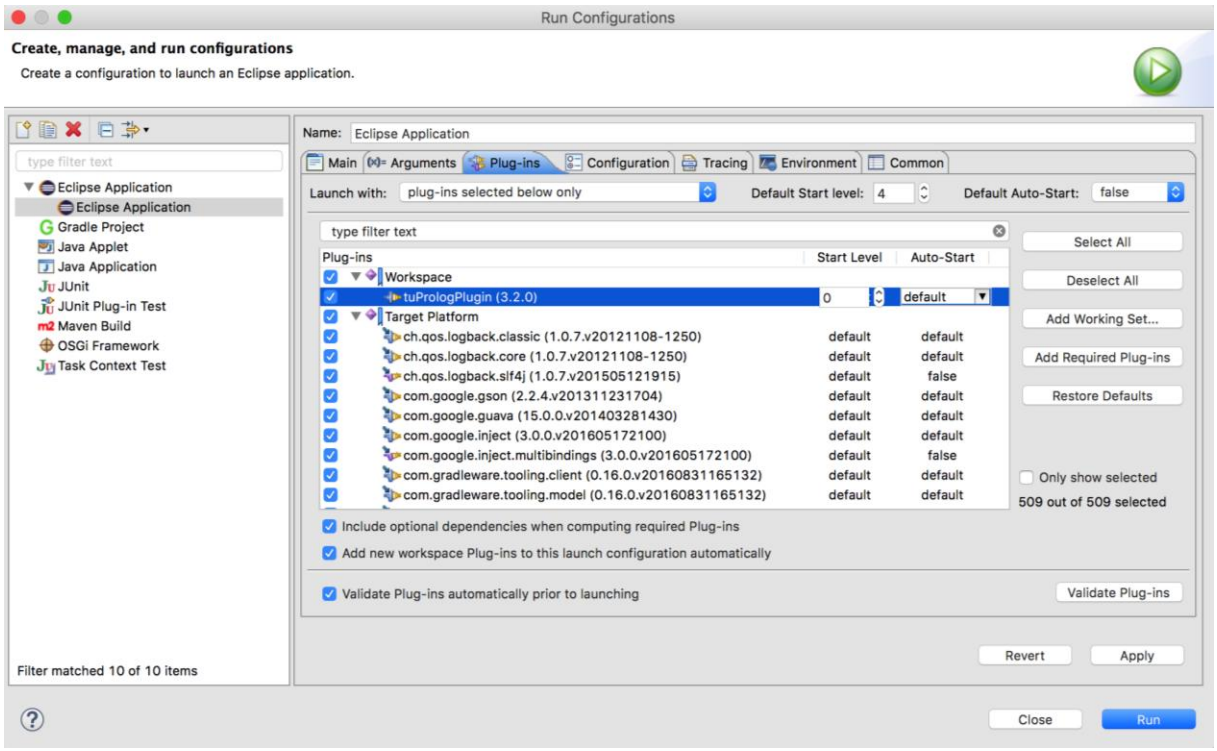


Figura 17. Finestra di configurazione dell'istanza runtime di Eclipse.

Nonostante l'istanza runtime di Eclipse rappresenti uno strumento di indubbia utilità nel collaudare il funzionamento del plug-in e nel bugfixing, esso può presentare qualche problema nel momento in cui si vogliono valutare precisamente gli effetti delle dipendenze. I problemi dell'istanza runtime sono gli stessi presenti anche nelle istanze separate di Eclipse e sono prevalentemente problemi di caching e di pooling.
Eclipse, infatti, mantiene copie locali delle porzioni di update sites già scaricate in precedenza, includendo quindi plug-in e feature. Questo viene fatto per velocizzare una eventuale nuova installazione, senza dover riscaricare i pacchetti, ma presuppone come ipotesi che un plug-in di una determinata versione non possa cambiare nel tempo. Questo è sicuramente vero per una release definitiva, ma non in fase di test, essendo impraticabile cambiare versione ad ogni tentativo. Ovviamente è possibile cancellare a mano le cache di Eclipse, ma, oltre a essere un'operazione macchinosa e lenta, lascia spazio ad errori, come dimenticanze o comportamenti imprevisti.

Il pooling, invece, è utilizzato da Eclipse per creare una directory comune nel quale installare i plug-in, in modo che questi vengano condivisi tra diverse installazioni di Eclipse stesso. Anche questa funzionalità è certamente molto utile all'utente, perché consente di evitare la duplicazione del software, ma risulta una complicazione non necessaria per chi deve collaudare le dipendenze ed ha sulla propria macchina diverse versioni di Eclipse, ognuna col proprio set indipendente di plug-in.

Fermo restando il fatto che è possibile agire manualmente per aggirare queste complicazioni, vi è un modus operandi ben più opportuno per questo tipo di problemi, ovvero le macchine virtuali. Attraverso una macchina virtuale è infatti possibile collaudare il proprio software avendo il totale controllo dei componenti installati, con la possibilità di salvare lo stato e rispristinarlo in ogni momento. In questo modo si eliminano i problemi di caching e di pooling: una macchina virtuale deve possedere solo un'installazione di Eclipse, con le caratteristiche volute, e ogni test sulle dipendenze deve essere seguito da un ripristino dello stato della macchina, il quale riporta la macchina stessa a uno stato iniziale, vergine.

Nella prossima sezione sarà illustrato VirtualBox, lo strumento scelto ai fini di questa tesi per la gestione delle macchine virtuali. Sarà inoltre fornita una spiegazione più dettagliata su come VirtualBox e le macchine virtuali possano risolvere i problemi descritti sopra.

4.2 Oracle VM VirtualBox

VirtualBox è un VMM (Virtual Machine Monitor) gratuito e Open Source sviluppato da Oracle Corporation. VirtualBox rappresenta, grazie alla sua versatilità, compatibilità e gratuità, uno strumento ampiamente utilizzato in ambito informatico per la creazione, la gestione e l'utilizzo di *macchine virtuali*.

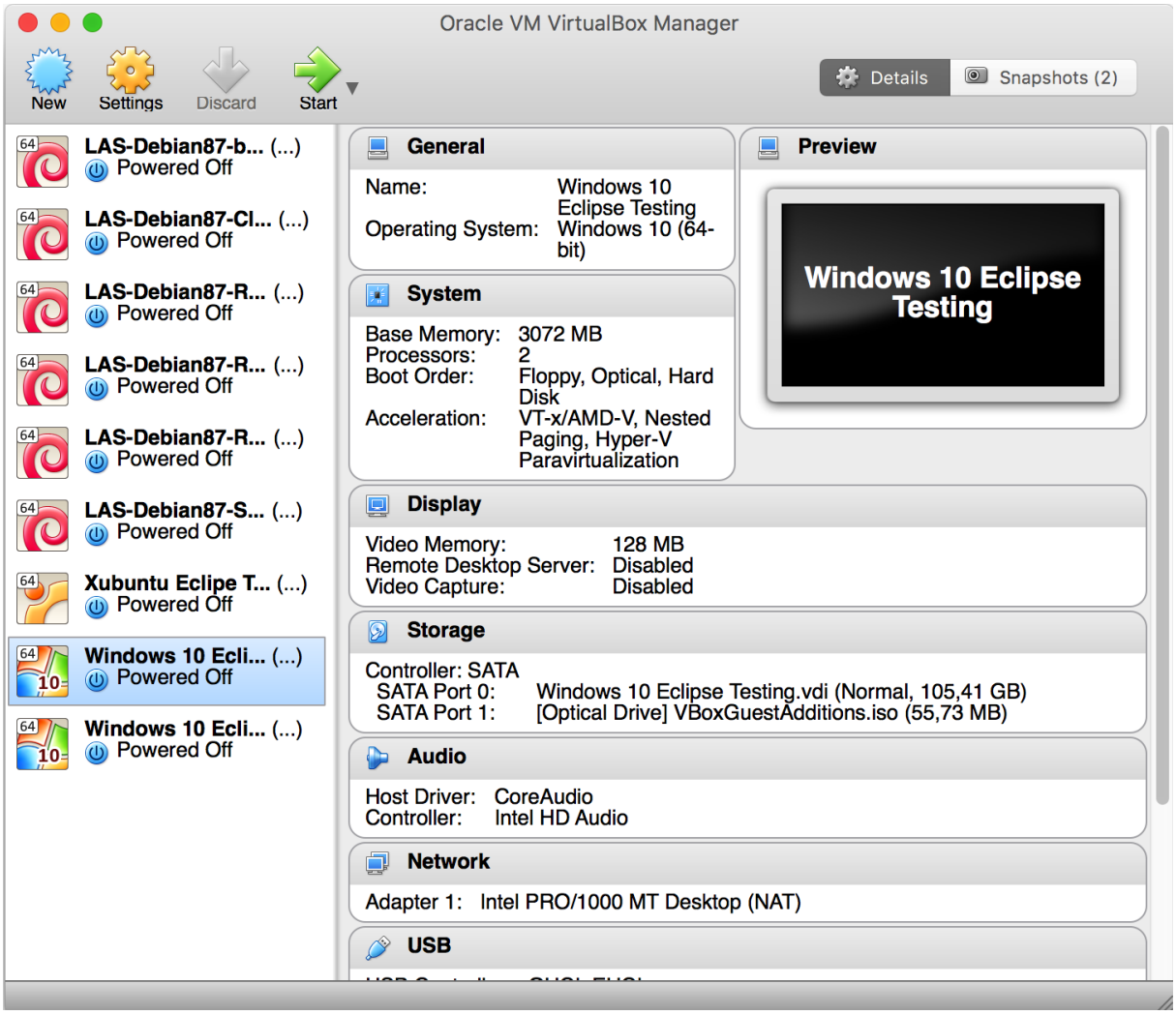


Figura 18. Schermata principale di VirtualBox. Sulla sinistra le macchine virtuali installate, sulla destra i dettagli della macchina selezionata.

Una macchina virtuale è sostanzialmente un'emulazione di una macchina fisica, sulla quale può essere installato un sistema operativo diverso da quello della macchina ospitante (detta *host*) e possono essere effettuate operazioni come su una qualunque macchina. La macchina virtuale prende il nome di *guest* nei confronti della macchina fisica che la ospita.
Attraverso VirtualBox è possibile creare macchine virtuali con a bordo, in particolare, Windows e Linux: in questo modo è semplice fare del testing cross-platform [11] [12].

Scelto quindi VirtualBox come strumento per il testing, si procede alla creazione di una macchina virtuale. Aprendo la procedura guidata da *Machine -> New*, è possibile scegliere il sistema operativo e i parametri di configurazione, come per esempio la RAM dedicata.

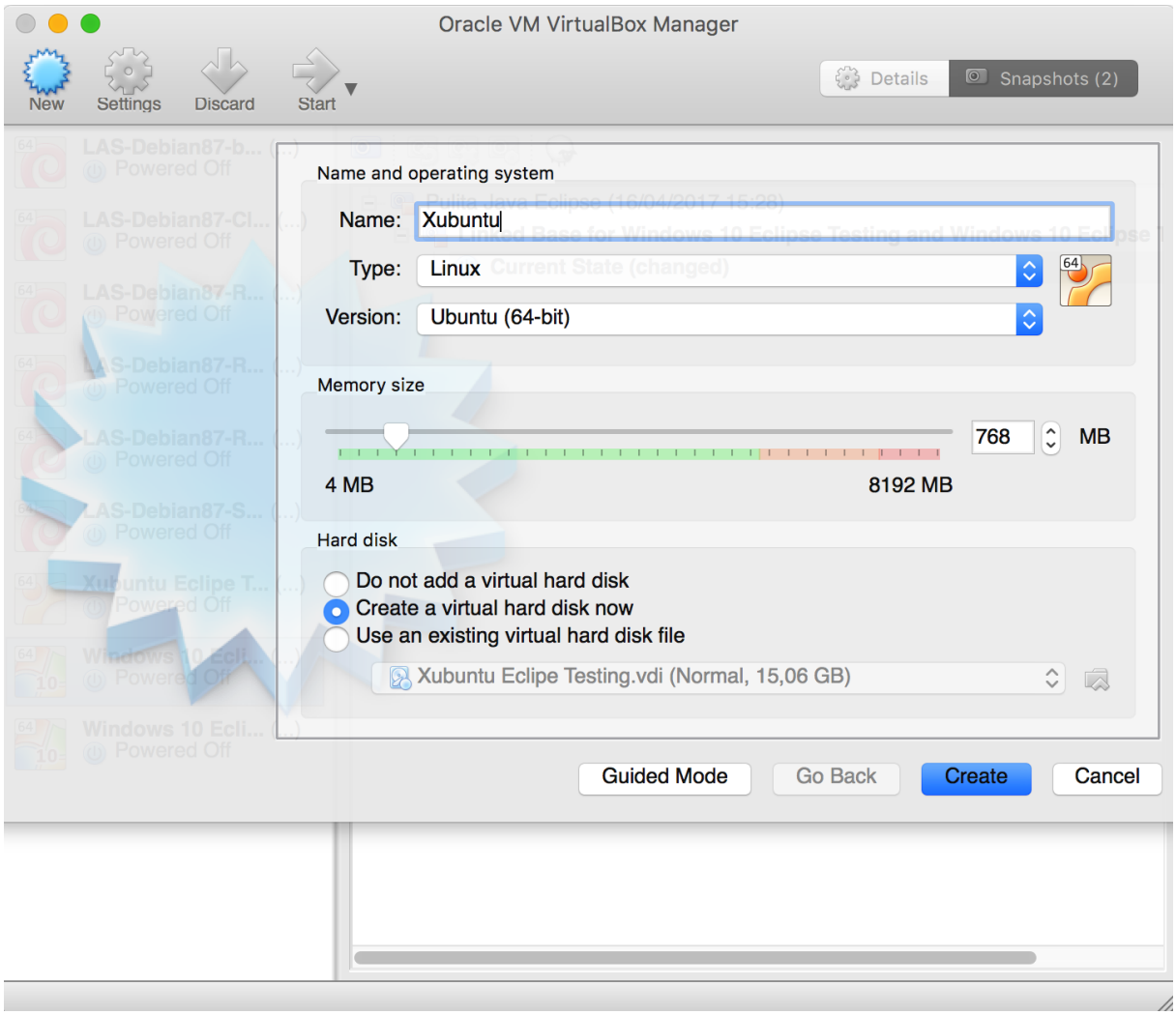


Figura 19. Wizard di creazione di una nuova macchina virtuale.

Scelti i parametri, che potranno essere modificati in modo molto più specifico dopo la creazione della macchina, bisogna procedere alla creazione di un *hard disk virtuale*, che fornirà alla macchina virtuale il supporto di memorizzazione. VirtualBox consente la creazione di hard disk virtuali dinamici, che quindi si comportano come file: occupano su disco soltanto la dimensione necessaria per contenere i file della macchina guest, al posto di occupare staticamente tutto lo spazio indicato come capienza massima.

Terminata la procedura e specificati gli ulteriori parametri, come la configurazione di rete o il numero di core, è possibile avviare la macchina per installare un sistema operativo.

Per poter trasferire file sulla nuova macchina si può installare il pacchetto noto come *VBox Guest Additions* sulla macchina guest, pacchetto che consente sia la copia bidirezionale di file, sia la condivisione degli appunti in memoria; alternativamente, rimangono validi gli strumenti di rete.

Grazie a VirtualBox è quindi possibile avere a disposizione un set di macchine virtuali con cui effettuare il testing, ma rimane ancora vivo il problema della configurazione di Eclipse in queste macchine. Infatti, nonostante il problema del pooling sia aggirato, rimane il caching, che viene comunque effettuato da Eclipse anche sulle macchine virtuali. Per ovviare a questo, VirtualBox mette a disposizione uno strumento molto potente, rappresentato dagli *snapshots*.
Uno *snapshot* è un'istantanea scattata allo stato della macchina virtuale, salvata in memoria attraverso un sistema di *dischi differenziali*. In questo modo è possibile tornare in qualunque momento allo stato in cui ci si trovava al momento della creazione degli snapshots.
Per crearne uno e visualizzare quelli presenti, è sufficiente selezionare una macchina e selezionare quindi il tab *Snapshots* sul pannello di destra.

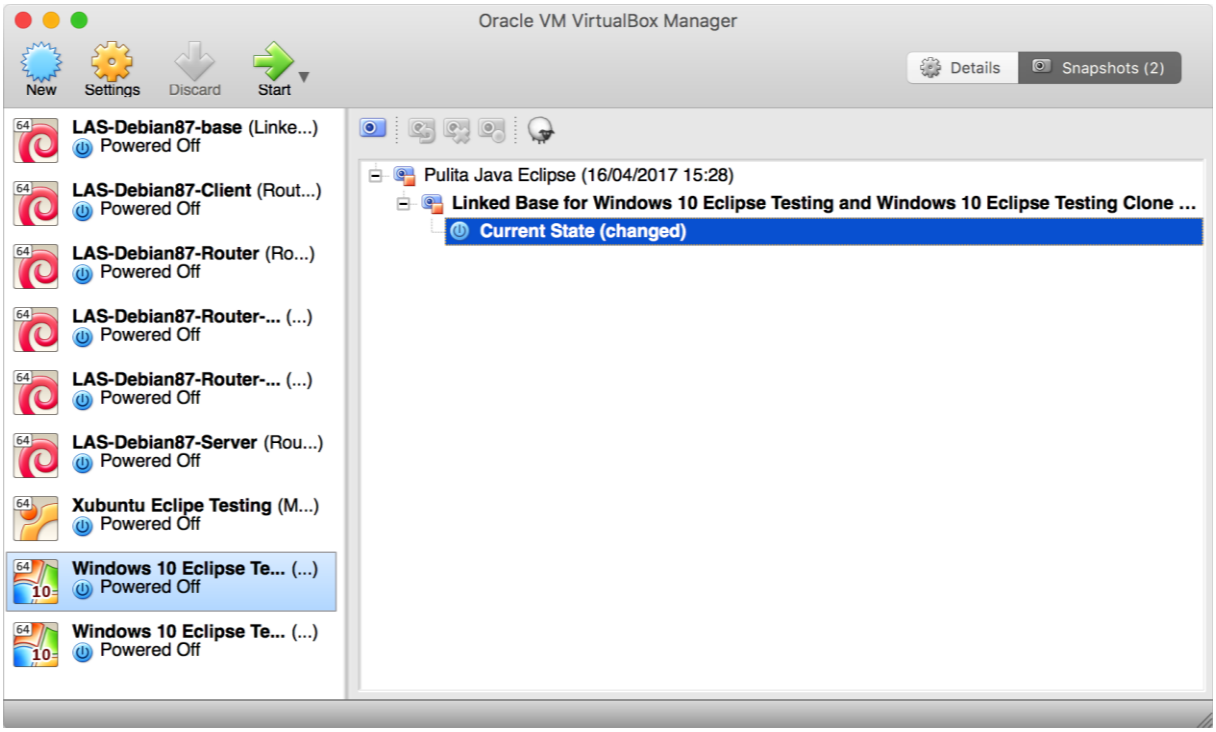


Figura 20. Tab degli Snapshots di VirtualBox.

Ai fini di questa tesi, è utile installare nella macchina guest sia il Java Development Kit, sia l'Eclipse SDK, per avere una configurazione base da cui partire. A questo punto si può scattare uno snapshot e si può procedere a effettuare i test.

Un meccanismo molto simile a quello degli snapshots è impiegato da VirtualBox per creare i *Linked Clones*, ossia cloni di macchine virtuali sempre basati sui dischi differenziali, in modo da minimizzare l'utilizzo di spazio su disco e il tempo di

installazione. Grazie ai *Linked Clones* è possibile collaudare contemporaneamente configurazioni diverse su macchine diverse, in modo semplice e veloce.

Con l'utilizzo di VirtualBox e dei suoi strumenti si è quindi in grado di avere la certezza di lavorare su una macchina pulita, con il vantaggio di ridurre notevolmente i tempi rispetto alla gestione di macchine fisiche, e di poter effettuare contemporaneamente test su più configurazioni. Si può quindi procedere alla revisione architetturale del plug-in, al fine di risolverne i problemi.

5 Revisione architetturale

Il primo problema da affrontare è necessariamente quello della dipendenza da Lucene, poiché senza risolverlo non è possibile avviare il plug-in e quindi occuparsi delle altre questioni.
Per la risoluzione del problema di dipendenze si profilano 3 possibilità distinte.

1. **Copia manuale dei pacchetti**
   All'interno della directory di installazione di Eclipse, è presente una cartella *plugins* che contiene anche i pacchetti di Lucene. Il plug-in necessita di *org.apache.lucene* e *org.apache.lucene.core* in versione 2.9, o in alternativa, a causa del refactoring, di *org.apache.lucene*, *org.apache.lucene.core*, *analysis*, *queries* e *misc* in versione 3.5[4]. È quindi sufficiente copiare nella directory *plugins* i pacchetti mancanti e riavviare Eclipse per far sì che il plug-in tuProlog possa risolvere le proprie dipendenze.
   Questa soluzione è certamente funzionante, ma è impensabile richiedere all'utente di agire manualmente sulle directory di installazione di Eclipse per poter installare un plug-in. Per tale motivo questa soluzione deve essere scartata.

2. **Uso di Orbit**
   Orbit è un progetto nato all'interno di Eclipse e consiste in una repository dalla quale è possibile scaricare bundle gratuiti di terze parti. Il suo scopo è quello di facilitare il lavoro degli sviluppatori che usano questi bundle, fornendo un luogo comune nel quale poterli trovare. Soltanto le librerie approvate da Eclipse Foundation possono essere inserite all'interno di Orbit, e di queste vengono mantenute anche le vecchie versioni, in modo da supportare più utenti possibile [13].
   La repository di Orbit relativa a Eclipse Neon[5] contiene tutti i pacchetti di Lucene in versione 3.5, perciò può essere impiegata per risolvere il problema del plug-in tuProlog.
   In particolare, importando il progetto della feature di tuProlog e aprendo il file *feature.xml,* è possibile modificare le informazioni relative alla feature. Aprendo il tab *Information* e la sezione *Sites to Visit* si possono inserire gli update sites da

[4] Tutti i pacchetti Lucene citati sono preceduti nel nome da "org.apache.lucene.", omesso per mantenere la leggibilità del testo

[5] *http://download.eclipse.org/tools/orbit/downloads/drops/R20160520211859/repository*

includere durante la ricerca di aggiornamenti e durante la fase di risoluzione delle dipendenze.

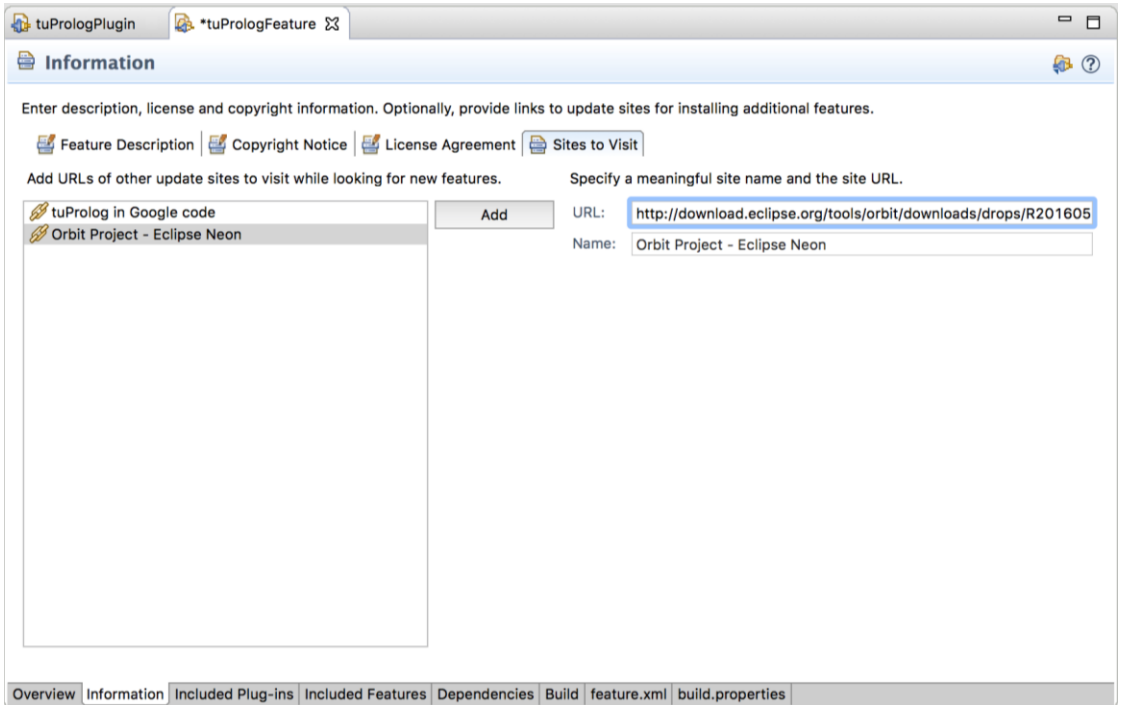


Figura 21. Editor del file feature.xml, in cui è stata inserita la repository di Orbit.

Questa soluzione è decisamente migliore della precedente, tuttavia mantiene nel progetto del plug-in il riferimento a una libreria esterna che, come già successo in passato, può essere soggetta a cambiamenti.

3. **Eliminazione della dipendenza**
   Analizzando attentamente il codice del plug-in, risulta evidente che i pacchetti Lucene non sono affatto usati. La loro presenza all'interno del codice può essere giustificata con la presenza, nel progetto originario, di una parte di Help facente uso diretto del motore Lucene; oppure con il fatto che, in passato, vecchie release di Eclipse avessero la tendenza a inserire la dipendenza da Lucene nei progetti con funzionalità relative all'Help di Eclipse stesso (org.eclipse.help.base).
   Una volta stabilito che Lucene non è necessario, è possibile procedere con la sua rimozione modificando il file MANIFEST.MF, oppure rimuovendolo dal tab Dependencies dell'editor di Eclipse.

Evidentemente la terza soluzione risulta la migliore, poiché evita il download di Lucene ed elimina la possibilità che, in futuro, un refactoring dei pacchetti Lucene possa invalidare nuovamente il plug-in: è stata quindi adottata.

5.1 Aggiornamento libreria tuProlog

Una volta ripristinato il funzionamento del plug-in si può procedere all'aggiornamento della libreria core di tuProlog.
Per farlo è sufficiente sostituire la libreria nella directory *lib* del progetto con la versione attuale della stessa, ed aggiungere al *classpath* il JAR appena importato.
Poiché i requisiti della libreria tuProlog 3.2.1 prevedono Java SE 1.8 come environment minimo, è necessario aggiornare il file MANIFEST.MF aggiungendo la seguente riga:

```
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
```

In questo modo qualunque versione di Java successiva o equivalente alla 1.8 sarà accettata.
Si noti che la sostituzione della libreria tuProlog è possibile poiché le interfacce sono rimaste invariate tra la versione 2.9 e 3.2.1, come ci si aspetta in un prodotto esplicitamente progettato nell'ottica "design for change". Le versioni di tuProlog che si trovano ora in fase di sviluppo presentano un leggero refactoring delle classi all'interno dei pacchetti, nonostante le interfacce rimangano invariate. Per questo in futuro si renderà necessario aggiornare gli *import* all'interno del plug-in per supportare le nuove librerie.

5.2 Supporto alla OOLibrary

Come già discusso in precedenza, la libreria chiamata JavaLibrary è ora deprecata nella release principale di tuProlog, in favore della OOLibrary.
Il plug-in, alla creazione di un progetto, crea un motore tuProlog e gli associa alcune teorie fondamentali, tra cui la JavaLibrary. Inoltre, andando su *Properties* dal menù contestuale di un progetto tuProlog è possibile accedere al menù "*Engines Management*", che consente di specificare le librerie associate ad ogni motore. Tentando di aggiungere una nuova libreria si apre una finestra di dialogo in cui è possibile digitare il nome della libreria desiderata, tuttavia la OOLibrary è indicata come non esistente.

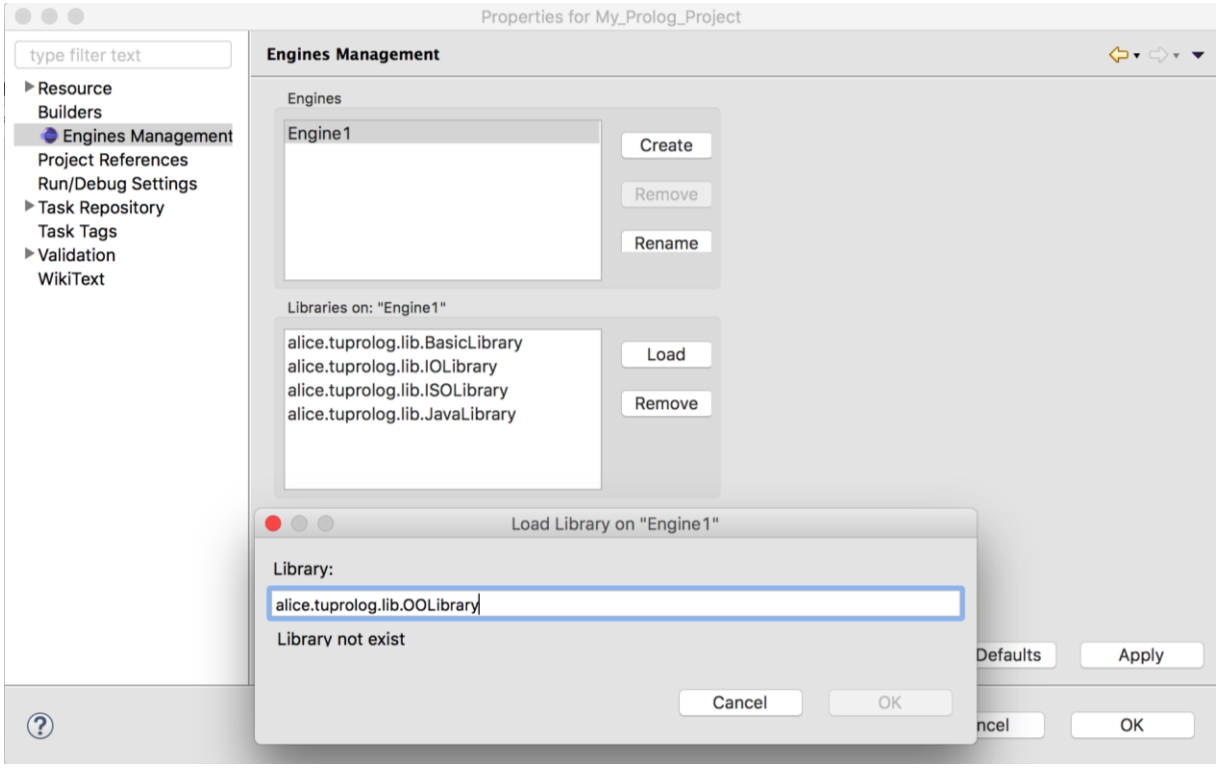


Figura 22. Finestra di gestione dei motori tuProlog.

Controllando le classi del plug-in che si occupano della gestione delle impostazioni e dei motori tuProlog si può notare che il problema è in realtà una dicotomia tra due questioni diverse: la gestione delle librerie di default, che prevede l'inserimento automatico della JavaLibrary; e l'aggiunta di nuove librerie, che non accetta la OOLibrary.
La prima questione vede il coinvolgimento delle seguenti classi:

- *alice.tuprologx.eclipse.properties.EnginesManagement*, che si occupa gestire i motori e, in particolare, di applicare i default quando viene premuto l'apposito bottone.

- *alice.tuprologx.eclipse.properties.PropertyManager*, che si occupa di inizializzare il Workspace e di creare e gestire correttamente i motori all'apertura di un progetto tuProlog.

- *alice.tuprologx.eclipse.wizards.PrologProjectWizardPage*, classe che gestisce il wizard di creazione di un nuovo progetto e imposta le librerie di default nel motore.

Per modificare questi comportamenti è sufficiente sostituire la stringa che rappresenta la JavaLibrary con quella per la OOLibrary.

La seconda questione, invece, coinvolge soltanto la classe *alice.tuprologx.eclipse.properties.LibraryValidator*. In questa classe vengono effettuati i controlli per determinare se la libreria scelta è valida oppure no, sarà in particolare sufficiente aggiungere la condizione per cui anche la OOLibrary sarà ritenuta valida.

5.3 Aggiornamento feature e update site

Ripristinato il normale funzionamento del plug-in e aggiunto il supporto agli ultimi sviluppi della release principale di tuProlog, è necessario aggiornare la feature per includere questo plug-in e non più quello vecchio, sostituendo anche il numero di versione per dare la possibilità all'Update Manager di riconoscere il nuovo plug-in e segnalare un aggiornamento agli utenti che ancora usano una versione precedente.
Per l'aggiornamento della feature è sufficiente importare il progetto della feature stessa nel PDE e modificare il file *feature.xml* con l'apposito editor. Bisogna rimuovere il vecchio plug-in (versione 2.9) e aggiungere quello nuovo, il quale eredita le prime due cifre del numero di versione da quello della libreria tuProlog utilizzata, ovvero 3.2. Sincronizzando i numeri di versione tra il plug-in e la feature, quest'ultima assume il numero corrispondente al plug-in principale da cui è composta, in questo caso l'unico presente.

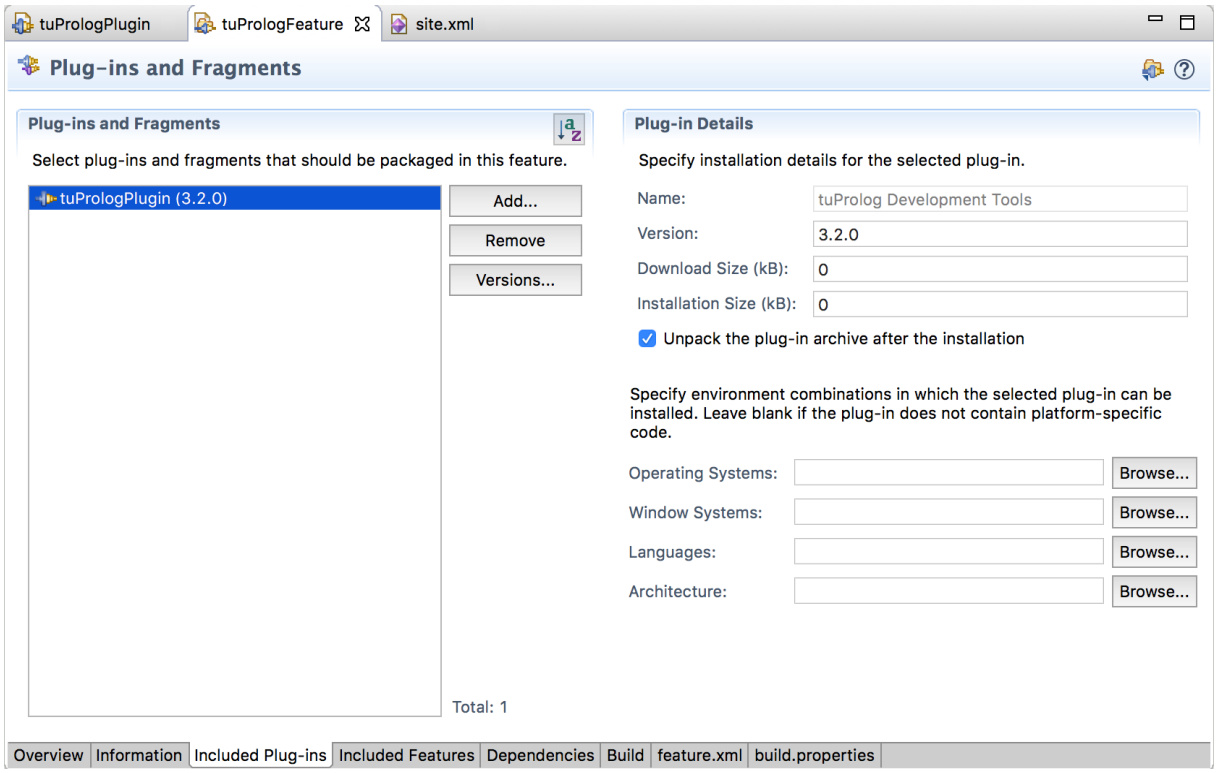
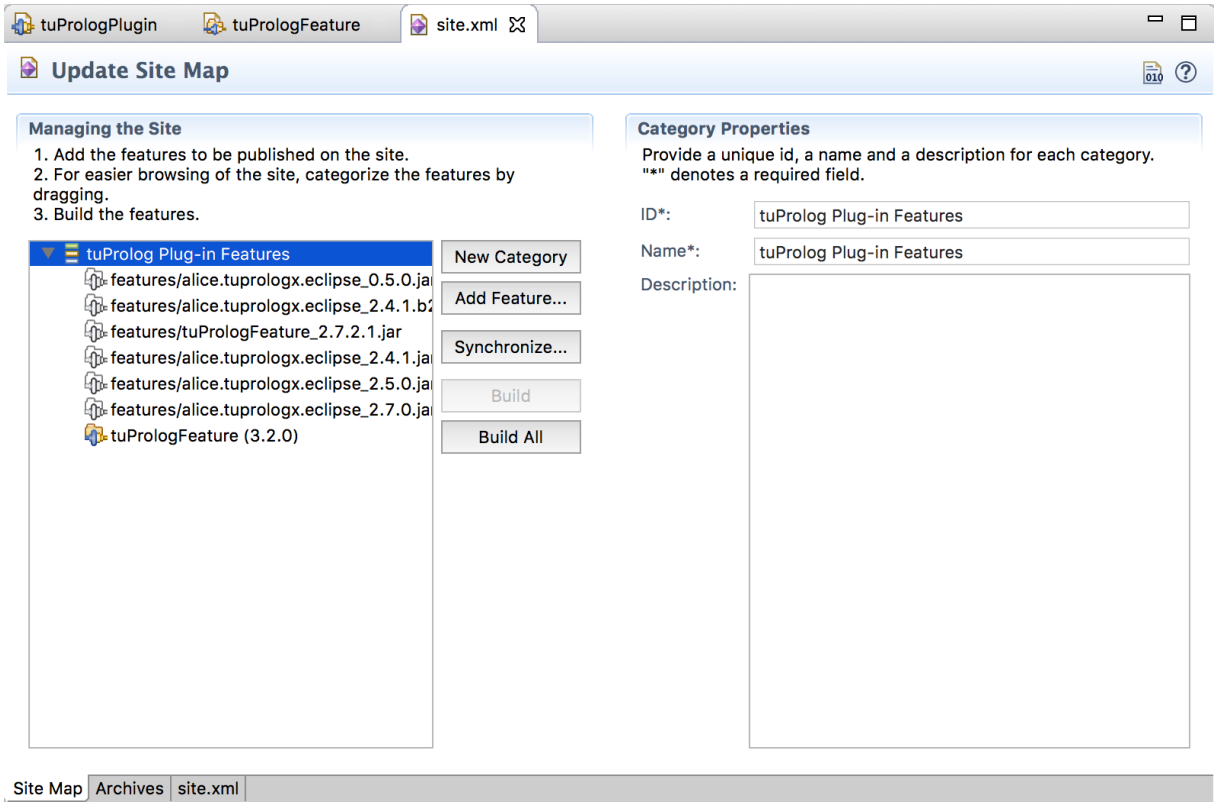


Figura 23. Editor della feature, aggiunto il plug-in in versione 3.2

Bisogna quindi aggiornare l'update site: per farlo è necessario importare l'update site nel PDE ed aggiungere le seguenti righe al file site.xml.

```
25    <feature url="features/tuPrologFeature_3.2.0.jar" id="tuPrologFeature" version="3.2.0">
26       <category name="tuProlog Plug-in Features"/>
27    </feature>
```

L'aggiunta di queste righe produce una voce relativa alla nuova versione del plug-in nella categoria "tuProlog Plug-in Features", dando all'update site la struttura seguente:



A questo punto si può procedere con la build, dopodiché l'update site è pronto per essere aggiornato sul serve

6 Conclusioni e sviluppi futuri

L'obiettivo della tesi, ovvero rendere il plug-in tuProlog nuovamente funzionante, trovando e risolvendo il problema che ne ha impedito il corretto funzionamento a partire da Eclipse Kepler, è stato raggiunto in primo luogo analizzando la piattaforma e le dipendenze del plug-in e in secondo luogo mettendo a punto un processo atto a garantire un collaudo corretto delle soluzioni individuate in seguito all'analisi.
La risoluzione del problema principale, ossia la dipendenza da Lucene, per quanto semplice di per sé, non è affatto semplice da collaudare a causa delle cache di Eclipse e dei suoi meccanismi di pooling, i quali portano molto facilmente a risultati inaffidabili durante i tentativi di installazione ed esecuzione. Il maggior impegno è quindi richiesto per capire come risolvere il problema che impedisce un testing corretto e per capire come trovare una procedura che lo possa garantire. Individuata la procedura e isolati i problemi di caching e pooling, l'eliminazione di una dipendenza risulta molto semplice, così come l'aggiornamento della libreria tuProlog e il cambiamento nei requisiti Java.
Potendo quindi avviare il plug-in nelle ultime versioni di Eclipse, si rende evidente la necessità di estendere il supporto alle librerie tuProlog in modo da supportare la recente OOLibrary, modificando anche i default per escludere la deprecata JavaLibrary.

Dopo un utilizzo più estensivo del plug-in e dopo un'attenta consultazione dei changelog di Eclipse sono state individuate, infine, ulteriori questioni da risolvere in futuro:

- Il plug-in, per la visualizzazione della propria toolbar, fa uso di extension points quali *action* e *actionSet,* che rappresentano gli strumenti messi a disposizione da Eclipse per creare comandi e menu all'interno del Workbench. Attualmente gli *actionSets* risultano deprecati e saranno in futuro rimossi dalla piattaforma [2]. Sarà quindi necessario adeguare la struttura del plug-in per adottare le soluzioni più recenti all'aggiunta di toolbar al Workbench.

- Se si fa uso di oggetti Java in una libreria tuProlog, attraverso le funzionalità offerte dalla JavaLibrary (od OOLibrary), vengono generati warning relativi all'uso di questi oggetti, poiché il *parser* Prolog non ne riconosce i termini, così come non riconosce i metodi Java invocati su questi oggetti. Il problema è stato risolto nella versione attuale di tuProlog stand-alone, ma è ancora presente nel plug-in.
  In particolare, la gestione di warning ed errori è molto differente tra il plug-in e la

versione stand-alone: quest'ultima mostra i messaggi di errore soltanto attraverso la pressione del bottone "View Debug Information", in seguito a un tentativo di esecuzione non riuscito; mentre la prima mostra tutto durante la digitazione, sfruttando gli strumenti di Eclipse e indicando anche file di provenienza e numero di riga dell'errore.

L'utilizzo, purché corretto, di oggetti Java non comporta la mancata esecuzione del codice, perciò non genera alcun tipo di warning in tuProlog stand-alone. Viceversa nel plug-in si analizzano tutti i termini della teoria creando un dizionario già in fase di scrittura della teoria stessa: se un termine non viene riconosciuto viene segnalato un warning di possibile errore. Questo è il caso degli operatori della OOLibrary, ma anche dell'invocazione di metodi Java tramite l'operatore '<-'. Il primo caso si può risolvere, per esempio, caricando le librerie e aggiungendo dinamicamente gli operatori al dizionario. Il secondo caso invece è molto più complesso, poiché si dovrebbe verificare l'esistenza lato Java dei metodi utilizzati nella teoria.

Sarà quindi necessario decidere se mantenere le funzionalità di segnalazione errori in tempo reale fornite da Eclipse, o se invece allineare il comportamento del plug-in a quello della versione stand-alone, visualizzando le informazioni di debug solo dopo un tentativo di esecuzione. È anche possibile adottare quest'ultimo approccio per i soli warning, mantenendo l'utile controllo errori a tempo di sviluppo.

7 Bibliografia

[1] Eclipse (software) - Wikipedia. [Online]. https://en.wikipedia.org/wiki/Eclipse_(software)

[2] Eclipse Documentation. [Online]. http://help.eclipse.org

[3] OSGi - Wikipedia. [Online]. https://en.wikipedia.org/wiki/OSGi

[4] OSGi Alliance. [Online]. https://www.osgi.org

[5] Equinox (OSGi) - Wikipedia. [Online]. https://en.wikipedia.org/wiki/Equinox_(OSGi)

[6] OSGi Service Platform - Core Specification. [Online]. https://osgi.org/download/r4v43/osgi.core-4.3.0.pdf

[7] TuProlog - Wikipedia. [Online]. https://en.wikipedia.org/wiki/TuProlog

[8] tuProlog - APICe. [Online]. http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome

[9] Apache Lucene - Wikipedia. [Online]. https://en.wikipedia.org/wiki/Apache_Lucene

[10] Apache Lucene Core. [Online]. https://lucene.apache.org/core

[11] Oracle VM VirtualBox. [Online]. https://www.virtualbox.org/

[12] VirtualBox - Wikipedia. [Online]. https://en.wikipedia.org/wiki/VirtualBox