UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Corso di Laurea in Informatica

TESI DI LAUREA TRIENNALE

"Sviluppo di un plugin Eclipse per la generazione di codice EPSILON per il refactoring di modelli software guidato dalle performance"

Candidato

Davide Di Gironimo

Relatore

Prof. Vittorio Cortellessa

_____

Co-Relatore

Dott. Ric. Davide Arcelli

_____

Anno Accademico 2015/2016

Indice

1. Introduzione

Nello sviluppo di un sistema software sono molteplici i requisiti che vanno considerati durante la fase di progettazione. Negli ultimi anni, i requisiti di *performance* hanno acquisito molta importanza poiché, se considerati sin dalle prime fasi del ciclo di vita, consentono di ottenere un prodotto di qualità abbassando i costi dovuti ad eventuali re-implementazioni successive al rilascio del prodotto stesso. Diventa dunque molto importante introdurre approcci mirati a soddisfare i requisiti prestazionali sin dalle prime fasi del ciclo di vita del software. La Software Performance Engineering (SPE) fornisce un approccio allo sviluppo di sistemi software mirato a raggiungere proprio tale obiettivo. Attraverso la raccolta di dati relativi al sistema software ed alla piattaforma su cui esso viene eseguito, la loro rappresentazione attraverso opportuni modelli, e la loro manipolazione attraverso appositi metodi, SPE consente di identificare le parti del sistema software che potrebbero avere prestazioni inaccettabili e di applicare *refactoring* volti a soddisfare i requisiti prestazionali, prima che il sistema stesso venga implementato. SPE aiuta quindi il rilascio di prodotti di qualità dal punto di vista delle prestazioni e, lavorando su modelli, permette un'effettiva riduzione dei costi di sviluppo, soprattutto perché permette di soddisfare, come già detto in precedenza, i requisiti di performance prima e durante l'implementazione. Per caratterizzare le parti prestazionalmente problematiche del sistema ed i refactoring che più probabilmente risolvono tali problematiche, in SPE ci si basa spesso sul concetto di *performance antipattern*, nato proprio con lo scopo di descrivere cattive pratiche di modellazione e proporre soluzioni per queste ultime.

L'obiettivo di questa tesi è di realizzare la visione d'insieme del lavoro in [1], in cui viene proposto un approccio per l'identificazione e la rimozione di performance antipattern in un unico ambiente di supporto. La piattaforma *Epsilon* [2] fornisce un ecosistema di linguaggi task-specific, interpreti e tools a supporto di svariate attività che si possono svolgere su modelli. Tra i vari linguaggi di tale piattaforma, Epsilon Validation Language (EVL) consente di definire proprietà che i modelli devono soddisfare e le azioni da intraprendere quando tali proprietà non sono soddisfatte, il tutto in un unico ambiente integrato in *Eclipse* [3]. EVL risulta dunque il candidato ideale per lo scopo di questa tesi.

Infatti, nel contesto SPE di quest'ultima, le proprietà che si vogliono soddisfare sono le condizioni di identificazioni dei performance antipattern, mentre le azioni da intraprendere quando le suddette proprietà non sono soddisfatte sono le azioni atte a rimuovere i performance antipattern.

Conformemente a tale contesto, il prodotto finale di questa tesi si manifesta sotto forma di un plugin Eclipse che, di fatto, consente di "progettare" e in ultima istanza di generare automaticamente, dei file EVL in cui le proprietà da verificare rappresentano le condizioni di individuazione di performance antipattern e le azioni da intraprendere sono volte alla loro rimozione.

Pertanto, il presente lavoro di tesi supporta la scrittura di file EVL, facilitandone la strutturazione e producendone automaticamente il codice, fornendo dunque un valido aiuto verso la realizzazione della visione fornita in [1].

2. Background

2.1 Cenni su UML ed Eclipse

La modellazione è l'operazione di design dell'applicazione software che si svolge prima di codificare l'applicazione stessa, e può essere di aiuto in quanto permette di lavorare ad un livello di astrazione più elevato rispetto a quello implementativo. Lo Unified Modelling Language (UML) è un linguaggio di modellazione che consente di definire, visualizzare e documentare i modelli di sistemi software. Tale linguaggio fornisce la possibilità di:

- Modellare un dominio.
- Scrivere i requisiti di un sistema software.
- Descrivere l'architettura del sistema.
- Descrivere struttura e comportamento di un sistema.
- Documentare un'applicazione.
- Generare automaticamente un'implementazione parziale di un sistema.

La sintassi astratta del linguaggio UML è definita attraverso un metamodello, mentre la sintassi concreta è definita tramite un insieme di elementi grafici aventi semantiche ben definite e attraverso i quali è possibile costruire *diagrammi*, anche fra loro correlati. Tali definizioni costituiscono lo standard UML, definito in principio, nel 1996, sotto il patronato dell'OMG (Object Management Group) che tuttora lo gestisce. Il linguaggio nacque con l'intento di riunire approcci definiti precedentemente, raccogliendo i punti di forza di ognuno, definendo uno standard unificato.

Eclipse è un noto Integrated Development Environment (IDE), fondato da un consorzio no-profit di società, noto come *Eclipse Foundation*, nato nel 2001. Fin dalla sua nascita, Eclipse è stato apprezzato in tutti gli ambiti della produzione software. Esso viene distribuito secondo la *Eclipse Public License*, che segue i principi fondamentali del software open source. Lo sviluppo di Eclipse è portato avanti sia da singoli programmatori che da grandi compagnie. La struttura della piattaforma Eclipse è basata sui plug-in, attraverso cui i membri della comunità (programmatori, progettisti, architetti software) possono contribuire all'estensione della piattaforma stessa, permettendo ai team di sviluppo di concentrarsi sull'espansione di singole aree, in base alle loro competenze

e/o necessità. La piattaforma permette dunque di integrare i tools sviluppati nel workbench di Eclipse per far sì che gli utenti finali possano usufruirne. I plug-in sviluppati possono essere inseriti nel workbench in determinati punti, tramite ***extension points***. La piattaforma Eclipse stessa, in realtà è costruita su plug-in disposti su livelli: ogni plug-in definisce ***extensions*** agli ***extensions points*** dei plug-in del livello sottostante e a sua volta fornisce la possibilità di essere esteso, dai livelli superiori, nella stessa maniera.

2.2 Piattaforma Epsilon

Epsilon [2], Extensible Platform of Integrated Languages for mOdel maNagment, è una piattaforma che definisce un insieme di linguaggi, interpreti di questi ultimi, e tool per operare su modelli conformi a metamodelli descritti in Ecore [3]. Ad esempio, tramite la piattaforma Epsilon è possibile verificare proprietà su modelli, confrontarli, trasformarli e generare codice a partire da essi. Ad oggi, Epsilon fornisce i seguenti linguaggi:

- Epsilon Object Language (EOL)
- Epsilon Validation Language (EVL)
- Epsilon Pattern Language (EPL)
- Epsilon Wizard Language (EWL)
- Epsilon Transformation Language (ETL)
- Epsilon Comparison Language (ECL)
- Epsilon Merging Language (EML)
- Epsilon Generation Language (EGL)

Come accennato, per ogni linguaggio Epsilon fornisce dei tool di supporto basati su Eclipse e un interprete che esegue i programmi scritti in questi linguaggi.

EOL rappresenta il linguaggio-cardine di Epsilon, fornendo le funzioni “primitive” per operare su modelli EMF. Oltre ad essere il linguaggio su cui tutti gli altri task-specific language si basano, può anche essere visto come un linguaggio a se per la gestione dei modelli.

Il linguaggio EVL consente di validare modelli rispetto a determinate proprietà/vincoli che possono essere specificati attraverso la sintassi del linguaggio; inoltre, EVL consente

anche di specificare programmaticamente azioni da intraprendere quando le proprietà specificate non sono verificate sul modello in analisi. Le specifiche di validazione EVL sono organizzate in moduli, ciascuno chiamato *EvlModule*, che contiene le operazioni definite dall'utente e può contenere import a *EolLibraryModule* e ad altri *EvlModule*. Un modulo Evl, può contenere un insieme di *invarianti* (*Invariant*) raggruppate in base al *contesto* (*context*) cui si riferiscono. Un *context* definisce il tipo di istanze su cui verranno eseguite le *invariant* che contiene. Ogni contesto può avere una *guardia* (*guard*) che limita la sua applicabilità ad un insieme ristretto di istanze del tipo definito. Le invarianti sono composte da un nome e da un corpo. Per le invarianti è anche possibile specificare un *messaggio* (*message*) da restituire in console quando quell'invariante non è verificata sul modello. Inoltre, per permettere di correggere in maniera semi-automatica gli elementi che non hanno rispettato dei vincoli, le invarianti possono contenere uno o più *fix*, ognuno composto da un *titolo* (*title*) ed un blocco imperativo, detto *do*, in cui vengono specificate le azioni da intraprendere sugli elementi del modello che non verifica l'invariante.

EVL è il linguaggio di riferimento per questo lavoro di tesi. Infatti, il plugin sviluppato per questa tesi permette di "modellare" (tramite una struttura ad albero) e generare un file EVL.

EPL, *Epsilon Pattern Language*, è un linguaggio che permette di stabilire la corrispondenza dei modelli con i pattern definiti.

EWL (*Epsilon Wizard Language*) invece permette la modifica di determinati elementi dei modelli in esame, tramite l'interazione dell'utente, con un interfaccia grafica.

2.3 Performance Anti-patterns

Un *design pattern* è una soluzione comune ad un problema che occorre in differenti contesti e fornisce una soluzione generale che può essere adattata ad un determinato contesto. I pattern fanno uso delle conoscenze derivanti dall'esperienza e che hanno portato alla definizione di "best practices" nella progettazione di sistemi software e rendono possibile il riuso di tali conoscenze e la loro applicazione nella progettazione di differenti tipi di sistemi software. Negli anni, gli sviluppatori di sistemi software si sono trovati di fronte più e più volte gli stessi problemi. Ad ogni correzione di un problema

però, nella nuova implementazione del sistema, vi erano elevate probabilità di riscontrarne altri, se non addirittura lo stesso. Questo fenomeno fa sì che certe implementazioni che abbiano conseguenze negative siano utilizzate come soluzioni a problemi di design ricorrenti. Alcune di queste soluzioni quindi si sono consolidate, e sono state raccolte dai pattern, che permettono il riutilizzo di tali soluzioni. I design pattern sono stati descritti per diverse categorie di problemi e soluzioni legati allo sviluppo del software, compresa l'architettura, la progettazione e lo sviluppo. Successivamente, hanno iniziato ad affermarsi gli *antipattern*. Introdotti da Brown ed altri [4], gli antipattern sono concettualmente simili ai pattern per il fatto che essi documentano soluzioni ricorrenti a problemi di design comuni. Prendono il nome di *antipattern* in quanto documentano le "bad practices" applicate durante lo sviluppo del software, così come le soluzioni ad esse. Pertanto, gli antipattern forniscono indicazioni su cosa evitare e come correggere, qualora venissero riscontrati, i problemi di design. Nell'ambito dei design pattern l'enfasi è sugli attributi qualitativi come riusabilità e manutenibilità, piuttosto che sulle performance. Quindi è diventato importante identificare quali sono quei pattern che hanno un buone caratteristiche che riguardano le prestazioni, e quelli che al contrario non hanno buone caratteristiche di performance. I *performance pattern* estendono i pattern includendo tali caratteristiche. I *performance antipattern* estendono, a loro volta, gli antipattern dando importanza alle caratteristiche prestazionali e sono stati introdotti da Smith e Williams [5, 6, 7]. Essi documentano i problemi di performance comuni e indicano quali sono le soluzioni per risolverli. Tali problemi sono spesso introdotti durante la fase di progettazione del software, ma non sono rilevati fino alla fase di testing o di rilascio. Il sistema in cui occorre un antipattern viene *rifattorizzato* (ristrutturato o riorganizzato) al fine di superarne gli aspetti negativi. Un *refactoring* è una trasformazione che preserva la correttezza del sistema e che migliora la qualità del software. I refactoring non alterano la semantica del sistema ma possono migliorare diversi attributi qualitativi del software, quali riusabilità, manutenibilità e prestazioni.

3. Approccio al refactoring di modelli guidato dalle performance nel contesto della piattaforma EPSILON

In questo capitolo vengono illustrati l'architettura del plug-in e l'approccio utilizzato per rappresentare la struttura di un programma scritto in EVL. Figura 3.1 illustra l'architettura del plug-in, sottolineando l'interazione fra i suoi vari componenti.

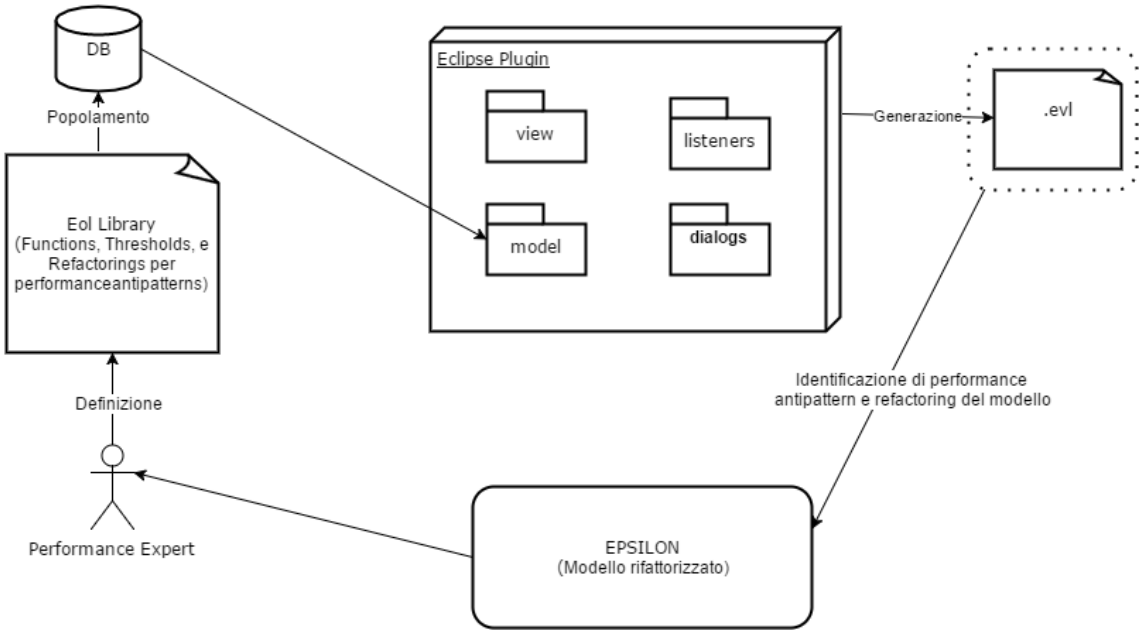


Figura 3.1 – Architettura del plug-in

Come si può vedere il cuore del plug-in è costituito dai seguenti package:

- Un package denominato *Model*, contenente le strutture dati atte a rappresentare la struttura di un file EVL Questo package contiene la classe Evl che è la classe principale, ovvero è quella che rappresenta il file Evl vero e proprio, contenente una List oggetti di tipo *Context*; questa classe rappresenta il costrutto evl *context* e contiene l'attributo *name* che indica il nome del context, e una collection di *Invariants* chiamata *containers*, che è una lista di oggetti di tipo *Container* che rappresenta gli *Invariants*. Le classi *Critque* e *Constraint* sono due sotto-classi di

Container e ne rappresentano i relativi costrutti. In ciascun *invariant* sono presenti, oltre al nome, contenuto nell'attributo *name*, i seguenti attributi:

- *check*: è un oggetto di tipo *Check* e rappresenta il costrutto *check* di EVL.
- *message*: è un istanza della classe *Message*, classe che rappresenta il costrutto *message* di EVL, ed ha come unico attributo una stringa contenente il campo del costrutto *message* di EVL.
- *fixList*: questo attributo è una *List<Fix>* e rappresenta i costrutti *fix* presenti negli *invariant* EVL.

La classe *Check* è stata implementata in questa tesi per rappresentare l'omonimo costrutto EVL, quindi deve poter contenere una lista di elementi che compongono la guardia del costrutto. Quindi è stata implementata la classe *Operation* che rappresenta ogni singola *Function* o *Threshold* che sono contenute nella guardia, e gli operatori logici fra loro. Le Function e le Threshold sono anch'esse rappresentate da due classi, la classe *F* e la classe *Threshold*. La classe che contiene le gli elementi nella classe Operation è la classe *Predicate* che è specializzata dalle classi *UnaryPredicate*, che contiene un'istanza della classe F che contiene una *function* che restituisce un valore booleano e quindi non necessita di essere confrontata con una *threshold* nella guardia, e dalla classe *BinaryPredicate*, che rappresenta al contrario quelle *function* che ritornano un real e devono essere confrontate con un'altra *function* o una *threshold* nella guardia, che contiene quindi un'istanza della classe *F*, un istanza della classe *Threshold* e un operatore di confronto fra essi (>,<,=,< >,>=,<=). La classe *Fix* contiene un Oggetto di tipo *Title* (contenente un solo attributo di tipo *String* che rappresenta il messaggio da applicare in caso di refactoring) e un'istanza della classe *Do* che contiene una lista di *String* ciascuna delle quali indica i nomi dei *do* contenuti nel costrutto *fix {}* di EVL.

- Un package *View* che insieme ai package *Listeners* e *Dialogs* contengono le classi che costituiscono la User Interface.

Le strutture dati vengono riempite da un Database, il cui contributo a questa tesi è quello di introdurre la persistenza dei dati, popolato manualmente a partire da una libreria EOL che è stata fornita per questo lavoro di tesi; questa libreria contiene delle *Function* e delle *Threshold* che forniscono, rispettivamente, funzioni per estrarre metriche dal modello in

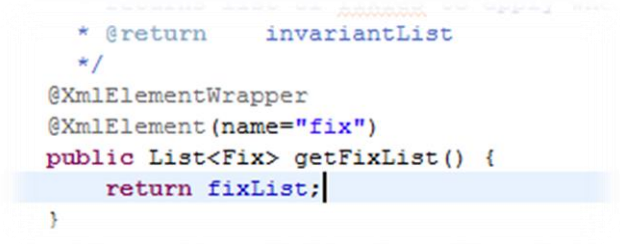
input e soglie con cui confrontare tali metriche, le quali riguardano il design e le performance dei modelli su cui il programma EVL verrà eseguito. La libreria EOL inoltre contiene le *funzioni per i refactoring* cioè la lista di comandi da effettuare, quando le condizioni espresse nella *check* del programma EVL si verificano.

La GUI del plug-in permette all'utente di creare in maniera guidata la struttura del programma. Il programma in linguaggio EVL che si sta creando viene mostrato all'utente tramite un modello ad albero. Mediante questa struttura l'utente può creare il file EVL inserendo a proprio piacimento i costrutti del linguaggio selezionando l'elemento dell'albero che si vuole popolare; una volta selezionato un dato elemento dell'albero, in base alla selezione, all'utente viene data la possibilità di cliccare su dei bottoni, posti di fianco all'albero, che permetteranno l'inserimento degli elementi che sono contenuti nell'elemento selezionato. Il plug-in inoltre fornisce la possibilità di salvare il progetto corrente, per poter riprendere la creazione del file EVL in seguito senza dover ricominciare da capo; Il salvataggio avviene mediante la serializzazione della classe *Evl.java*, che contiene tutti i costrutti del linguaggio, in Xml. Per effettuare la serializzazione tutte le classi sono state opportunamente annotate, con delle annotazioni XML, come le annotazioni presenti in figura 3.2 e in figura 3.3, in cui si possono notare le annotazioni *@XmlRootElement* e *@XmlSeeAlso({Critique.**class**,Constraint.**class**})* e le annotazioni *@XmlElementWrapper* e *@XmlElement(name="fix").*

Figura 3.2 - Annotazioni Xml

```
/**
 * It represents the Invariant contained in the context
 *
 *@author Davide Di Gironimo
 */
@XmlRootElement
@XmlSeeAlso({Critique.class,Constraint.class})
public class Container {
      * Type of theinvariant : Critique or constraint;
    protected String type;
      * Name of the Invariant
```

Figura 3.3 - Annotazioni Xml



Queste annotazioni sono presenti in tutte le classi del modello, per permetterne la serializzazione.
Il file .evl generato sarà conforme alla struttura e alla sintassi del linguaggio EVL e potrà essere eseguito, tramite la piattaforma *Epsilon*, sui modelli desiderati.

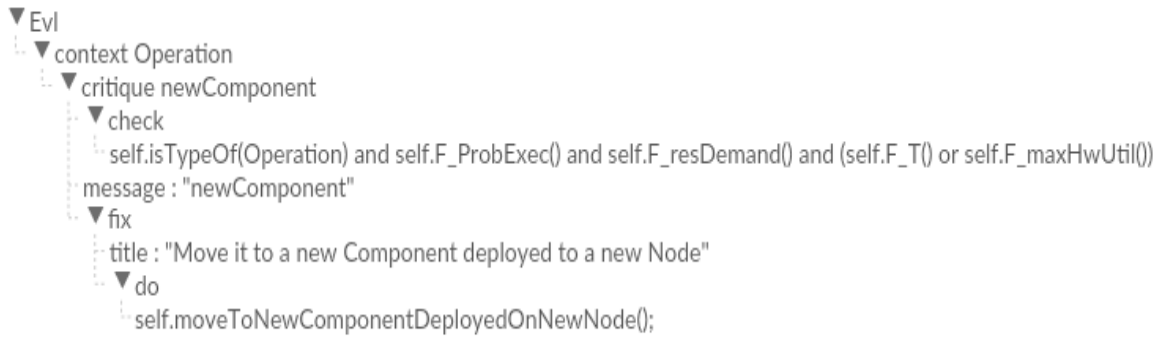
3.1 Modello ad albero per la strutturazione di file EPSILON

Durante la progettazione dell'interfaccia grafica si è presentata la necessità di rendere visibile all'utente la struttura del file EVL progressivamente creato. Per permettere quindi all'utente di poter gestire i costrutti del linguaggio senza effettivamente vedere il codice è stata scelta una struttura ad albero per rappresentare la gerarchia dei costrutti del linguaggio EVL. Questo tipo di rappresentazione permette di mantenersi conformi alla sintassi di EVL e di mostrare all'utente solo le informazioni indispensabili per la creazione del programma. Grazie a questa scelta sono state agevolate le operazioni di "completamento" dell'albero, in maniera guidata ed intuitiva, selezionando il nodo desiderato. Nella figura 3.1.2 è rappresentato, sotto forma di albero, il listato in figura 3.1.1; Si può notare come l'albero sia conforme alla struttura sintattica dello snippet di codice EVL mostrato, senza perdita di informazione.

Figura 3.1.1 – Snippet di codice in EVL

```
context Operation {
  critique newComponent{
    check{
      self.isTypeOf(Operation) and self.F_probExec() and self.F_resDemand()
      and (self.F_T() or self.F:maxHwUtil())
    }
    messagge : " newComponent"
    fix{
      title : "Move it to a new Component deployed to a new Node"
      do{
        self.moveToNewNodeComponentDeployedOnNewNode();
      }
    }
  }
}
```

Figura 3.1.2 – Rappresentazione ad albero dello snippet in figura 3.1.1



3.2 Libreria di funzioni EPSILON per l'estrazione di metriche e refactoring

Come già detto nell'introduzione al capitolo, questa libreria contiene le *Function*, le *Threshold* ed i *Refactoring* (anche riferiti come *Do*). Le Function e le Threshold inserite in questa libreria, devono necessariamente riferirsi a delle metriche specifiche dei modelli su cui andranno verificate; tali metriche possono riferirsi sia ad aspetti legati al design dei modelli (per esempio possono indicare il numero di connessioni di una componente), sia

alle performance dei modelli (ad esempio il tempo di risposta di un servizio). Detto ciò, è di fondamentale importanza che il creatore di questa libreria abbia le conoscenze necessarie per poter definire parametri che si riferiscano alle performance e al design dei modelli. Questa figura è denominata *Performance Expert*, e detiene le conoscenze sui performance anti-pattern e sui refactoring da applicare quando la corrispondenza del modello UML con un anti-pattern viene verificata. Nella figura seguente (Figura 3.3.1) sono esposti alcuni esempi di thresholds, distinte in thresholds di design e thresholds di performance; tra le threshold di performance si possono notare threshold che indicano per esempio il massimo e il minimo utilizzo della risorsa, o soglie di utilizzo dell'hardware o della rete. Nella figura 3.3.2, invece, possiamo vedere un esempio di Function di design; la Function si chiama *F_numClientsConnects* e predica sul context *Component*, ritornando un valore boolean. Possiamo notare come questa Function prenda in ingresso una threshold, che è di design. Nella figura 3.3.3 è presente un esempio di che predica su parametri di performance. Questa Function, è chiamata *F_resDemand* e predica sul context *Operation* restituendo anch'essa un valore boolean. *F_resDemand* prende in ingresso la threshold di performance *th_maxResDemand*.

Figura 3.3.1 – Esempio di Thresholds

```
/******** DESIGN **************
var _th_maxMsgs() : Integer = 3;
var _th_maxRemMsgs() : Integer = 2;
var _th_maxRemInst() : Integer = 1;
var _th_maxConnects() : Integer = 4;
var _th_maxExF() : Integer = 2;

/******** PERFORMANCE **************

//Resource Demand
var _th_maxResDemand() : List( Real ) = new List( Real );
_th_maxResDemand().add(15); //computation
_th_maxResDemand().add(7); //storage
_th_maxResDemand().add(5); //bandwidth

var _th_minResDemand() : List( Real ) = new List( Real );
_th_minResDemand().add(5); //computation
_th_minResDemand().add(10); //storage
_th_minResDemand().add(3); //bandwidth

//Hw Utilization
var _th_maxHwUtil() : Real = 0.85;

//Network utilization
var _th_maxNetUtil() : Real = 0.9;
var _th_minNetUtil() : Real = 0.5;

var _th_SrtReq() : Real = 15; //response time

var _th_SthReq() = 0.9; //throughput
var th_maxQL : Integer = 130; //queue lenght
```

Figura 3.3.2 – Esempio F di Design

```
operation Component F_numClientConnects(th_maxConnects : Integer) : Boolean{

  if(not self.hasStereotype("PaRunTInstance"))
    return false;

    if(self.getRequireds().size() >= th_maxConnects){
    return true;
  }
  return false;
}
```

Figura 3.3.3 – Esempio F di Performance

```
operation Operation F_resDemand(th_maxResDemand : List(Real)) : Boolean{

  var turnback : Boolean = true;

  if(not self.hasStereotype("GaStep"))
    return false;

  var gaStep = self.getStereotype('GaStep');
  var servCount = self.getValue(gaStep, 'servCount');

  var i : Integer = 0;
  if(servCount.isDefined() and servCount.size() == 0){
    turnback = false;
  }else{
    for(demand in servCount){

      if(demand.asReal() < th_maxResDemand.at(i) ){ //al primo demand non > della soglia relativa ritorna false
        turnback = false;
        break;
      }
      i = i+1;
    }
  }
  return turnback;
}
```

3.3 Database di funzioni EPSILON per l'estrazione di metriche e refactoring

Il plug-in sviluppato si basa su un database, implementato appositamente per questo progetto di tesi, che contiene le Function, le Threshold e i Refactoring, che vengono inseriti nel database a partire dalla libreria EOL. Data la libreria EOL generata dal Performance Expert, il database viene popolato manualmente da una figura che non deve necessariamente essere la stessa che crea la libreria, in quanto le conoscenze necessarie per inserirla nel database non sono le stesse che si richiedono per la creazione di tale libreria. Quindi, dato che il database viene popolato a partire dalla libreria EOL, deve avere le tabelle per poter contenere le Function, le Threshold, i Refactoring e i Context, ma anche le relazioni necessarie fra tali tabelle perché le Function e i refactoring sono legati al contesto su cui possono essere eseguite, e le Threshold sono legate alle Function con cui possono essere confrontate. Quindi è necessario che il database sia predisposto a contenere tali relazioni, che essendo tutte n a m, prevedono una tabella per ciascuna relazione. Ogni volta che il plug-in viene eseguito, la lista di context presenti nella relativa tabella del database viene estratta, e in base al context selezionato ed inserito nell'albero, da parte dell'utente, vengono estratte dal database la lista di Function e la lista dei Refactoring in relazione con esso. Analogamente selezionando una Function, il plug-in estrae dal database la lista di Threshold in relazione con la Function.

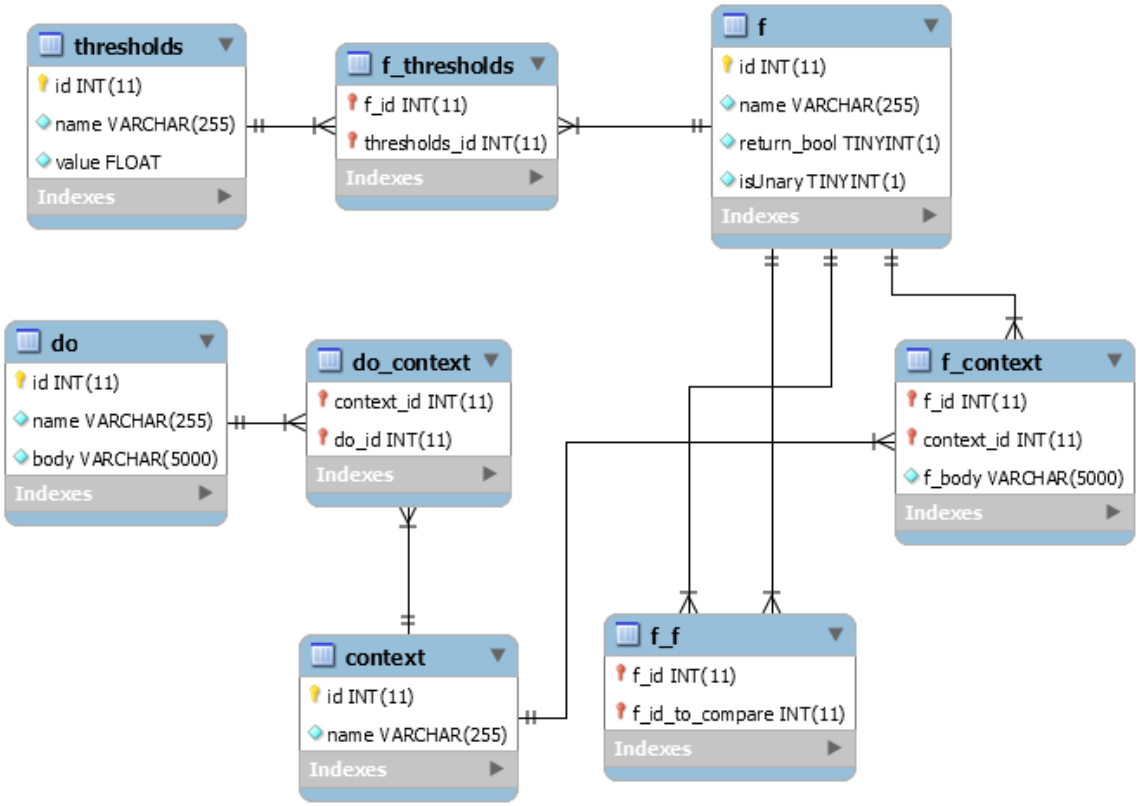


Figura 3.2.1 – Diagramma ER

Nella figura 3.2.1 è illustrato il diagramma ER del database realizzato; nel seguito vengono descritte brevemente le tabelle e le relazioni fra esse:

- Context: In questa tabella vengono inseriti tuti i context su cui verranno eseguite le operazioni dei linguaggio EVL; contiene i campi *id*, primary key della tabella, e *name* che contengono rispettivamente un id univoco per il record e il nome del context.
- F: Qui sono presenti tutte le function, progettate dal Performance Expert; la tabella contiene i campi *id* (primary key), che contiene l'id univoco per ogni record, *name* che indica il nome della Function e *return_bool* che è un campo boolean, e indica se il valore di ritorno della Function è un boolean (viene inserito il valore true nel campo) o se ritorna un valore di tipo Real (false).
- F_Context: Questa tabella rappresenta una relazione *n* a *m* fra le tabelle Context e F, dando la possibilità di poter associare le Function ai context su cui possono

essere utilizzate. Essa contiene i campi relativi agli id delle due tabelle che mette in relazione, ovvero *f_id* e *context_id*, oltre al campo *f_body* contenente il corpo della function relativa al context. I due id sono ovviamente due chiavi esterne, ognuna relativa alla propria tabella. La primary key di questa tabella è la coppia di campi f_id e context_id.

- Thresholds: Qui sono inserite le soglie con cui confrontare le function. La tabella contiene i campi *id*, *name* e il campo *value* contenente il valore di tipo Real della Threshold. La chiave primaria è il campo *id*.
- F_Thresholds: In questa tabella è rappresentata la relazione fra la tabella F e le Thresholds; relazione *n* a *m*; la tabella contiene i campi *f_id* (foreign key della tabella F) e *threshold_id* (foreign key della tabella *Thresholds*), che in coppia formano la primary key della tabella.
- Do: Contiene i refactoring da inserire nel corpo del costrutto *Fix* del linguaggio EVL. Nella tabella *Do* ci sono i campi *id* che è univoco quindi Primary Key, il campo *name* contenente il nome della funzione, e *body* che contiene il corpo della funzione.
- Do_Context: Questa tabella rappresenta una relazione *n* a *m* fra la tabella Do e la tabella Context, e indica quali refactoring possono essere applicati su ogni Context. La tabella *Do_Context* contiene i due id delle tabelle che mete in relazione, cioè *do_id* e *context_id* che sono due chiavi esterne e che insieme formano la chiave primaria della tabella.

4. Caso di studio

In questo capitolo verrà esposto un caso di studio su alcuni modelli in esempio. Per prima cosa verrà generato un programma in EVL con il plug-in sviluppato in modo da riscontrare la corrispondenza del modello dato con dei design antipattern (chiamati rispettivamente *Concurrent Processing Systems*, *Extensive Processing* e *Pipe and Filter*). Una volta ottenuto, il file EVL, tramite la piattaforma Epsilon verrà eseguito sul modello di esempio e se esso corrisponderà a uno o più design antipatterns Epsilon permetterà

all’utente di scegliere quali refactoring applicare in base agli antipattern che sono stati riscontrati sul modello.

4.1 Modello UML di esempio

Il modello UML su cui verranno eseguite le operazioni di validazione e di refactoring è illustrato nelle seguenti figure che mostrano delle viste del modello chiamate Static View che rappresenta i contenuti statici di un software come le classi, oggetti, interfacce e le relazioni fra esse (Figura 4.1.1), Deployment View che descrive il sistema in termini di risorse hardware, dette “Nodi”, e le relazioni fra esse (Figura 4.1.2) e la Dynamic View (Figura 4.1.3) usata per rappresentare i comportamenti dei componenti statici del software. Nella Figura 4.1.2 è illustrata la static view tramite un Component Diagram, la Figura 4.1.4 illustra la Deployment View tramite un Deployment Diagram mentre la Figura 4.1.6 rappresenta la Dynamic View con un Sequence Diagram.

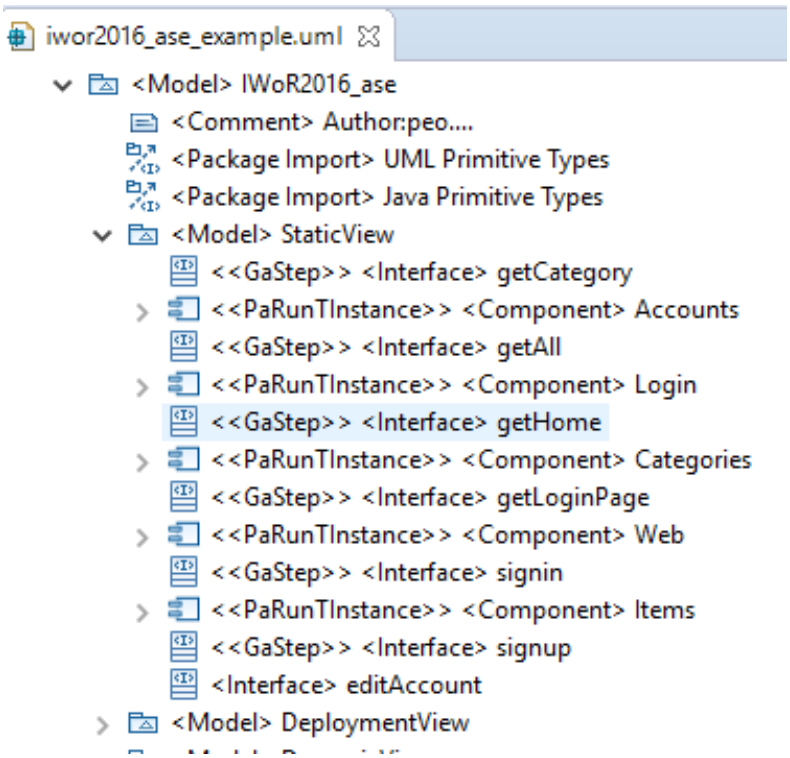


Figura 4.1.1 - Static View

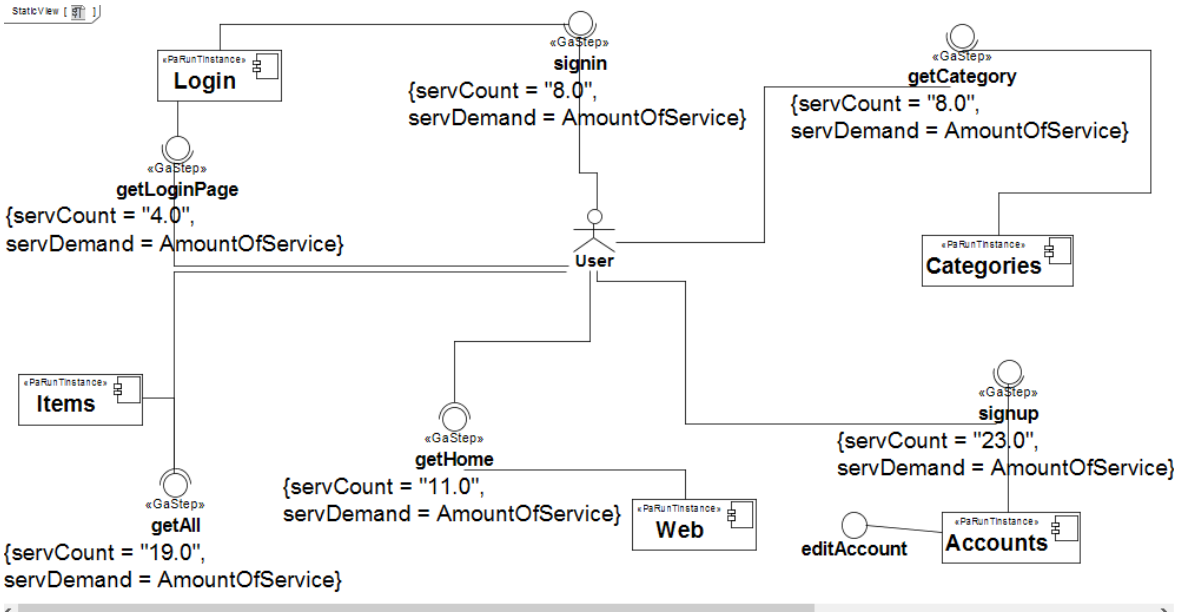


Figura 4.1.2 - Component Diagram

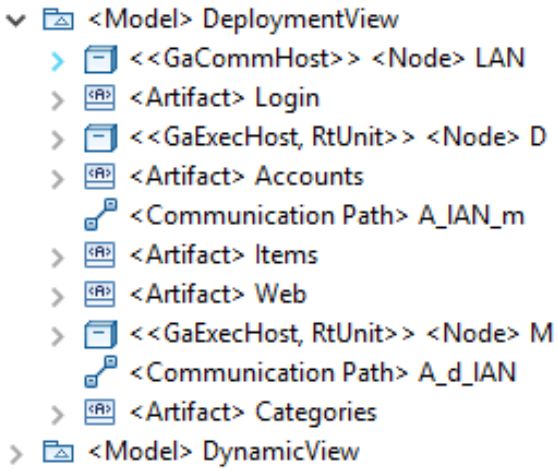


Figura 4.1.3 – Deployment View

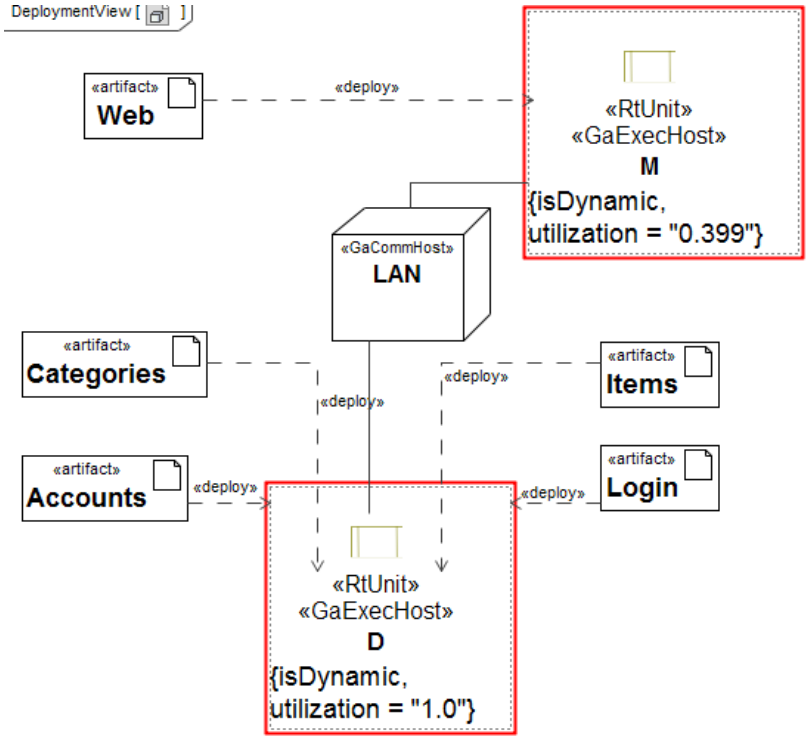


Figura 4.1.4 - Deployment Diagram

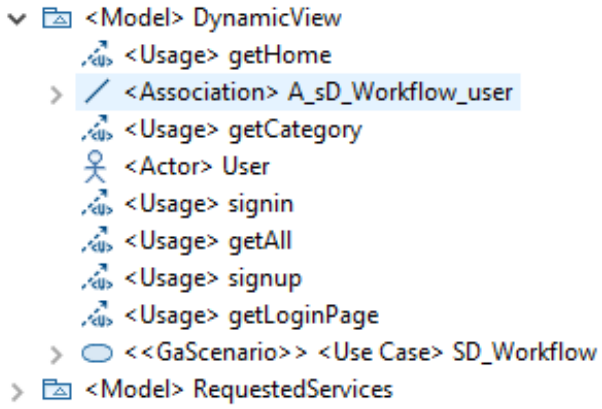


Figura 4.1.5 - Dynamic View

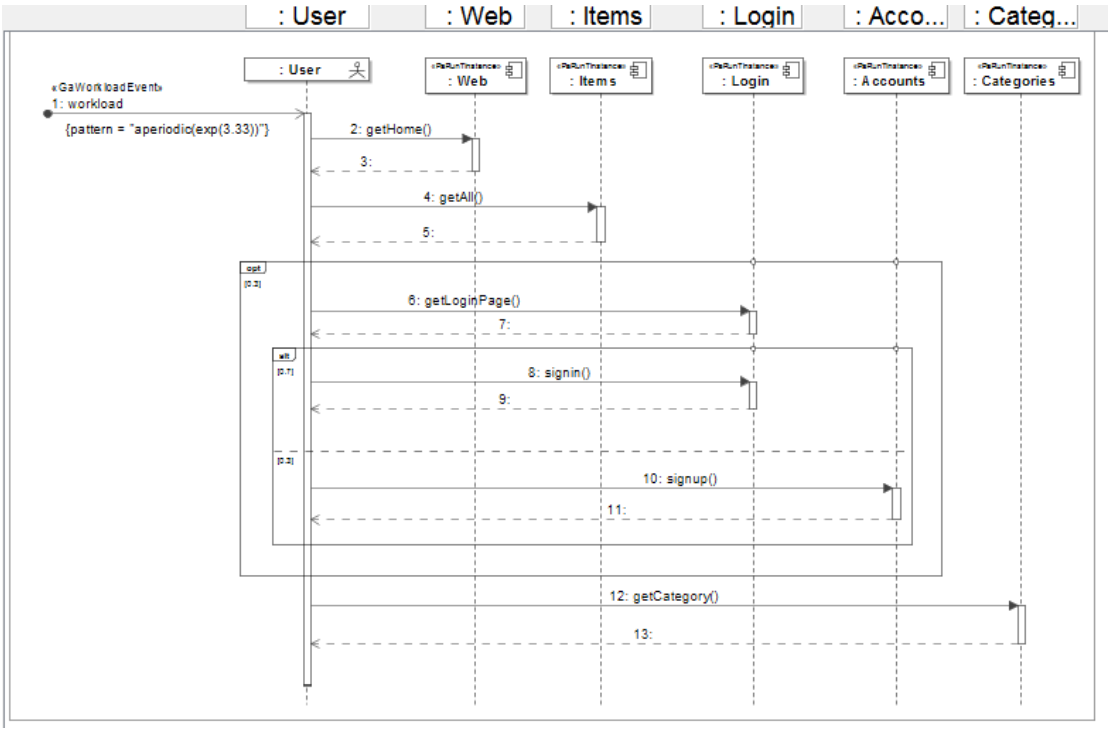


Figura 4.1.6 - Sequence Diagram

Un caso d'uso del modello viene mostrato in Figura 4.1.7.

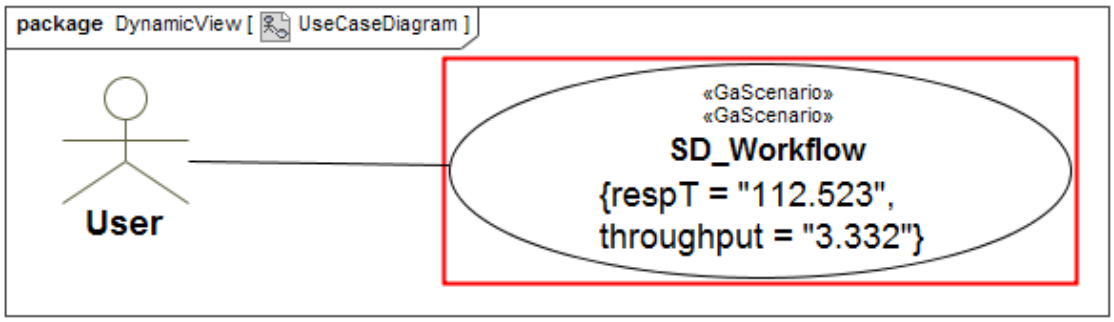


Figura 4.1.7 - Use Case Diagram

4.2 Uso del tool e generazione automatica del codice EPSILON

In questa sezione verrà creato il file EVL che successivamente verrà utilizzato, per verificare la presenza degli antipattern sopracitati e per la loro eventuale correzione, eseguendo il programma sviluppato nella piattaforma Epsilon; il file EVL verrà creato

con il plug-in sviluppato e per questo tutte le fasi della sua creazione saranno descritte in maniera tale da illustrare il funzionamento del plug-in. Il primo programma EVL sarà generato per verificare la corrispondenza del modello UML ai design antipattern *Pipe and Filter, Extensive Processing e Concurrent Processing System*. Una volta installato il plug-in ed eseguito Eclipse, nella schermata iniziale di Eclipse stesso si potrà notare la presenza di un icona nella sua toolbar principale (Figura 4.2.1).

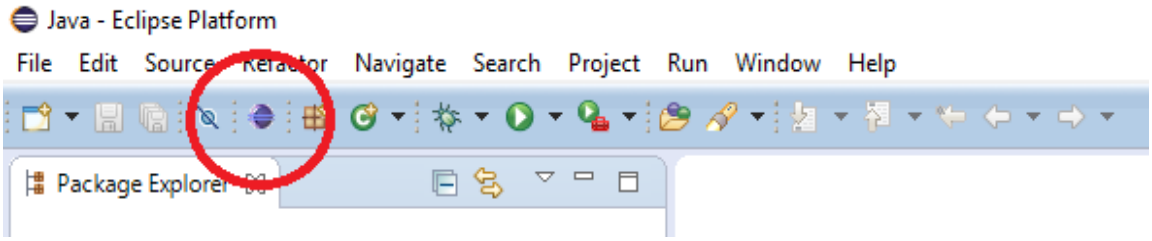


Figura 4.2.1 – Schermata di Eclipse con il Plug-in integrato;

Cliccando sull'icona evidenziata si aprirà l'interfaccia del plug-in, che ci permetterà di creare un programma scritto in linguaggio EVL. La Figura 4.2.2 rappresenta la schermata che si presenta all'apertura del plug-in.

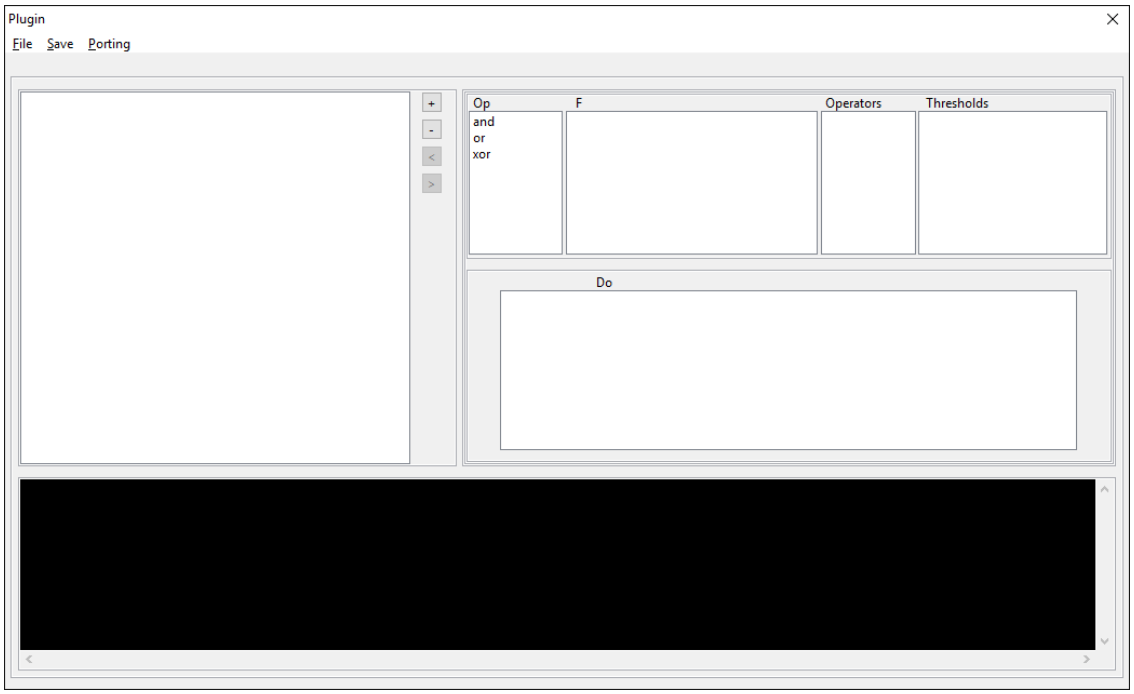


Figura 4.2.2 – Schermata iniziale del Plug-in

Trovandoci di fronte alla schermata come in Figura 4.2.2 possiamo notare la presenza del menu principale e dei pulsanti posizionati di fianco alla struttura ad albero, che inizialmente appunto è vuota; cliccando sulla voce ‘File’ del menù abbiamo la possibilità di:

- Creare un nuovo progetto vuoto.
- Salvare il progetto corrente in Xml.
- Aprire un progetto precedentemente salvato.
- Uscire dal plug-in.

Cliccando invece sulla voce ‘Save’ del menù possiamo salvare la struttura finora creata in EVL. Nella parte destra del plug-in sono presenti dei form che ci permettono di popolare i costrutti *check* e *do* del linguaggio EVL.

A questo punto possiamo iniziare la creazione del programma EVL; cliccando sul bottone ‘+’ appare un menu a discesa che permette l’inserimento dei costrutti del linguaggio a partire dall’elemento dell’albero correntemente selezionato. Partendo quindi dall’albero vuoto, come in Figura 4.2.2, cliccando sul bottone ‘+’ possiamo aggiungere

soltanto dei context al nostro albero. Una volta effettuata questa operazione appare una schermata come in Figura 4.2.3, in cui possiamo scegliere su quale context operare, selezionando il context scelto da una combo box.

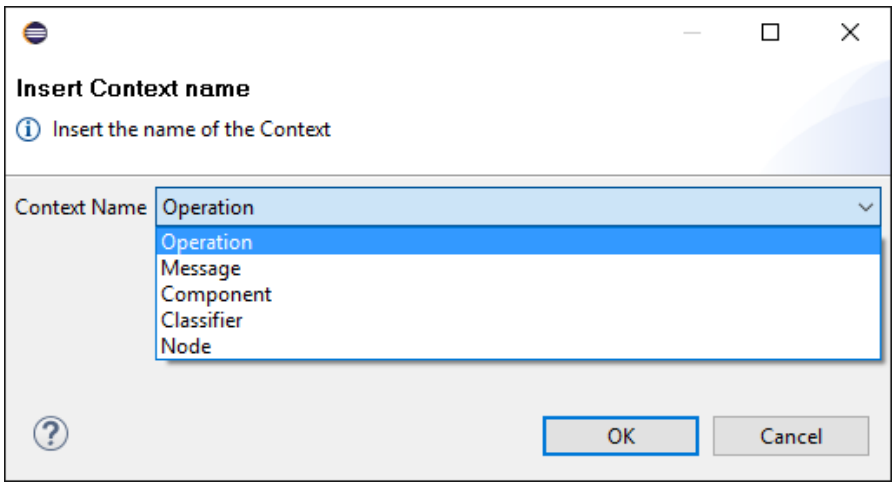


Figura 4.2.3 – Inserimento Context

Ora nell'albero è presente il context inserito e possiamo iniziare ad inserire i costrutti al suo interno. Selezionando il context e cliccando sul bottone '+', nel menu a discesa troviamo una voce per l'inserimento degli invariant (Figura 4.2.4), e selezionandola si aprirà una schermata come in Figura 4.2.5, dove possiamo scegliere il tipo dell'invariant da inserire e il suo nome.

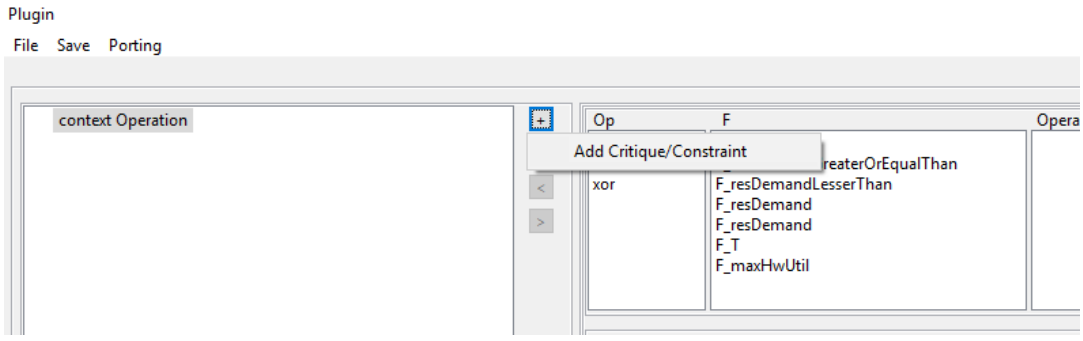


Figura 4.2.4 – Inserimento invariant

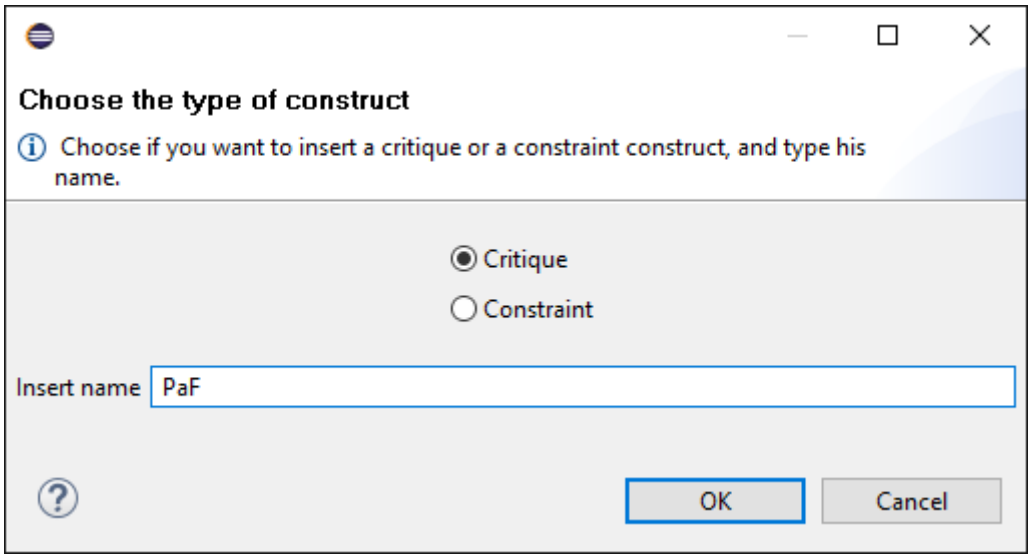


Figura 4.2.5 - Inserimento Invariant

Allo stesso modo in cui è stato inserito il primo invariant nel context, possiamo inserire tutti i costrutti annidati nell'invariant aggiunto, come i costrutti *check, message* e *fix*. Inserito il costrutto *check*, bisogna popolarlo con l'ausilio della parte 'destra' dell'interfaccia in cui, come si può notare dalla Figura 4.2.5, sono presenti delle liste di *Function, Threshold* e di operatori. Selezionando l'elemento check dall'albero viene popolata la lista delle *Function* che possono essere applicate al *context* in cui il *check* è contenuto. Selezionando quindi la *Function*, se essa ritorna un valore boolean possiamo inserirla nel corpo del check scegliendo opportunamente gli operatori dalle due liste, mentre se la *Function* restituisce un valore real va confrontata con una *Threshold*, quindi la lista relativa alle threshold viene popolata, e l'utente deve scegliere opportunamente in base ai propri scopi la threshold e gli operatori insieme alla Function. Composta la Function, possiamo inserirla nel corpo del check, cliccando sul bottone '<', ovvero una freccia verso l'albero, come in Figura 4.2.6. Il corpo del check deve essere compilato inserendo le *Function* che permettono all'utente di effettuare dei controlli su determinate condizioni dei modelli; dato che lo scopo della creazione del file EVL in questo caso d'uso è quello di verificare se il modello d'esempio è conforme ai design anti-patterns Concurrent Processing Systems, Extensive Processing e Pipe and Filter, sarà necessario che il file EVL generato contenga più context su cui operare, più Function per ogni check

e più Do, quindi più Function possono essere combinate, con l'ausilio degli operatori logici, per i raggiungimento di questi scopi.

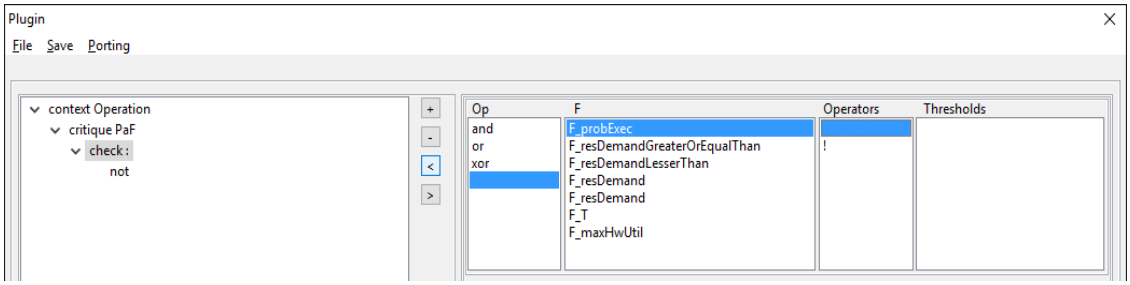


Figura 4.2.6 - Inserimento Function

Completato il corpo del check, proseguiamo nella creazione del programma inserendo tutti i costrutti del linguaggio, come quanto fatto per i costrutti context e l'invariant critique. Una volta inseriti tutti i costrutti contenuti nell'invariant '*critique PaF*', mancano solo il popolamento del costrutto *Do*, che come già detto contiene le funzioni per il refactoring, in una maniera analoga a quanto già fatto per il popolamento del corpo del check: selezionando dall'albero l'elemento *do*, la lista presente nella parte destra della GUI (Figura 4.2.7) sarà popolata con tutte le funzioni applicabili sul *context* che contiene l'elemento dell'albero selezionato.

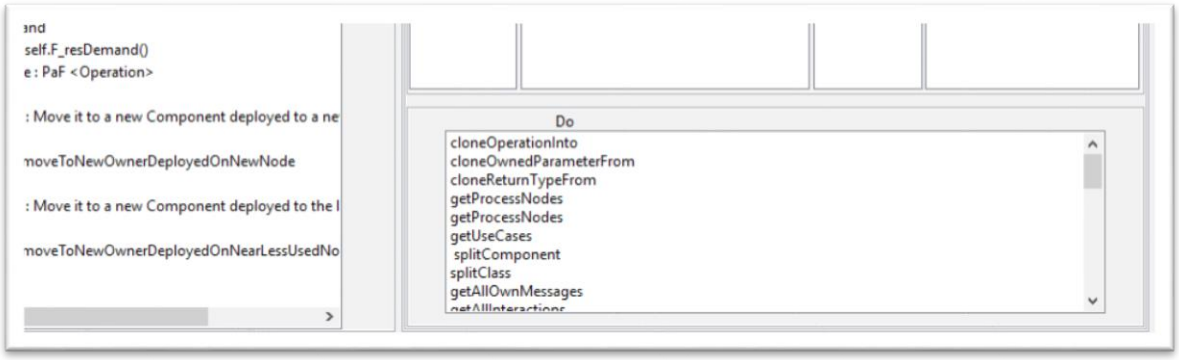


Figura 4.2.7 - Lista di Do popolata in base al context

Per inserire una funzione dalla lista Do, nell'elemento *do* dell'albero, si deve selezionare l'elemento dell'albero che si desidera popolare, selezionare dalla lista Do la funzione da

inserire per poi cliccare sul bottone '<'. Effettuate queste operazioni l'albero sarà popolato come in Figura 4.2.8, dai cui si nota come la funzione di refactoring sia stata inserita nel costrutto do. Il costrutto *title* indica l'etichetta da mostrate all'utente per identificare il do contenuto nello stesso costrutto Fix. Successivamente vedremo che il contenuto del costrutto *title* non è altro che l'elemento di una lista a discesa che l'utente dovrà selezionare se vorrà effettuare quella determinata operazione sul modello.

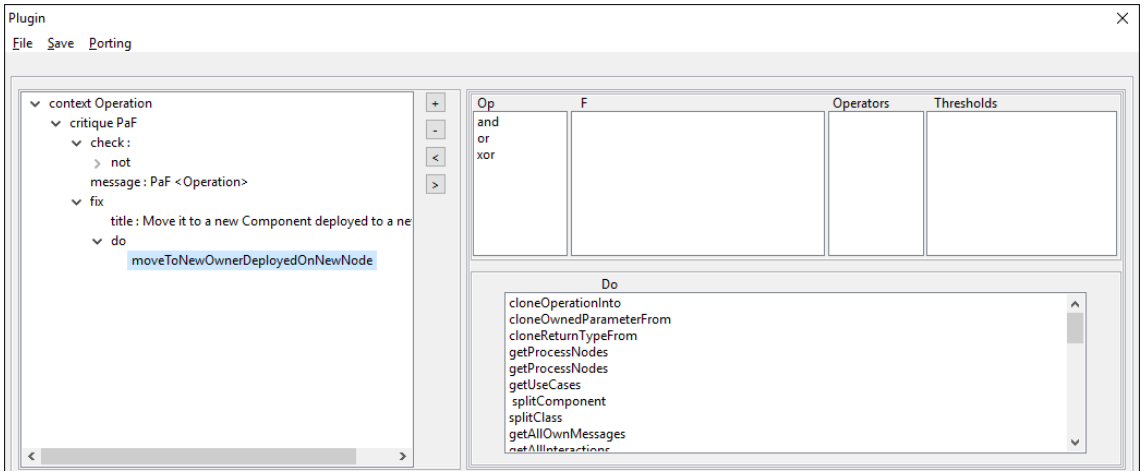


Figura 4.2.7 - Inserimento Do

Completata la creazione relativa al context per la rilevazione del *Pipe and Filter Anti-pattern*, vanno creati i context per rilevare rispettivamente il *Concurrent Processing System Anti-pattern* (Figura 4.2.8) e l'*Extensive Processing Anti-pattern* (Figura 4.2.9).

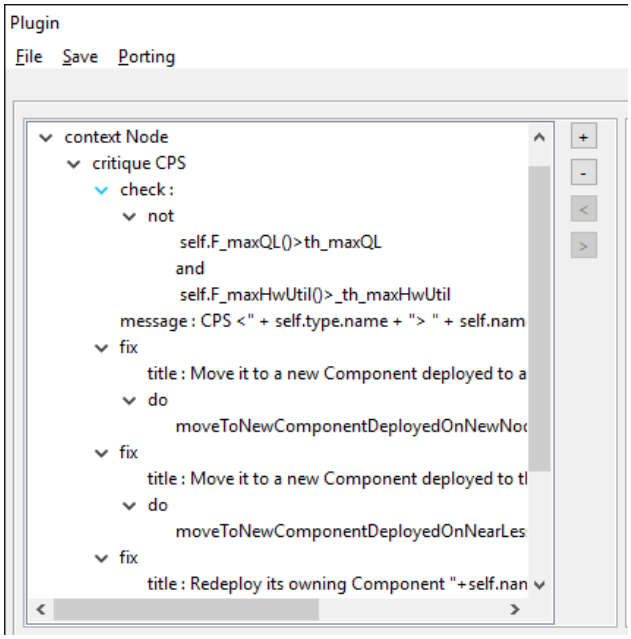


Figura 4.2.8 – Creazione context per il rilevamento dell'anti pattern Concurrent Processing System

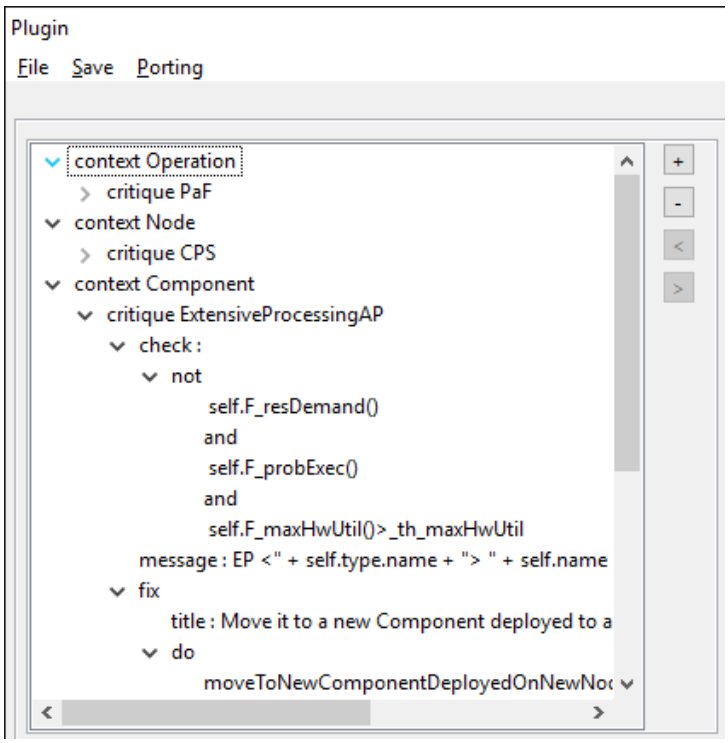


Figura 4.2.9 - Creazione context per il rilevamento dell'anti-pattern Extensive Processing

Ora che l'albero EVL è stato completato per poter rilevare e correggere i tre anti-patterns, si può procedere con il salvataggio del file evl. Per salvare il file '.evl' è necessario cliccare la voce 'Save' del menu principale per poi cliccare su 'Save in .evl'. Di seguito viene mostrato l'estratto di codice generato tramite l'ausilio del plug-in.

```
context Node {
    critique CPS {
        check:
            not ( self.F_maxQL(th_maxQL()) and self.F_maxHwUtil(th_maxHwUtil()) )
        message : "CPS <" + self.type.name + "> " + self.name
        fix {
            title : "Move it to a new Component deployed to a new Node"

            do {
                self.moveToNewComponentDeployedOnNewNode();
            }
        }
        fix{
            title : "Move it to a new Component deployed to the less used neighbour Node"
            do{
                self.moveToNewComponentDeployedOnNearLessUsedNode();
            }
        }
        fix{
            title : "Redeploy its owning Component "+self.name+" to the less used neighbour Node"
            do{
                self.redeployOnNearLessUsedNode();
            }
        }
        fix{
            title : "Change its owning Component from "+self.name+" to the one with the lowest demand"
            do{
                self.moveToLessCriticalComponent();
            }
        }
    }
}
```

```
context Operation {
    critique PaF {
        check:
            not (self.F_probExec() and  self.F_resDemand(th_maxResDemand()) and
                    (not self.F_T(th_SthReq()) or not self.F_maxHwUtil(th_maxHwUtil()) ))
        message : "PaF <Operation> " + self.name
        fix {
            title : "Move it to a new Component deployed to a new Node"
            do {
                self.moveToNewOwnerDeployedOnNewNode();
            }
        }
        fix{
            title : "Move it to a new Component deployed to the less used neighbour Node"
            do{
                self.moveToNewOwnerDeployedOnNearLessUsedNode();
            }
        }
        fix{
            title : "Redeploy its owning Component "+self.class.name+" to the less used neighbour Node"
            do{
                self.redeployOnNearLessUsedNode();
            }
        }
        fix{
            title : "Change its owning Component from "+self.class.name+" to the one with the lowest demand"
            do{
                self.moveToLessCriticalOwner();
            }
        }
        fix{
            title: "Decomposition"
            do{
                self.decomposition();
            }
        }
    }
}
```

```
context Component {
   critique ExtensiveProcessingAP{
       check:
           not (self.F_resDemand(th_maxResDemand(), th_minResDemand()) and
                   self.F_probExec() and
                       ( self.F_maxHwUtil(th_maxHwUtil())) or self.F_RT(th_SrtReq()) )
       message : "EP <" + self.type.name + "> " + self.name
       fix {
           title : "Move it to a new Component deployed to a new Node"

           do {
               self.moveToNewComponentDeployedOnNewNode();
           }
       }
       fix{
           title : "Move it to a new Component deployed to the less used neighbour Node"
           do{
               self.moveToNewComponentDeployedOnNearLessUsedNode();
           }
       }
       fix{
           title : "Redeploy its owning Component "+self.name+" to the less used neighbour Node"
           do{
               self.redeployOnNearLessUsedNode();
           }
       }
       fix{
           title : "Change its owning Component from "+self.name+" to the one with the lowest demand"
           do{
               self.moveToLessCriticalComponent();
           }
       }
   }
}
```

Nella prossima sezione verrà mostrata l'esecuzione del file EVL mostrato tramite la piattaforma Epsilon sul modello d'esempio.

4.3 Identificazione di performance anti-pattern e refactoring del modello di esempio

Ora non resta che eseguire il file EVL sul modello d'esempio illustrato. Per fare ciò è necessario eseguire Epsilon e creare un progetto in cui andranno inseriti il modello UML, il file EVL generato e le librerie EOL in cui sono contenute le Function, le Threshold e le funzioni per i Refactoring. Per poter eseguire il file EVL sul modello bisogna configurare una "Run Configuration" in cui andranno selezionati il file EVL da eseguire e il modello che si vuole validare. Nelle Figure 4.3.1 e 4.3.2 viene illustrata la configurazione utilizzata per eseguire il file EVL creato, chiamato *evl.evl*, sul modello d'esempio *iwor2016_ase_example.uml*.

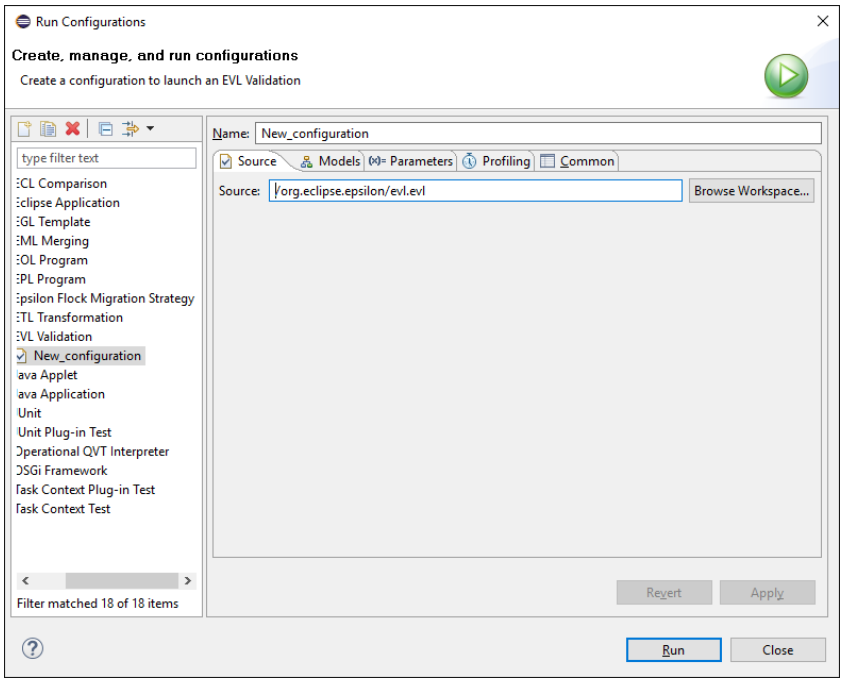


Figura 4.3.1 - Run Configuration, selezione file EVL

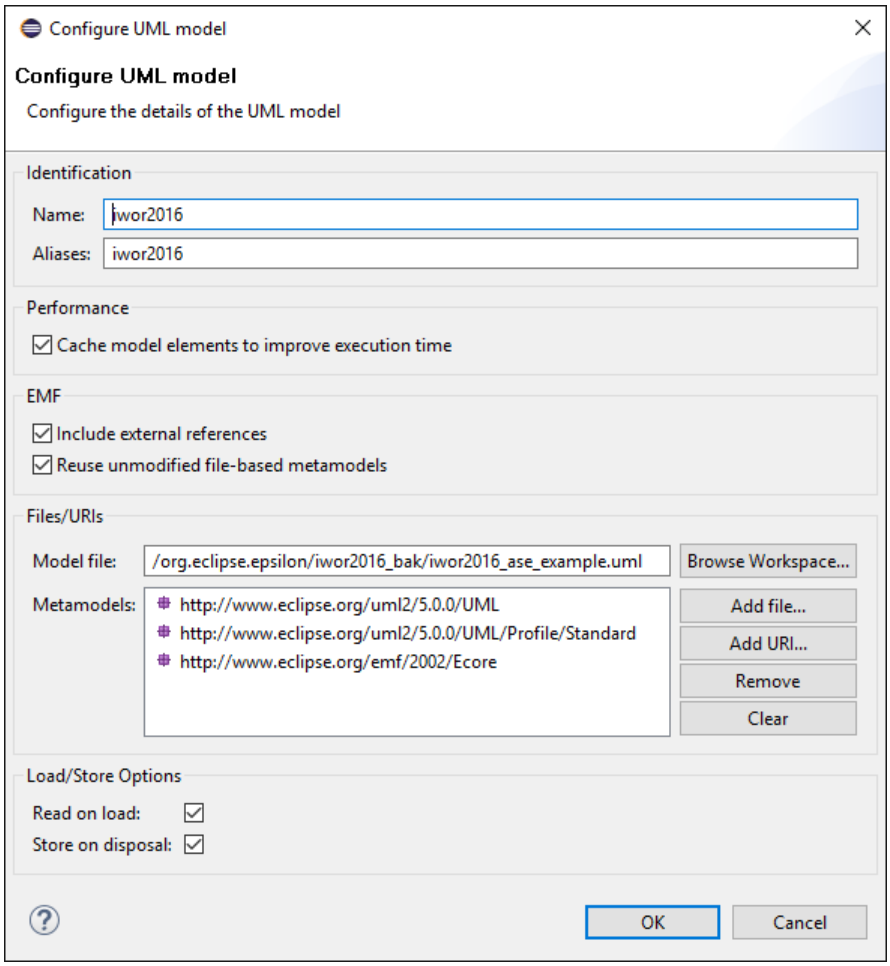


Figura 4.3.2 - Run Configuration, configurazione modello UML

Cliccando su *Run* verrà eseguita la configurazione creata e Epsilon restituirà, in una apposita sezione del suo Workbench, quali sono gli antipattern riscontrati nel modello. In Figura 4.3.3 viene mostrata la schermata ottenuta, da cui si può notare come Epsilon abbia notificato la corrispondenza del modello d'esempio con gli antipattern *Concurrent Processing System (CPS)*, *Extensive Processing (EP)* e *Pipe And Filter (PaF)*.

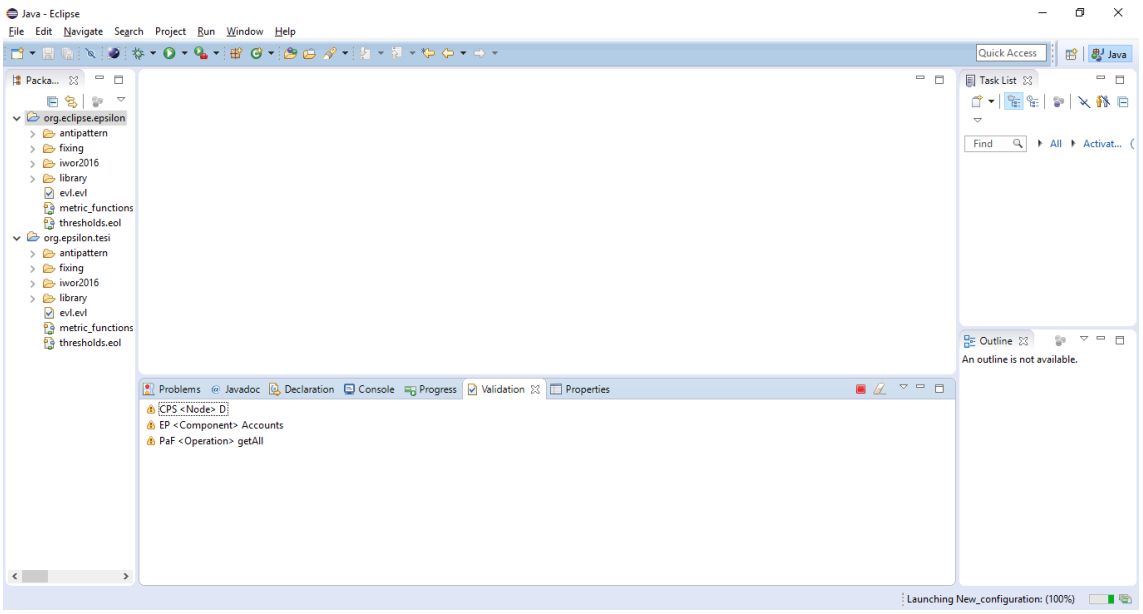


Figura 4.3.3 - Antipattern detection

Per rifattorizzare il modello e renderlo non più conforme agli antipattern basta cliccare con il tasto destro sull'antipattern desiderato, e selezionare uno dei refactoring che appare nel menu a tendina. Le funzioni per i refactoring presenti nel menu, non sono altro che i Do aggiunti nel costrutto fix del file EVL creato, e gli elementi del menu sono dati dal costrutto title del fix (Figura 4.3.4).

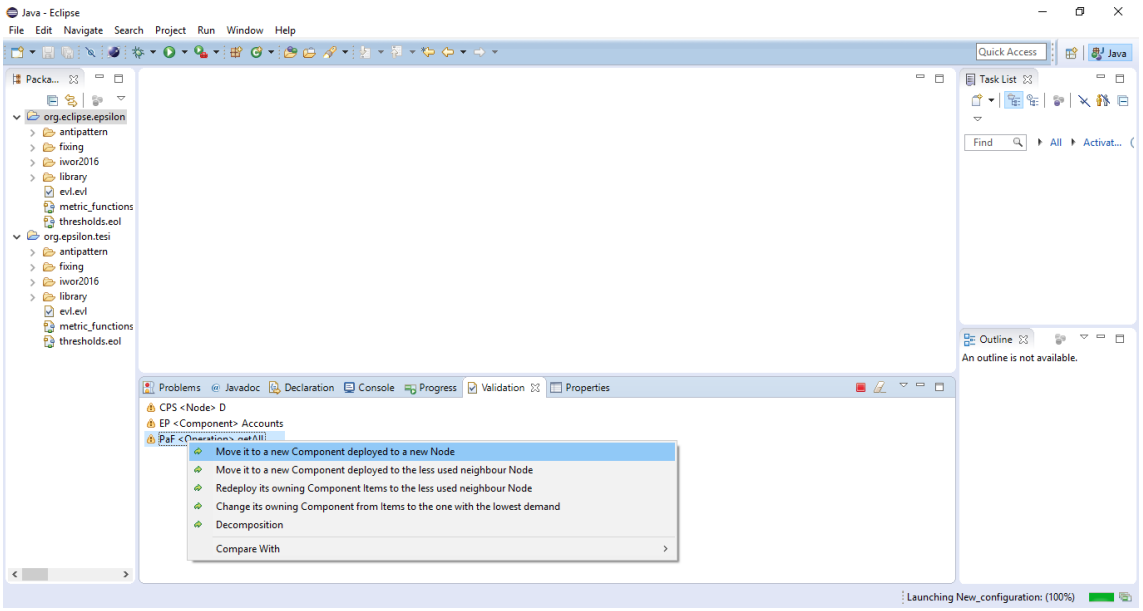


Figura 4.3.4 - Scelta del refactoring

Selezionando una delle quattro operazioni verranno eseguite le operazioni descritte dal messaggio stesso effettuando il refactoring necessario per risolvere il problema riscontrato sul modello. Per questo caso d'uso verranno effettuati due refactoring, *Decomposition* per l'antipattern Pipe and Filter, mentre *Move it to a new component deployed on a new node* è stato il refactoring scelto per l'antipattern Extensive Processing. Nelle Figure 4.3.5 e 4. 3.6 è presente la static view del modello prima e dopo i refactoring, mentre nelle Figure 4.3.7 e 4.3.8 c'è la deployment view prima e dopo i refactoring. Confrontando le due Static View possiamo notare come i component accounts e items siano stati divisi e le Operation in loro contenute, rispettivamente *getAll()* per il component items mentre *signup( )* per il component accounts, sono state spostate nei "nuovi" component. Dal confronto fra le due Deployment View invece si nota come gli artifacts items e accounts siano stati "splittati".

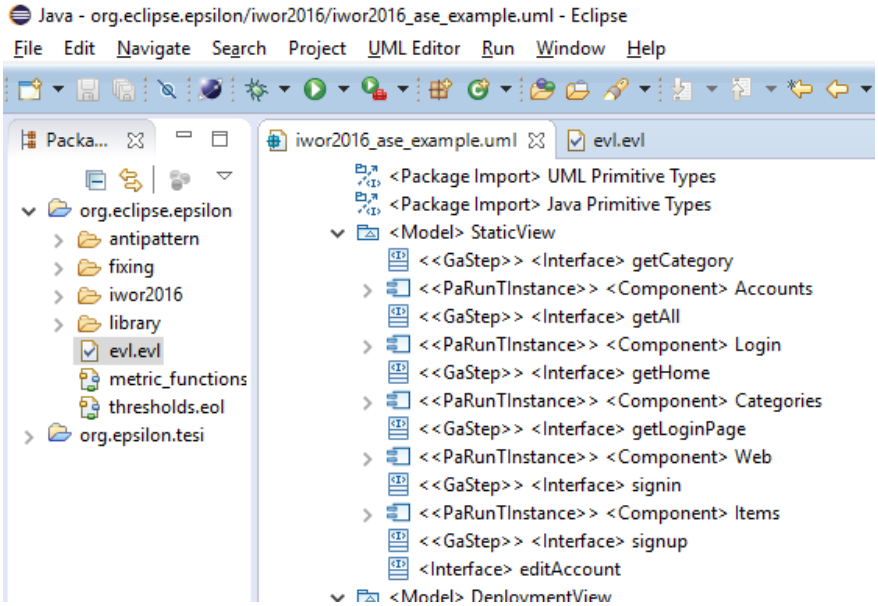


Figura 4.3.5 - Static View prima del refactoring

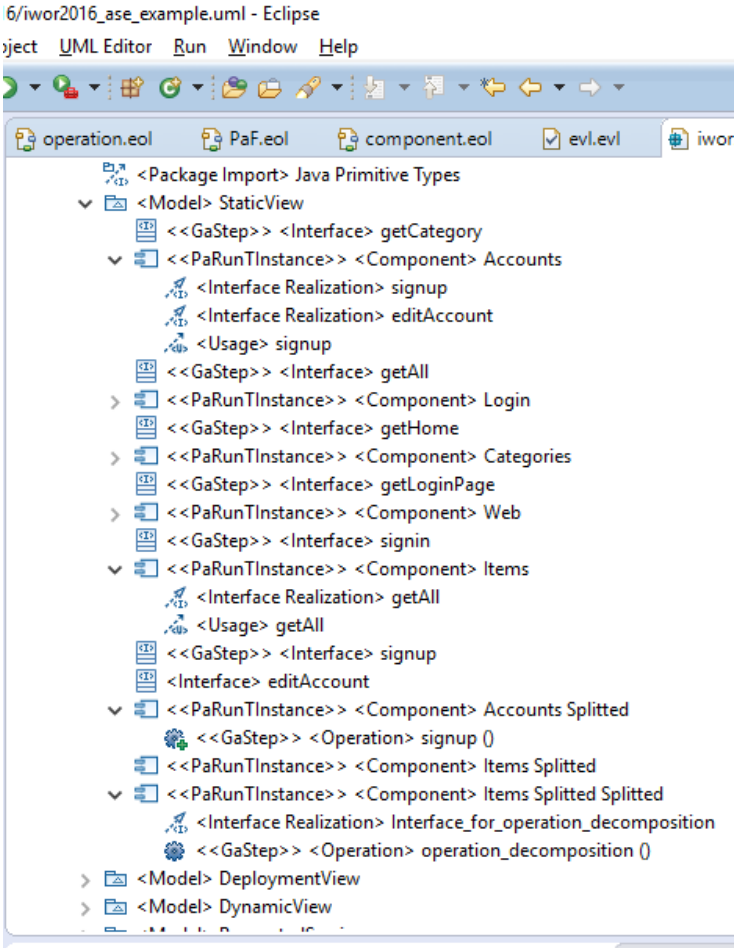


Figura 4.3.6 - Static View Dopo i refactoring

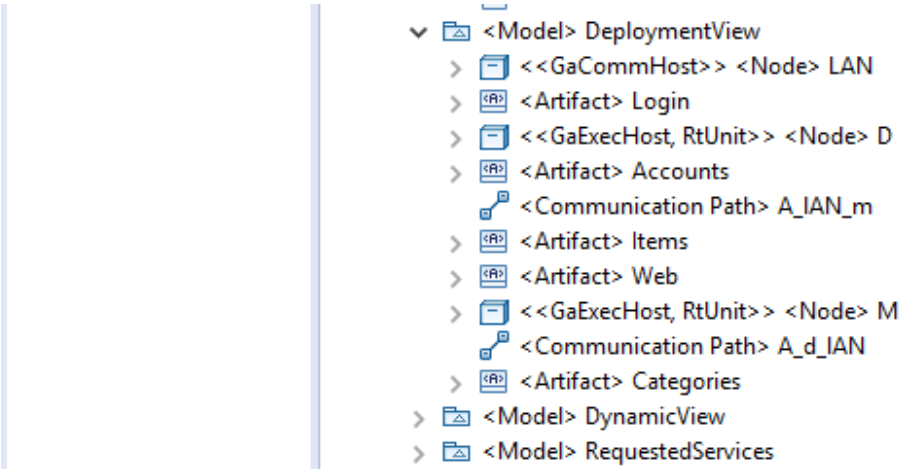


Figura 4.3.7 - Deployment View prima del refactoring

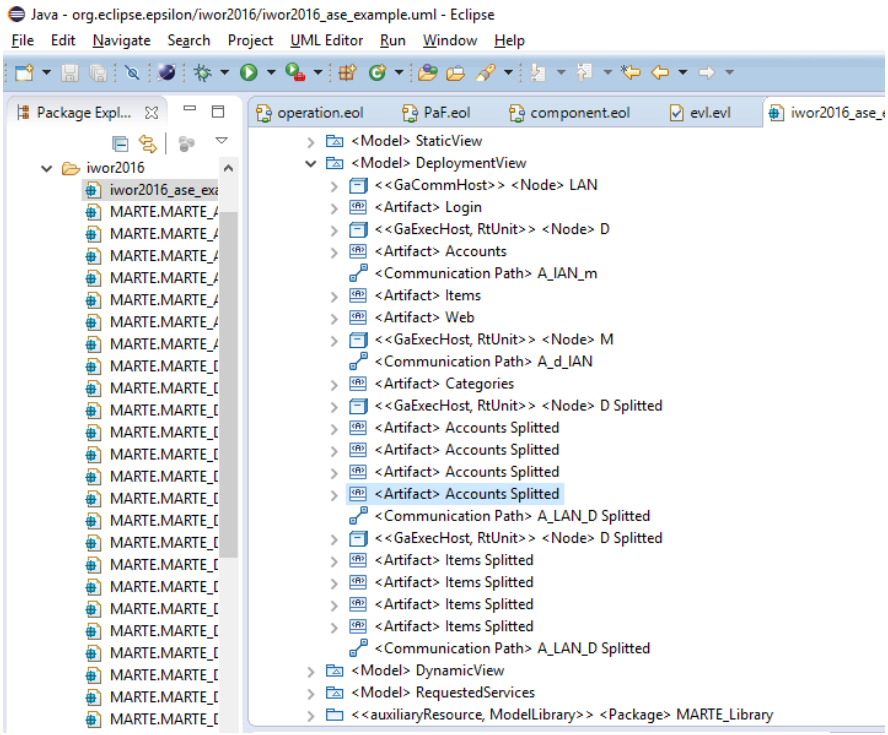


Figura 4.3.8 - Deployment View dopo il refactoring

Nelle Figure 4.3.9 e 4.3.10 è illustrata la Operation getAll() prima e dopo il refactoring. Da questo confronto si deduce tramite il confronto dei rispettivi parametri che la Operation è stata semplicemente distribuita in maniera differente, dato che i sopra citati parametri di performance sono gli stessi.

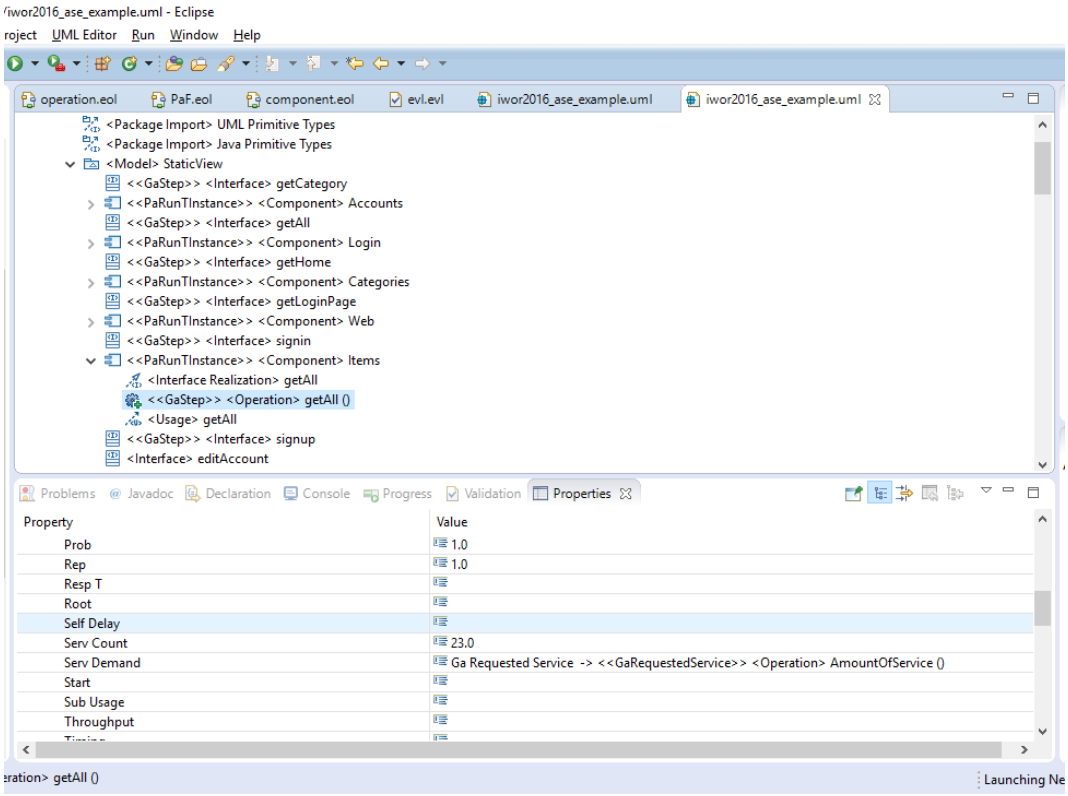


Figura 4.3.9 - Operation getAll() prima del refactoring

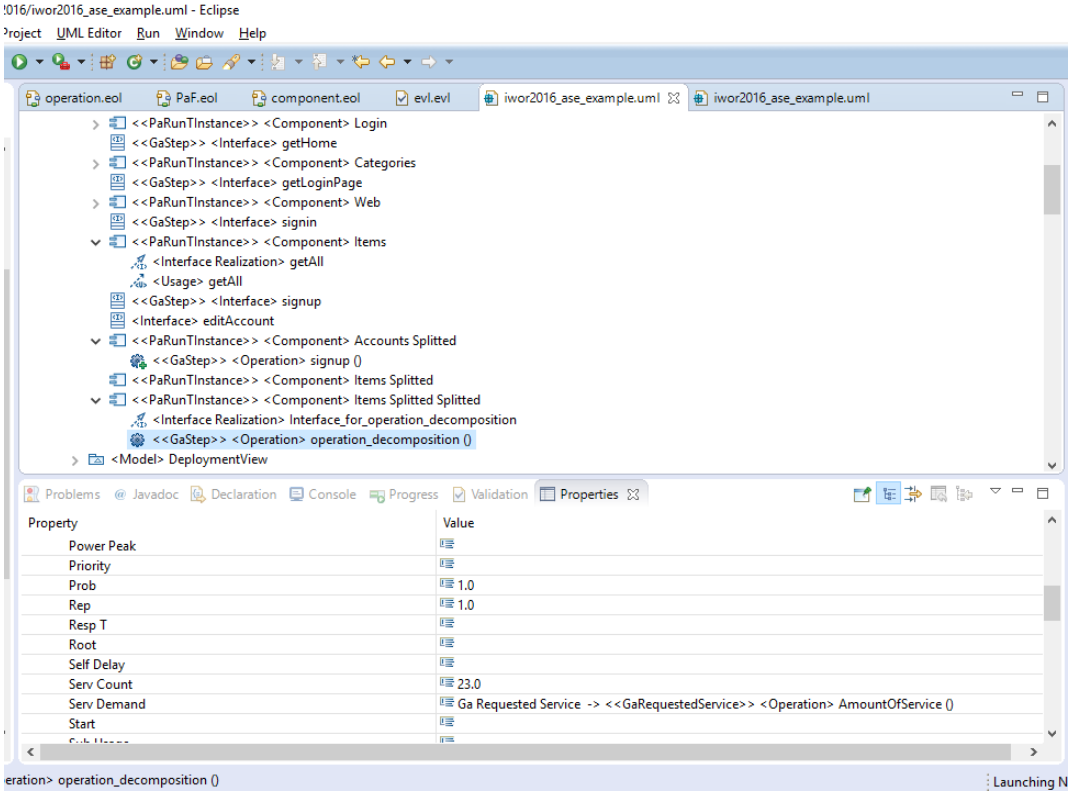


Figura 4.3.10 - Operation getAll() dopo il refactoring

Nelle Figure 4.3.11 e 4.3.12 sono illustrati un Component Diagram e un Deployment Diagram del modello rifattorizzato che illustrano la Static view e la Deployment view nelle figure 4.3.6 e 4.3.8.

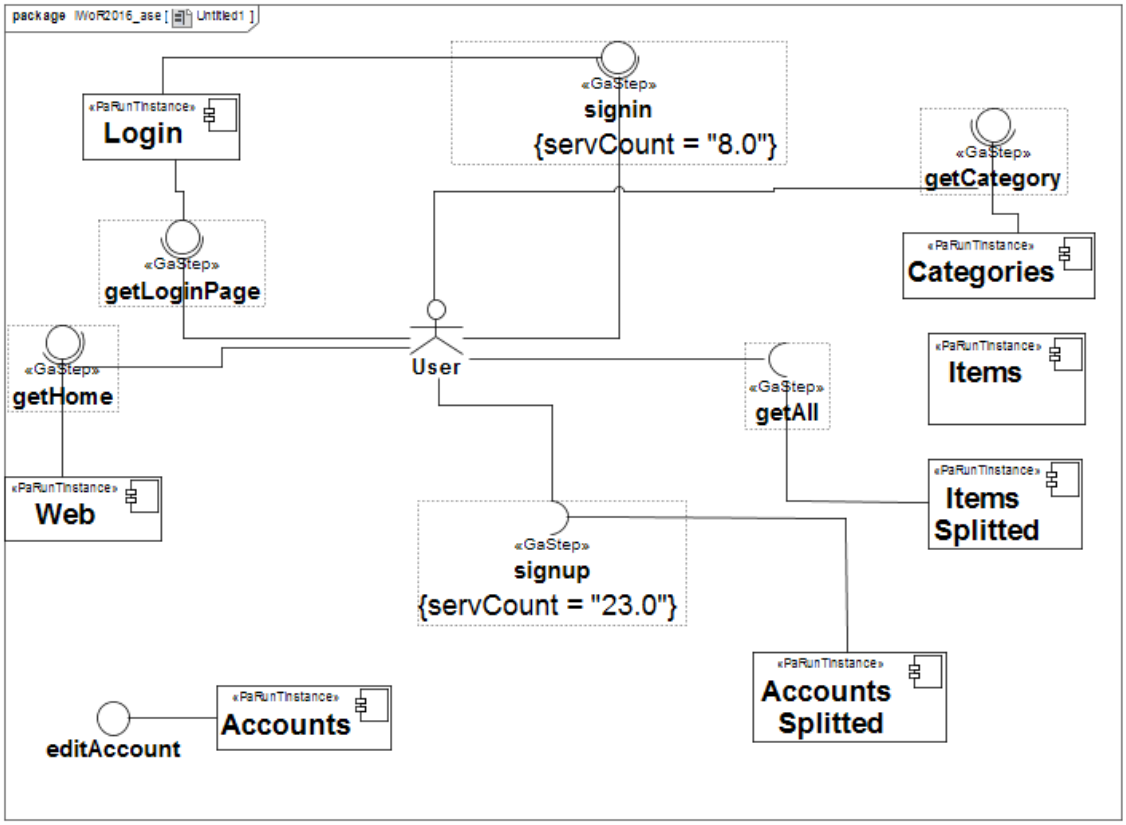


Figura 4.3.11 – Component Diagram del modello rifattorizzato

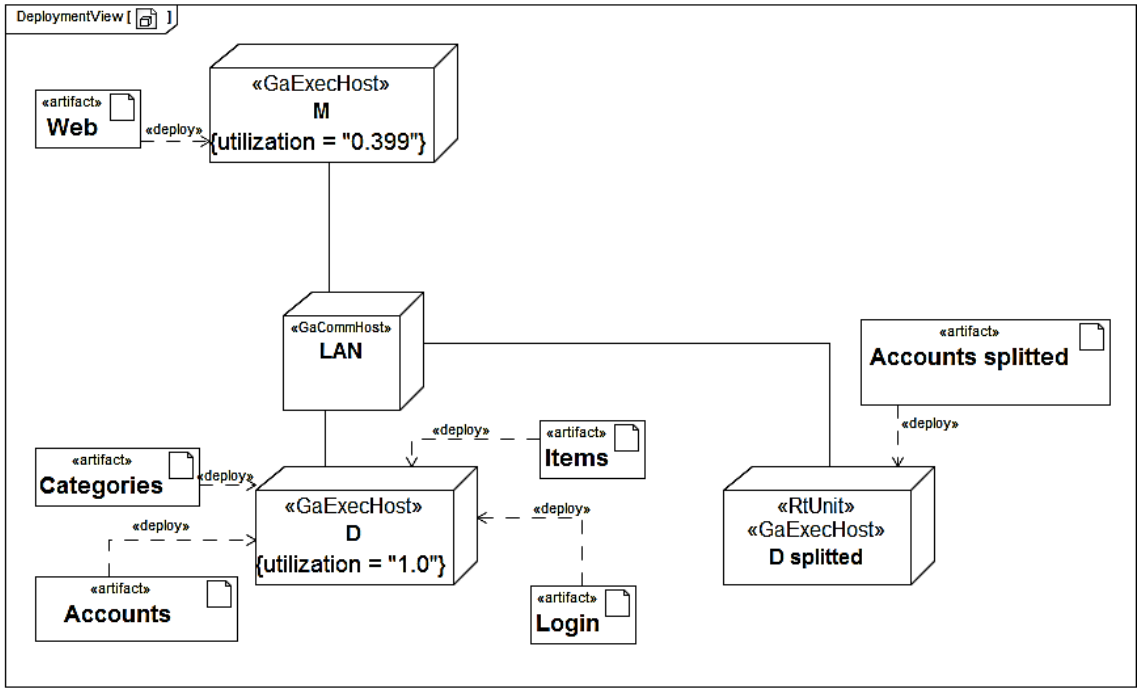


Figura 4.3.12 – Deployment Diagram del Modello rifattorizzato

5. Conclusioni e direzioni future

Per questa tesi è stato sviluppato un plugin per la piattaforma Eclipse che agevola la programmazione di un file EVL per il rilevamento e l'eventuale refactoring di performance antipattern su modelli. Sono stati discussi gli scopi che hanno portato allo sviluppo di questo tool, ed il plugin è stato descritto in tutti i suoi aspetti e le sue funzionalità. Infine ne è stato illustrato l'utilizzo mediante un caso d'uso in cui il codice generato è stato eseguito su un modello che presentava dei performance antipattern e per questo su esso sono stati effettuati dei refactoring al fine di rimuoverli per migliorare le performance del modello stesso, come mostrato nel caso di studio proposto.

Un obiettivo futuro del presente lavoro di tesi è quello di integrare all'interno del plug-in sviluppato una funzionalità di *traduzione in EPL ed EWL* del file generato in EVL, come descritto nel lavoro in [8]. EPL ed EWL sono due linguaggi Epsilon che nascono per scopi differenti fra loro e dunque hanno struttura e sintassi differente. Tuttavia, le loro differenti semantiche di esecuzione (ovvero i differenti modi in cui i linguaggi sono interpretati dagli interpreti in Epsilon) possono essere opportunamente sfruttate per ottenere differenti supporti al refactoring di modelli software guidato da performance antipattern. Per esempio, EPL abiliterebbe sessioni di refactoring in cui una sequenza predefinita di regole di individuazione di antipattern verrebbe verificata sul modello in input e, ad ogni occorrenza, la corrispondente azione di refactoring sarebbe contestualmente eseguita. EWL, invece, abiliterebbe l'individuazione di performance antipattern direttamente nell'editor di modellazione, ad ogni click destro su un elemento del modello per il cui tipo esistono regole di detection; ad ogni occorrenza di un antipattern, sarebbero dunque abilitate le azioni di refactoring previste, una delle quali può essere scelta dall'utente ed applicata, ottenendo un nuovo modello.

L'integrazione di una funzionalità di traduzione EVL-EPL ed EVL-EWL del file generato sarebbe dunque un'opportunità per ampliare il supporto al refactoring del plug-in sviluppato. A tal fine, sarà necessario individuare ed implementare dei mapping sintattici e semantici fra i costrutti dei tre linguaggi considerati.

Un ulteriore sviluppo futuro del presente lavoro di tesi consiste nell'introduzione di una interfaccia grafica per l'ampliamento del database che permetta di aggiungervi Function, Threshold e Do. Sarebbe infine molto interessante rendere il database pubblico, in modo

da avere un repository di riferimento in cui memorizzare e condividere contenuti, favorendo modularità e riuso.

Bibliografia

[1] D. Arcelli, V. Cortellessa, D. Di Pompeo – **Towards a Unifying Approach for Performance-Driven Software Model Refactoring.** In MPM Workshop (2015). B. Combemale, J. De Antoni, J. Gray, D. Balasubramanian, B. Barroca, S. Kokaly, G. Mezei, P. Van Gorp Eds. CEUR-WS.org, pp. 42-51.

[2] Dimitris Kolovos, Louis Rose, Antonio García-Domínguez, Richard Paige: **The Epsilon Book**. (2016).

[3] Eclipse: http://www.eclipse.org/

[4] W. J. Brown, R. C. Malveau, H. W. M. Ill, T. J. Mowbray: **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. 1998.

[5] C. U. Smith, L. G. Williams: **Software performance antipatterns**. In Workshop on Software and Performance (2000), pp. 127–136.

[6] C. U. Smith, L. G. Williams: **New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot**. In International Computer Measurement Group Conference (2002), pp. 667–674.

[7] C. U. Smith, L. G. Williams: **More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot**. In International Computer Measurement Group Conference (2003), pp. 717–725.

[8] D. Arcelli, V. Cortellessa, D. Di Pompeo – **Automated translation among EPSILON languages for performance-driven UML software model refactoring.** In 1st International Workshop on Refactoring (2016). Accepted for publication.