ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SECONDA FACOLTÀ DI INGEGNERIA
SEDE DI CESENA

CORSO DI DIPLOMA UNIVERSITARIO IN INGEGNERIA

INFORMATICA E AUTOMATICA

ELABORATO

In

INGEGNERIA DEL SOFTWARE

Analisi e sviluppo di uno strumento di verifica di consistenza a supporto della metodologia SODA

CANDIDATO:
ALAN FOLIGATTI

RELATORE:
Chiar.mo Prof. Ing. ANDREA OMICINI

CORRELATORE:
Dott. Ing. AMBRA MOLESINI

Anno Accademico 2010/2011
Sessione III

Ringraziamenti

Ringrazio il Prof. Andrea Omicini e l'Ing. Ambra Molesini per la professionalità con cui mi hanno supportato, durante lo svolgimento di questa tesi.

Ringrazio la mia famiglia ed in particolare la mia compagna , che in questi anni mi ha sempre sostenuto incoraggiandomi a non mollare mai.

Ringrazio gli amici e colleghi conosciuti durante il percorso Universitario, ed in particolare una cara amica.

Indice

INTRODUZIONE

Negli ultimi anni c'è stata una notevole crescita di interesse verso un paradigma computazionale noto come paradigma 'multi-agente'. Esso è l'oggetto di studio di una nuova area di ricerca che riunisce ed integra aspetti dell'Intelligenza Artificiale con altri tipici dei Sistemi Distribuiti, con l'idea di sviluppare modelli e architetture per agenti intelligenti.
In questo contesto si ha l'esigenza di analizzare, progettare e sviluppare strumenti che aiutino e siano da supporto alle metodologie di sviluppo di sistemi multi-agente complessi.

La metodologia SODA, si colloca in questo scenario con lo scopo di definire una guida nella progettazione di sistemi ad agenti, focalizzandosi sia sull'aspetto sociale di questi, sia sull'ambiente in cui il sistema andrà ad operare, trascurando invece la realizzazione delle caratteristiche intra-agente.
Il processo SODA, come del resto la maggior parte dei processi di questo tipo, per la descrizione di sistemi ad agenti, si articola in diverse fasi, con lo scopo di produrre un certo numero di tabelle, che servono a descrivere le caratteristiche relazionali degli agenti, nonché gli agenti stessi, contestualizzati nell'ambiente che li contiene.
Le tabelle prodotte dal processo, sono in genere numerose, creando una difficoltà oggettiva per il progettista, se quest'ultimo le dovesse organizzare manualmente; per questo motivo nasce la necessità di definire uno strumento a supporto della metodologia, per organizzare le fasi SODA, e la relativa documentazione prodotta.

Scopo di questa tesi è analizzare e sviluppare la parte preposta al controllo dell'integrità dei dati (Check&SODA), facente parte di un nuovo plugin per piattaforma Eclipse che supporti il progettista durante le attività di progettazione con SODA, tramite una componente grafica Graph&SODA.
Parallelamente il controllo (check) deve mettere a disposizione le risorse necessarie al fine di agevolare lo sviluppo ed il mantenimento delle tabelle SODA, che manualmente vengono manutenute tramite il modulo di gestione tabelle (Kit&SODA).

Le caratteristiche finali del dello strumento, che si viene a progettare, dovranno essere quelle di portabilità ed estendibilità, in un contesto completamente open-source.

A seguito di quanto detto, la tesi si compone del primo capitolo, in cui viene data una descrizione della piattaforma Eclipse relazionata al concetto di plug-in, che rappresenta l'elemento atomico, utile a definire nuovi strumenti di sviluppo che si integrano nella piattaforma stessa, aumentandone le funzionalità. Successivamente, nel secondo capitolo, viene descritta la metodologia SODA.

La definizione dei concetti sviluppati nei primi due capitoli, serve come base, per la progettazione del modulo di controllo, che troviamo definito nel terzo e quarto capitolo. Seguono le conclusioni nel capito 5.

1. ECLIPSE E PLUG-IN

Parlando di Eclipse, parliamo di un ambiente integrato (integrated development environment - IDE) di programmazione open-source scritto interamente in Java, per la realizzazione di progetti informatici scritti principalmente in Java ma anche in altri linguaggi tipo C++.

Eclipse si compone di un core-framework OSGi (Open Service Gateway initiative) sviluppato interamente sul set di bundle Equinox che rappresenta un esempio completo di sviluppo modulare, di componenti e servizi: tutto quello che racchiude la tecnologia OSGi.

1.1 FILOSOFIA OSGi

OSGi [1] è una specifica di gestione del ciclo di vita del software, orientata ai servizi (Service Oriented Architecture).
E' quindi definito uno standard, con il quale si può facilmente sviluppare software Java in modo modulare, dinamico e facile da mantenere; da qui l'esigenza di definire i bundle, vale a dire moduli software che tendono ad essere auto-contenuti, cioè indipendenti dagli altri moduli e dove non sia possibile l'indipendenza, allora definire attraverso un *manifest* le interfacce esportate ed importate, per dichiararle a tempo di build-time e deploy-time. In questo modo vengono risolti i problemi di class-loading, causati dall'interdipendenza dei moduli.

Da quanto esposto i framework OSGi sono piattaforme *Java-based* che, secondo un approccio *microkernel* a plug-in, forniscono le specifiche per sviluppare applicazioni che implementino servizi, permettendone la registrazione di nuovi, aggiornare o rimuovere gli esistenti a run-time, senza compromettere l'operatività della macchina, su cui stanno girando.
Un pregio della piattaforma sta nell'interoperabilità del modello di cooperazione tra bundle che prevede la possibilità di ricercare, individuare ed usare in maniera condivisa i servizi forniti da diverse applicazioni nell'ambito della stessa *virtual machine*, con conseguenti vantaggi in termini di prestazioni e consumo delle risorse.

In sintesi possiamo vedere OSGi come:

- un sistema modulare per la piattaforma Java.
- un sistema dinamico, il quale consente l'installazione, l'avvio, lo stop e la rimozione a runtime.
- orientato ai servizi, i quali possono essere dinamicamente registrati ed utilizzati nella virtual machine.

OSGi si presenta con diverse implementazioni, per esempio Knopflerfish OSGio Apache Felix, Eclipse Equinox che è attualmente l'implementazione di riferimento delle specifiche OSGi.

1.2 STRUTTURA OSGi

Le funzionalità dei frameworks OSGi [2] si dividono nei livelli (Figura 1.1):

- Security layer
- Module layer
- Life Cicle layer
- Service layer
- Actual Service

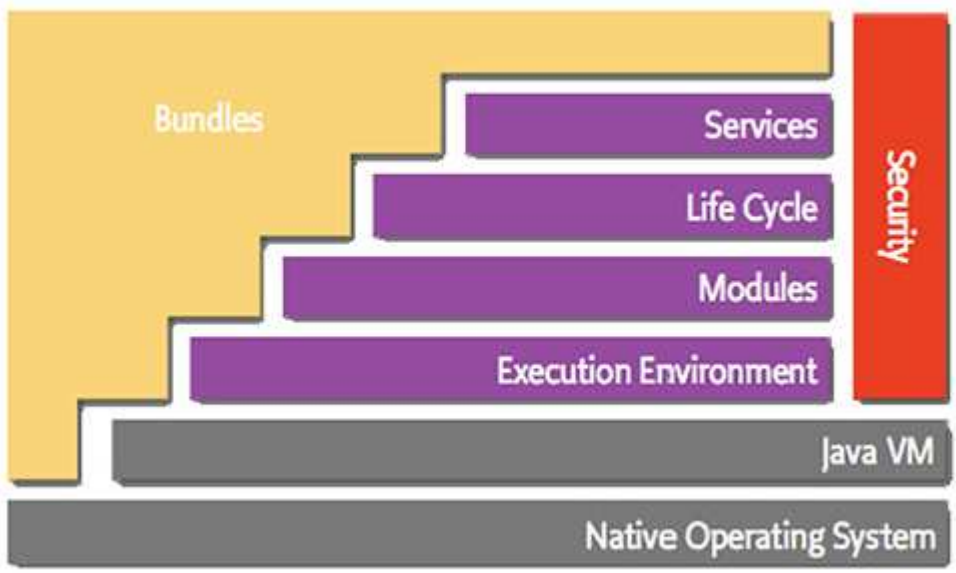


Figura 1.1: Architettura del framework OSGi.

1.3 BUNDLE E SERVIZI

Strutturalmente le applicazioni Java sono composte da classi organizzate in package, un programma Java cerca le classi di cui ha bisogno in un insieme di directory o file jar, tale insieme è definito dal "classpath".

Allo stesso modo il *bundle*, che indica un modulo per OSGi, è organizzato in package contenente classi ed altre risorse per fornire servizi, corredato di un *manifest.mf* file [1], che prevede oltre alle informazioni standard previste dal linguaggio, informazioni addizionali utili al framework OSGi per far si che l'archivio jar diventi appunto un bundle.

Di conseguenza un qualsiasi jar può diventare un bundle, ed un bundle può essere utilizzato come semplice jar dalla JVM, che ignorerà le informazioni eventualmente elencate nel file manifest.mf; taluni bundle sono caricati all'occorrenza tramite un class-loader.

Detto quindi che il manifest.mf può essere assimilato ad un file di configurazione, la prima caratteristica gestita è:

- l'indicazione delle interfacce di import-package ed export-package; il primo fornisce le informazioni su quali package devono essere importati perché il bundle funzioni correttamente, il secondo da indicazioni sui package che il bundle renderà visibili; tale meccanismo risolve problemi di interdipendenza tra i bundle. Il framework OSGi ha in questo modo la possibilità di conoscere subito i moduli che devono essere disponibili per il funzionamento di altri.

- Altro aspetto gestibile nel manifest.mf è il bundle-version, che permette di far convivere più versioni dello stesso jar all'interno della JVM, cosa che con Java non è possibile fare.

In Figura 1.2 è presente il contenuto di un manifest OSGi con le relative informazioni.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Popup Plug-in
Bundle-SymbolicName: de.vogella.rcp.intro.commands.popup; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: de.vogella.rcp.intro.commands.popup.Activator
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Figura 1.2 MANIFEST.MF

Bundle-Name è un breve testo descrittivo, mentre Bundle-SymbolicName è l'identificatore univoco per il pacchetto plug-in. Bundle-Version definisce la versione del bundle, che deve essere incrementata, se una nuova versione del bundle viene pubblicata.
Un bundle può definire un punto di attivazione via Bundle-Activator, rappresentato da una classe che definisce l'avvio e l'arresto del bundle.
Con il Bundle-RequiredExecutionEnvironment si specifica quale versione di Java è necessaria per eseguire il pacchetto, se questo requisito non è soddisfatto allora il runtime OSGi non carica il bundle.

Con le logiche evidenziate si ottiene la gestione delle dipendenze a livello di package, cosa non possibile in ambiente Java.

OSGi si propone come *architettura orientata ai servizi* (Service Oriented Architecture) all'interno della JVM; questo approccio è reso possibile attraverso il *registro dei servizi* (service registry), i bundle espongono i servizi identificati tramite interfaccia ed è possibile registrarli nel registro dei servizi.
I servizi a loro volta possono utilizzare altri servizi e se uno di questi non è disponibile, il bundle può aspettare sino a quando non si renda tale.

Come evidenziato di seguito (Figura 1.3), OSGi risulta essere un modello collaborativo dove i bundle possono collaborare attraverso i servizi e la condivisione organizzata dei package.

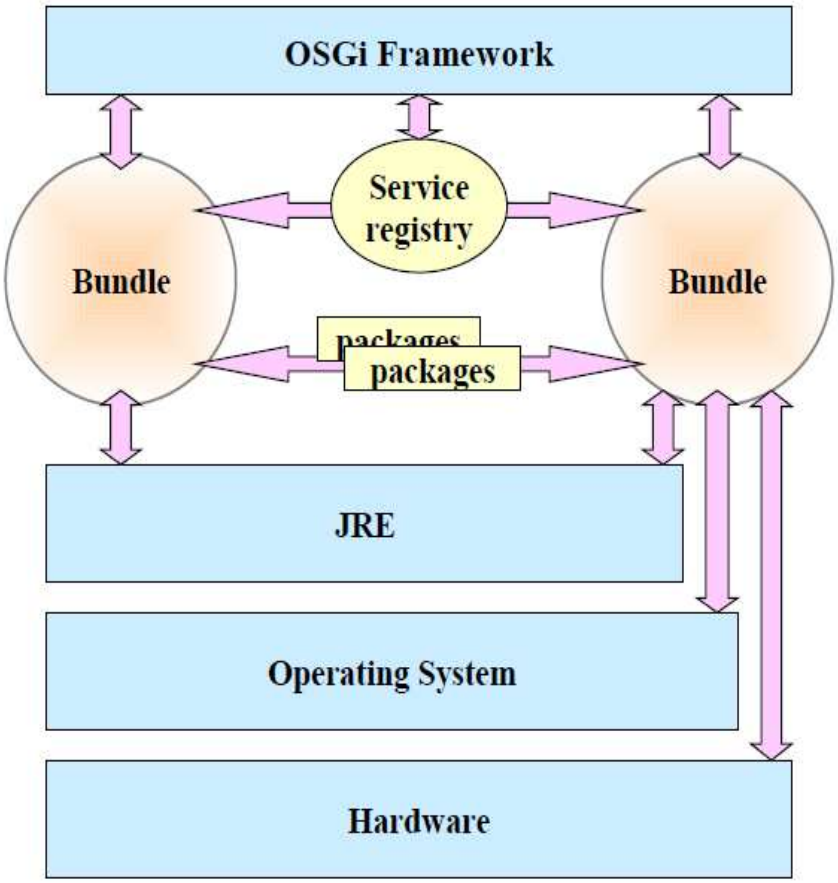


Figura 1.3 Modello collaborativo

La gestione delle dipendenze avviene a due livelli:

- Dipendenze dei bundle/package, che ritroviamo come: Require-Bundle (dipendenza con un intero bundle) oppure Import-Package (dipendenza con i package specificati)
- Dipendenze dei servizi.

Dai livelli indicati, possiamo distinguere le relazioni:

Bundle-to-package: un bundle può richiedere codice non contenuto nell'archivio jar, in questo caso importa il codice sotto forma di package:

- Il codice richiesto deve essere dichiarato esplicitamente nel file manifest del richiedente.
- Il codice richiesto deve essere esplicitamente esportato da un altro bundle.

- I bundle richiesti devono essere presenti nel framework, per permettere al richiedente di essere attivato.
- Questo tipo di dipendenza viene gestita dal framework.

  Bundle-to-service: un oggetto contenuto in un bundle può usare servizi esterni dichiarati nel manifest.

  Service-to-service: l'implementazione di un servizio può richiedere l'uso di un servizio registrato.

Ogni servizio all'interno del framework OSGi ha un identificatore univoco, tali servizi sono registrati tramite il BundleContext, passando il nome dell'interfaccia, l'implementazione del servizio e le proprietà; mentre la ricerca dei servizi viene fatta fornendo al framework il nome dell'interfaccia ed un filtro LDAP con le condizioni di ricerca.

Per LDAP (Lightweight Directory Access Protocol) si intende un protocollo standard per l'interrogazione e la modifica dei servizi di directory, ossia un insieme di programmi che provvedono ad organizzare e memorizzare informazioni su reti di computer e su risorse condivise disponibili tramite la rete.

Se la security è abilitata può essere necessario assegnare ai servizi gli opportuni permessi.

All'interno del framework è possibile riferirsi ai servizi attraverso la classe "ServiceReference" e per ogni servizio registrato nel framework viene creato un oggetto "ServiceRegistration".

1.4 LA PIATTAFORMA ECLIPSE

La piattaforma di Eclipse [3] è strutturata attorno al concetto di plug-in che rappresenta la più piccola unità di modularizzazione. Il plug-in e bundle nel contesto Eclipse sono intercambiabili, infatti un plug-in Eclipse è anche un bundle OSGi e viceversa.

I plug-in contengono codice e/o dati e/o documentazione, che contribuiscono alle funzionalità del framework; in figura 1.4 si può vedere la struttura della piattaforma Eclipse.

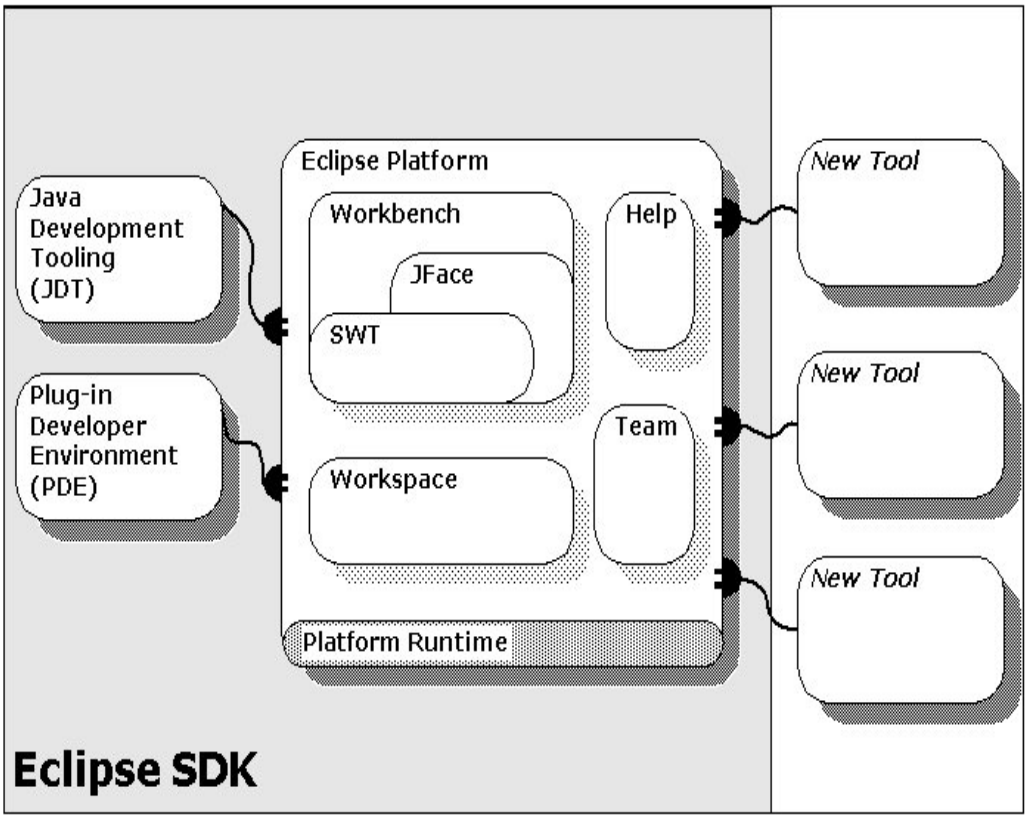


Figura 1.4 Struttura di Eclipse

Nei plug-in è possibile definire le:

- ESTENSIONI (*extension*): Specifica una contribuzione, vale dire uno sviluppo di un punto di estensione già presente.
- PUNTI DI ESTENSIONE (*extension-point*): Specifica un punto attraverso il quale è possibile specializzare una nuova funzionalità per il sistema.

La gestione delle funzionalità genera dei sottosistemi di plug-in, aventi come strato inferiore il core framework, che si occupa di gestire il ciclo di vita del codice soprastante organizzato nei bundle.

Genericamente la relazione tra le classi plug-in è illustrata di seguito (figura 1.5) [4]:

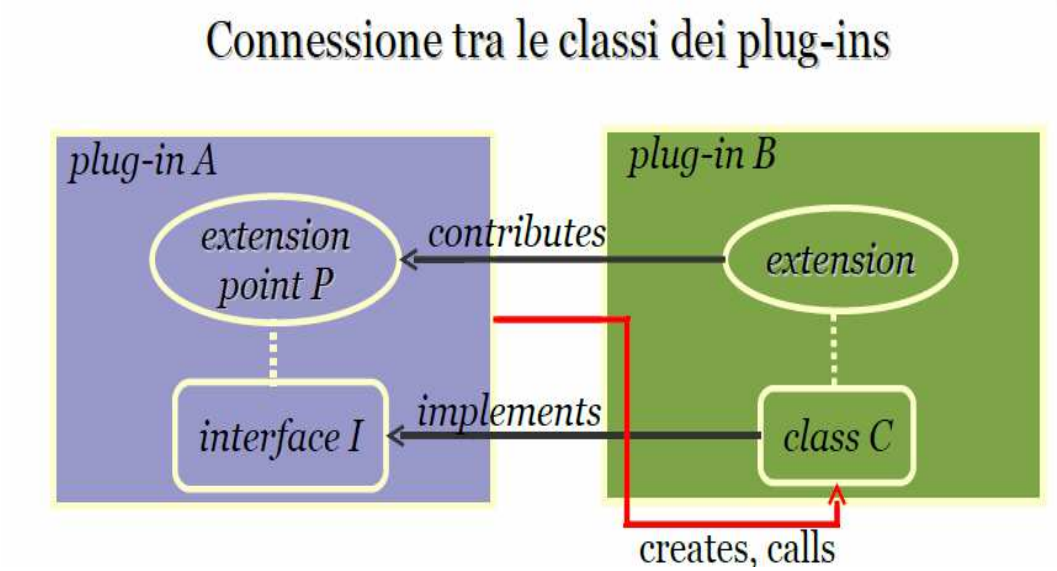


Figura 1.5 Legame tra extension-point ed extension.

Plug-in A:

- Dichiara l'extension-point P
- Dichiara l'interfaccia I associata a P

Plug-in B:

- Implementa l'interfaccia I mediante la classe C
- Contribuisce la classe C all'extension-point P

Il plug-in A crea C ed invoca i suoi metodi.

1.4.1 LE PARTI DELLA PIATTAFORMA ECLIPSE

Platform runtime: Si occupa di amministrare il ciclo di vita dei servizi, gestendone i registri. Definisce il primo livello dei punti di estensione; si occupa della ricerca dinamica dei componenti mantenendo aggiornato il loro stato.
In fase di start-up, la piattaforma run-time individua il set di plug-in disponibili, legge i loro manifest, e costruisce il registro plug-in di sistema in memoria. Eventuali problemi, come ad esempio punti di estensione mancanti, sono rilevati e registrati. Il registro plug-in di sistema è disponibile tramite l'API della piattaforma. I plug-in possono essere aggiunti, sostituiti o cancellati dopo l'avvio.

WorkSpace: è costituito da uno o più progetti di livello superiore, dove ogni progetto trova corrispondenza in una directory specificata dall'utente nel file system. Ogni progetto a sua volta è strutturato in package, che trovano corrispondenza in sottodirectory del file system.
Tutti i file nell'area di lavoro sono direttamente accessibili ai programmi e strumenti standard del sistema operativo sottostante.
Per ridurre al minimo il rischio di perdere i file accidentalmente, un meccanismo di basso livello tiene traccia del precedente contenuto di qualsiasi file che è stato modificato o eliminato da strumenti integrati.
L'area di lavoro fornisce un meccanismo di marcatori (marker) per l'annotazione delle risorse. I marcatori sono utilizzati per registrare annotazioni diverse, come messaggi di errore del compilatore, le voci “to-do” di elenco, i segnalibri, i punti di interruzione e debugger.

Workbench: L’interfaccia utente (User Interface) di Eclipse è costruita intorno al Workbench che fornisce la struttura generale e la possibilità di estensioni. Il Workbench si compone di due toolkit:

SWT: un set di widget e la libreria grafica integrata con il sistema nativo a finestre.

JFace: un kit di strumenti di interfaccia utente implementata utilizzando SWT comune che semplifica le attività di programmazione dell'interfaccia utente.

SWT: Standard Widget Toolkit definito attraverso una API indipendente, permette la gestione delle finestre, per l’integrazione con il sistema operativo.
La UI di Eclipse è realizzata con SWT.

Java SWT permette la gestione a basso livello dei widgets per il controllo delle liste, text fields e bottoni, ma non di quelli ad alto livello come alberi o rich-text. Internamente, l'attuazione SWT fornisce implementazioni separate e distinte in Java per ogni sistema a finestre.

JFace: JFace è un toolkit di interfaccia utente con classi per la gestione di molte comuni attività di programmazione dell'interfaccia utente. Le API di JFace sono indipendenti dal sistema operativo, e sono progettate per funzionare con SWT senza nasconderne il contenuto.

JFace comprende i componenti UI toolkit di registri di immagine, di dialogo e le preferences, la registrazione delle operazioni a tempo di esecuzione. Due delle sue caratteristiche più interessanti sono le *actions* e gli *viewers*. Le prime permettono di definire i comandi per l'utente, mentre le seconde permettono la presentazione dei dati in applicazioni strutturate quali liste, tabelle o alberi.

A differenza di SWT e JFace, che sono entrambe toolkit per l'interfaccia utente generale, il Workbench offre la personalizzazione dell'interfaccia utente della piattaforma Eclipse, e fornisce le strutture in cui gli strumenti interagiscono con l'utente. A causa di questo ruolo centrale, il Workbench è sinonimo della piattaforma Eclipse essendo l'interfaccia con la finestra principale che l'utente vede quando la piattaforma è in esecuzione (vedi Figura 1.6).

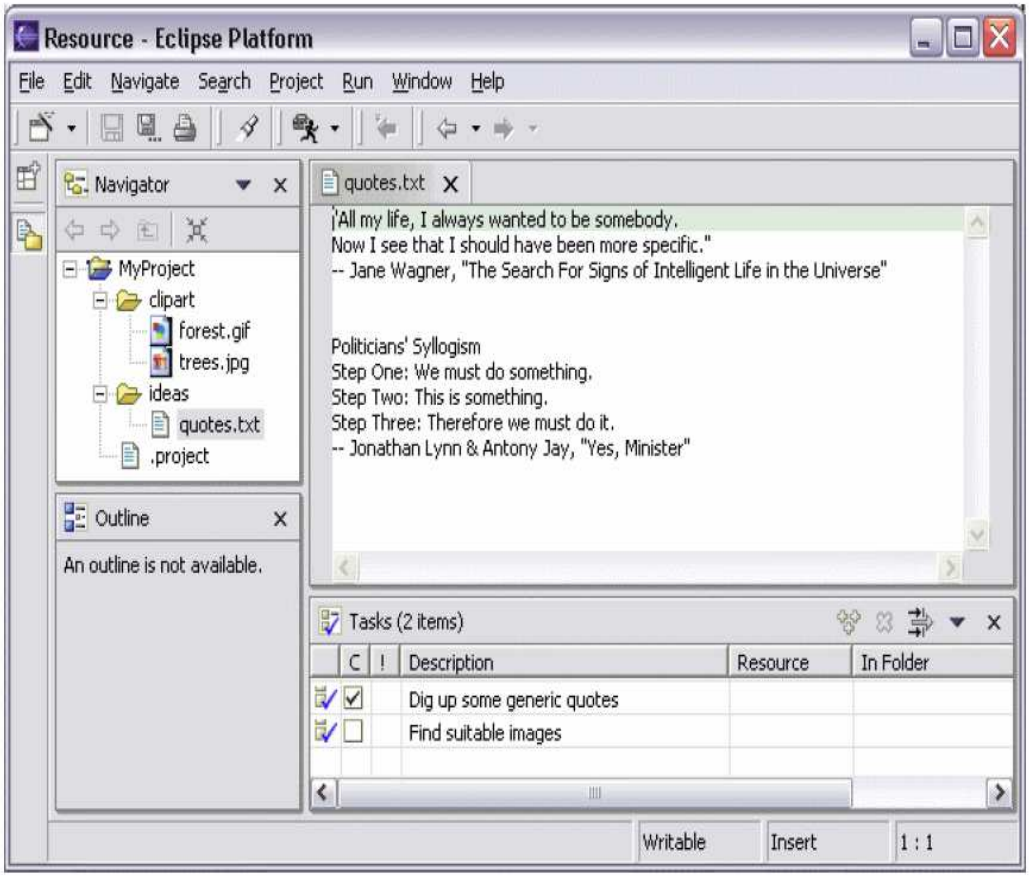


Figura 1.6 Sono presenti editor, view e perpectives. *L'editor* permette di aprire, editare e salvare oggetti. *View* provvede a fornire informazioni relativamente agli oggetti sui quali si lavora. *Perspective* rappresenta una sezione contenente sia editor e/o view.

Team: definisce un modello di programmazione utilizzabile quando lo sviluppo viene eseguito da più persone, consentendone di gestire le risorse comuni ed il versioning.

Help: Meccanismo che consente di definire gli strumenti di documentazione e contribuire all'uso di uno o più libri online.

Java development toolkit (JDT): Eclipse è dotato di un sistema di sviluppo Java, che consente di scrivere , modificare, testare e compilare progetti Java. JDT è implementato attraverso un gruppo di plug-in, con interfacce UI-plug-in ed una infrastruttura non-UI, sviluppata in un core a parte; la separazione fra UI e non-UI permette una gestione più dinamica delle configurazioni a basso livello della JTD, svincolando la parte grafica, che può essere personalizzata da terze parti.

Di seguito in figura 1.7 sono riportate le relazioni che fanno anche da punti di connessione, tra le parti del Workbench e la JDT UI, JDT Core.

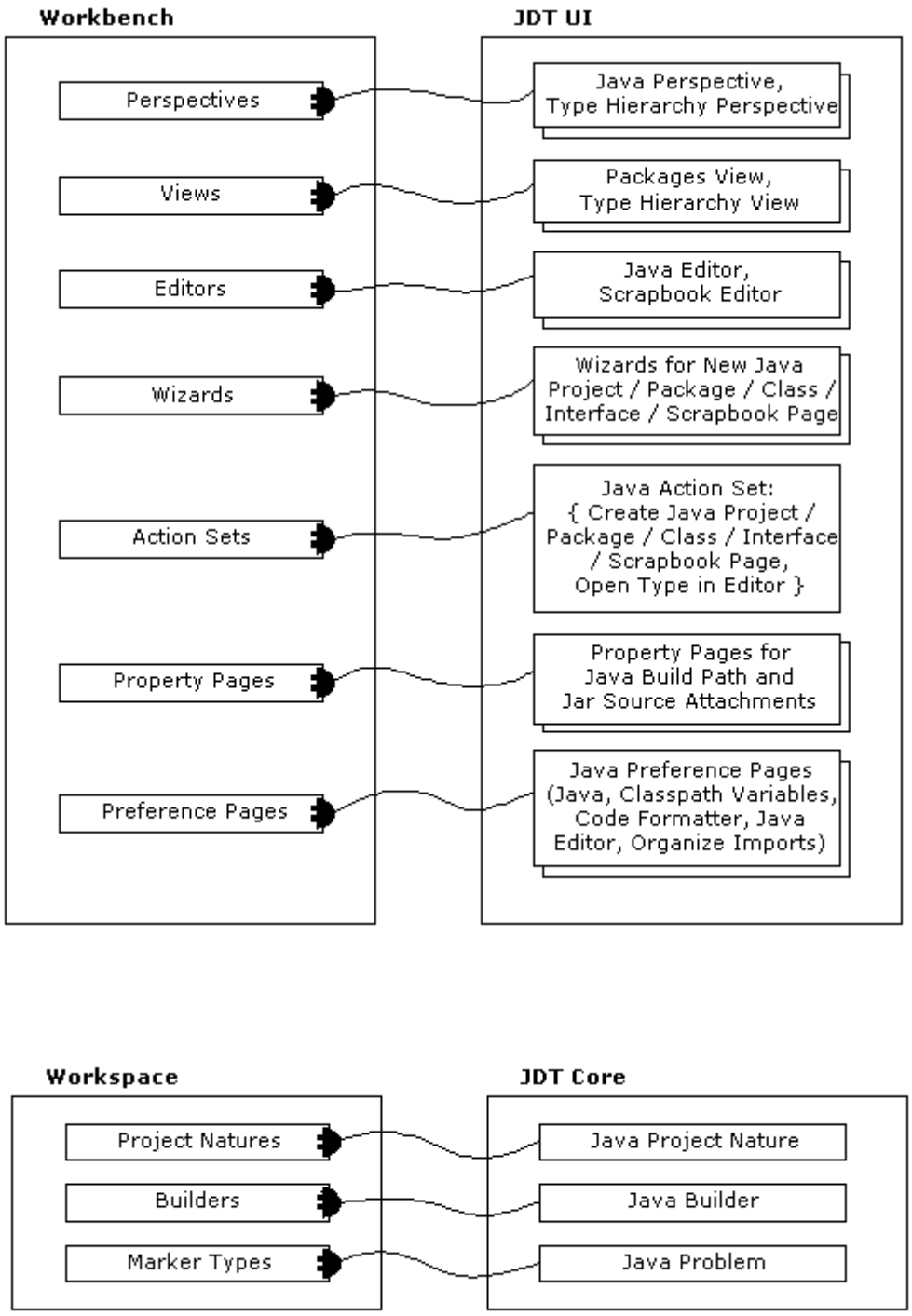


Figura 1.7 punti di connessione JDT ed Eclipse platform

Dalla figura si vede chiaramente che sul lato del Workspace risiedono gli extension-point, che vengono estesi tramite l'interfaccia JDT UI e JDT Core sulla sinistra.

E' presente anche la JDT Debug che mette a disposizione supporto specifico per il lancio di programmi ed il debug per il linguaggio Java.

JDT fa uso di diversi punti di estensione della piattaforma, aggiungendo funzionalità specifiche per Java, le quali si presentano sotto forma di viste, editor e azioni apposite per Java.
Per fare questo JDT si appoggia sul Java Development Kit (JDK), incapsulandone le funzionalità. JDK è il sistema software rilasciato alle origini da Sun, successivamente acquisita da Oracle, per la programmazione basata sulla piattaforma Java.

Plug-in Development Environment (PDE): rappresentato anch'esso da un plug-in, definisce l'ambiente di sviluppo, modifca, debug , compilazione e deploy dei plug-in.

1.4.2 PLUG-IN VISTI DA VICINO - MODELLO

Un plug-in di Eclipse è un componente che fornisce un certo tipo di servizio nel contesto del Workbench Eclipse [5].
In questo contesto gli oggetti possono essere configurati all'interno del sistema a tempo di esecuzione. Il Runtime Eclipse fornisce un'infrastruttura per sostenere l'attivazione ed il funzionamento di una serie di plug-in che lavorano insieme per fornire un ambiente unico per le attività di sviluppo. All'interno di un'istanza in esecuzione, un plug-in si attiva tramite la relativa classe che lo costituisce. La classe plug-in Eclipse deve estendere "org.eclipse.core.runtime.Plugin" abstract, che fornisce servizi generici per la gestione delle istanze dei plug-in.

Un'installazione Eclipse include una cartella denominata "plugins" in cui vengono distribuiti i singoli plug-in.
Ogni plug-in è descritto in un file manifesto XML, chiamato PLUGIN.XML, il quale indica le risorse da attivare a runtime e le relazioni con altri plug-in.

I contenuti dei file manifesto vengono messi a disposizione dal codice tramite il registro API di sistema. Le specifiche analizzate vengono memorizzate nella cache chiamata *registri plug-in system*.

Di seguito è mostrato un esempio di plugin.xml (Fig. 1.8), in cui possiamo vederne la struttura base.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  name="JUnit Testing Framework"
  id="org.junit"
  version="3.7"
  provider-name="Eclipse.org">
  <runtime>
    <library name="junit.jar">
      <export name="*"/>
    </library>
  </runtime>
</plugin>
```

Listing 2.1. A Minimal Plug-in Manifest File.

Figura 1.8 plugin.xml

Nello specifico riportato in figura viene gestito il servizio di JUnit.

Il plugin.xml descrive come il plug-in estende la piattaforma, quali sono gli extension-point pubblici e come implementa le funzionalità.
Il manifesto è scritto in xml, e viene analizzato (parsato) al caricamento del plugin, da parte della piattaforma. Tutte le informazioni del plugin, relative all'interfaccia UI, che servono per la visualizzazione delle icone, menu ed altro, sono riportate nel manifest (plug-in xml). Il codice implementato per la gestione delle funzionalità, è situato in un jar separato e caricato al momento dell'attivazione del plugin. In Figura 1.8 le istruzioni cominciano con le indicazioni dell'xml version e lo standard di decodifica dei caratteri. Il tag "plugin" rappresenta la radice, nella quale vengono racchiuse le informazioni che identificano il plug-in, con il suo nome, il package in cui si trova, la versione e l'id che lo identifica in modo univoco.
Nella parte di run-time, sono indicate le librerie che servono al plug-in per poter funzionare.

1.4.2.1 ESTENSIONI

L’aggiunta di nuovi servizi per l'utente, implica che uno o più elementi dell'interfaccia utente vengano aggiunti al Workbench Eclipse, ad esempio l’aggiunta di un help utente richiede l’inserimento del menù all’interno dell’interfaccia grafica di Eclipse.

L’aggiunta di elementi nuovi che estendono le funzionalità sono detti estensioni (extension).
Un'estensione è dichiarata da un *extender plug-in* e può richiedere a sua volta una serie di plug-in per modificarne il comportamento. Una modificazione comportamentale comprende l'aggiunta di elementi di elaborazione per l’*host plug-in* (plug-in che deve essere esteso).

Qualsiasi plung-in può essere esteso in differenti modi; per esempio l’interfaccia utente del Workbench può essere ampliata, in ogni caso, l’estensione deve essere conforme ad unico insieme di requisiti di configurazione e avere un unico insieme di caratteristiche comportamentali.
In questo ambito un plug-in fornisce più punti di estensione (*extension-point*) definendo quindi un insieme di nuovi comportamenti coerenti con la funzionalità d’origine del plug-in.

Una estensione può aggiungere anche un *calback object***,** vale a dire oggetti Java che servono per passare funzioni come parametri tra il chiamante (host plug-in) ed il chiamato (extension). In genere sono classi speciali che rappresentano un singleton, che estendono la classe “Plugin” o la sottoclasse “AbstractUiPlugin”, fornendo i metodi:

- *getStateLocation():* fornisce il percorso della cartella in cui il plug-in può scrivere dati persistenti.
- *openStream():* fornisce un input stream per un file che è relativo alla directory di installazione del plug-in.
- *getPluginPreferences():* fornisce una tabella chiave-valore di informazioni di base.
- *getImageRegistry():* da accesso ad un registro di immagini condivise.
- *getDialogSettings():* tabella persistente di settaggi per i vari dialogs e wizards.

- *getWorkbench():* metodo di servizio che fornisce l'istanza dell'IWorkbench corrente.

In Figura 1.9 sono illustrati i rapporti tra gli elementi di una estensione, in questo caso, l'estensione del Workbench Eclipse e le voci di menu del sistema di help di Eclipse. In questa estensione, l'host di plug-in è l'interfaccia utente di Eclipse Workbench "org.eclipse.ui", il cui menù può essere esteso tramite un'estensione del punto conosciuto come "actionSets" (che è l'extension-point).
Il plug-in che estende è l'interfaccia utente del sistema di help di Eclipse "org.eclipse.help.ui". Al fine di rendere disponibili funzioni di help per l'utente, l'interfaccia utilizza l'estensione "actionSets" per estendere le voci dell'interfaccia del Workbench tra i quali, 'Aiuto -> Argomenti' (tramite la classe 'HelpContentsAction') della guida e della 'ricerca-> Guida' (tramite la classe 'OpenHelpSearchPageAction'). L'estensione è definita dall'extender plug-in.

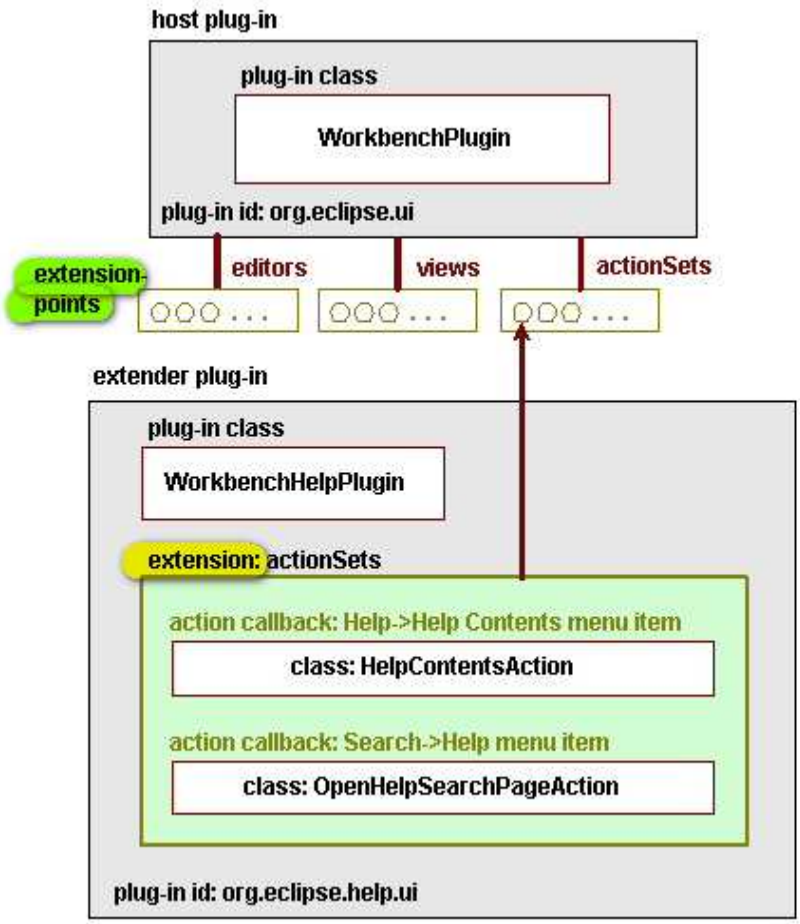


Figura 1.9 Gli elementi di una estensione. Il pug-in del Workbench viene esteso attraverso l'actionSet, che definisce le specifiche relative all'help in linea.

1.4.3 ELEMENTI CHE COSTITUISCONO UN'ESTENSIONE

Vediamo ora più da vicino il ruolo svolto dai vari oggetti che partecipano ad una estensione. Sono due gli ruoli da evidenziare, uno di *host* e l'altro di *extender*, poi abbiamo un generico oggetto di *callback*.

1.4.3.1 HOST PLUG-IN

Nel contesto di una particolare estensione, un plug-in che si distingue nel ruolo di host fornisce il punto di estensione che deve essere esteso. L'host plug-in è in grado di fornire servizi e di fungere da controllore alle sue estensioni.

All'interno del plug-in.xml dell'host, il punto di estensione viene identificato con un tag XML. Di seguito un esempio di come il Workbench Eclipse definisce un punto di estensione per la UI (punto 1):

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
1   id="org.eclipse.ui"
    name="Eclipse UI"
    version="2.1.0"
    provider-name="Eclipse.org"
    class="org.eclipse.ui.internal.UIPlugin">

2   <extension-point id="actionSets" name="Action Sets"
      schema="schema/actionSets.exsd"/>
    <!-- Other specifications omitted. -->
</plugin>
```

Listing 2.3. Declaring an Extension-Point.

Figura 1.9 plugin.xml

Per identificare il punto di estensione in modo univoco all'interno del contesto globale, il punto di estensione viene definito all'interno di un identificatore qualificato con il tag "extension-point". (punto 2).

1.4.3.2 EXTENDER PLUG-IN

In figura 1.10 è riportato l'esempio di un extender plug-in.

```
<plugin
        id="org.eclipse.help.ui"
        name="Help System UI"
        version="2.1.0"
        provider-name="Eclipse.org"
        class="org.eclipse.help.ui.internal.WorkbenchHelpPlugin">
    <!-- ... -->
    <!-- Action Sets -->
    <extension
        1 point="org.eclipse.ui.actionSets">
        <actionSet
                label="Help"
                visible="true"
                id="org.eclipse.help.internal.ui.HelpActionSet">
        2 <action
                label="&Help Contents"
                icon="icons/view.gif"
                helpContextId="org.eclipse.help.ui.helpContentsMenu"
                tooltip="Open Help Contents"
                class="org.eclipse.help.ui.internal.HelpContentsAction"
                menubarPath="help/helpEnd"
                id="org.eclipse.help.internal.ui.HelpAction">
            </action>
            <!-- ... other actionSet elements -->
        3 <action
                label="&Help..."
                icon="icons/search_menu.gif"
                helpContextId="org.eclipse.help.ui.helpSearchMenu"
            4 class="org.eclipse.help.ui.internal.OpenHelpSearchPageAction"
                menubarPath="org.eclipse.search.menu/dialogGroup"
                id="org.eclipse.help.ui.OpenHelpSearchPage">
            </action>
        </actionSet>
```

Figura 1.10 plugin.xml

Nell'esempio riportato vediamo che l'extender plug-in chiamato "org.eclipse.help.ui" (punto 2), viene configurato in modo da estendere l'actionSet dell'host (punto 1).

Al punto 1 ogni 'extension' tag è riferito ad un 'extension-point' tramite un ID che segue la convenzione sintattica dei packages Java.

Ai punti 2 e 3, le azioni indicate sono rese disponibili attraverso le voci di menu Workbench Help->Help Contents e Search->Help, rispettivamente.

Mettiamo anche in evidenza che le classi presenti all'interno dei tag 'action' rappresentano il *callback object* che mette in atto le azioni a seguito del verificarsi di un evento.

1.4.3.3 RELAZIONE TRA PLUG-IN ED EXTENSION OBJECTS

Le relazioni che intercorrono tra i due elementi sono:

- Multipli punti di estensione possono trovarsi nell'host plug-in.
- Un plug-in può essere al tempo stesso un host ed una estensione di altri plug-in.
- Più plug-in possono estendere il medesimo punto di estensione.
- Un plug-in che estende può definire differenti estensioni, per gli host plug-in che va ad estendere.
- Un plug-in che estende può definire multipli oggetti di ritorno all'host plug-in esteso.

1.4.3.4 STRUTTURA DI UN EXTENSION-POINT (SCHEMA)

Quando in un host plug-in si definisce un punto di estensione, oltre a dichiarare l'estensione vera e propria nel file plugin.xml, serve anche la definizione della sintassi da utilizzare.

Lo schema di seguito riportato rappresenta la definizione della grammatica con cui il file di manifest (plugin.xml), utilizza i tag per la parametrizzazione dei punti d'estensione.

Il file in questione viene salvato come ".exsd".

```
<schema targetNamespace="org.eclipse.ui">
   <element name="extension">
      <complexType>
         <sequence>
            <element ref="actionSet" minOccurs="1" maxOccurs="unbounded"/>
         </sequence>
         <attribute name="point" type="string" use="required"> </attribute>
         <attribute name="id" type="string"> </attribute>
         <attribute name="name" type="string"> </attribute>
      </complexType>
   </element>
   <element name="actionSet">
      <complexType>
         <sequence>
            <element ref="menu" minOccurs="0" maxOccurs="unbounded"/>
            <element ref="action" minOccurs="0" maxOccurs="unbounded"/>
         </sequence>
         <attribute name="id" type="string" use="required"> </attribute>
         <attribute name="label" type="string" use="required"> </attribute>
         <attribute name="visible" type="boolean"> </attribute>
         <attribute name="description" type="string"> </attribute>
      </complexType>
   </element>
   <element name="action">
      <complexType>
         <choice>
            <element ref="selection" minOccurs="0" maxOccurs="unbounded"/>
            <element ref="enablement" minOccurs="0" maxOccurs="1"/>
         </choice>
         <attribute name="id" type="string" use="required"> </attribute>
         <attribute name="label" type="string" use="required"> </attribute>
         <attribute name="toolbarPath" type="string">
         <attribute name="icon" type="string"> </attribute>
         <attribute name="tooltip" type="string"> </attribute>
         <attribute name="class" type="string"> </attribute>
      </complexType>
   </element>
</schema>
```

Listing 2.5. Extension-Point Schema Definition (.exsd File).

Figura 1.11 Schema XSD.

Lo schema definisce un “actionSets” come elemento che include alcuni attributi (‘label’, ‘icon’, …), ed una sequenza (<sequence>) che include menu ed azioni.

Il codice che da vita al servizio di estensione va inserito nel plug-in che estende l’host.

Ogni sequenza di elementi contiene più elementi membro, ad esempio l’estensione “actionsets” è una sequenza di più ‘actionSet’ membri, dove il numero può variare in un range delimitato da in minimo di occorrenze, ad un massimo di occorrenze (minOccurs-maxOccurs. Vedi punto 1 in figura 1.11).

1.5 IL PROCESSO ESTENSIONE

Sino ad ora il modello di estensione di Eclipse è stato presentato ad un livello alto, e sono state introdotte le specifiche dichiarative delle estensioni ed i loro oggetti. In questa sezione, vedremo come tali dichiarazioni sono trattate nella programmazione per sostenere gli obblighi dell’host plug-in. In altre parole, vedremo il tipo di codice host plug-in che i progettisti devono scrivere per ogni estensione.

Prendendo come riferimento il plug-in “org.eclipse.ui” ed il relativo “actionsSets extension-point”, nel momento in cui il plug-in viene attivato, le estensioni configurate vengono processate; nel caso specifico UI di Eclipse configura propriamente i bottoni, i menu e il callBack object che deve essere invocato.

1.5.1 OTTENERE IL REFERENCE DELL’ESTENSIONE TRAMITE IL PUNTO DI ESTENSIONE.

I punti di estensione attivati, i relativi schemi (grammatiche) ed altre informazioni che li denotano si trovano nell’host plug-in. Ad esempio le informazioni del plug-in “org.eclipse.ui” su come processare l’actionsSets extension dell’UI help plug-in “org.eclipse.help.ui”, risiedono su di esso.

Ogni plug-in per poter processare i propri punti di estensione, ha bisogno di ottenerne la lista dal core Eclipse run-time, e analizzare la giusta versione dei membri dell’estensione; per questo tipo di operazione il core Eclipse interroga il plug-in registry API.

Il codice di esempio seguente mostra come utilizzare il plug-in API registro di sistema, per iterare su tutti i membri di tutte le estensioni.

```
package com.bolour.sample.eclipse.demo;

import org.eclipse.core.runtime.IConfigurationElement;
import org.eclipse.core.runtime.IExtension;

public interface IProcessMember {
    public Object process(IExtension extension,
      IConfigurationElement member);
}

package com.bolour.sample.eclipse.demo;

import org.eclipse.core.runtime.IConfigurationElement;
import org.eclipse.core.runtime.IExtension;
import org.eclipse.core.runtime.IExtensionPoint;
import org.eclipse.core.runtime.IPluginRegistry;
import org.eclipse.core.runtime.Platform;

public class ProcessExtensions {
1   public static void process(String xpid, IProcessMember processor) {
        IPluginRegistry registry = Platform.getPluginRegistry();
        IExtensionPoint extensionPoint =
            registry.getExtensionPoint(xpid);
        IExtension[] extensions = extensionPoint.getExtensions();
        // For each extension ...
2       for (int i = 0; i < extensions.length; i++) {
            IExtension extension = extensions[i];
3           IConfigurationElement[] elements =
              extension.getConfigurationElements();
            // For each member of the extension ...
4           for (int j = 0; j < elements.length; j++) {
                IConfigurationElement element = elements[j];
5               processor.process(extension, element);
            }
        }
    }
}
```

Listing 3.1. Iterating over the Extensions and the Members of an Extension-Point.

Figura 1.12

La classe “Process Extensions” si occupa di processare i membri oggetto delle estensioni, analizzando tutte le estensioni relative ai punti di estensione e per ogni, analizzare tutti i relativi membri.

1.6 TIPOLOGIE DI PLUG-IN MESSE A DISPOSIZIONE DA ECLIPSE

Eclipse mette a disposizione dell’operatore una procedura per creare nuovi plug-in, per cui tramite un wizard si viene guidati nella creazione dello scheletro del progetto plug-in.

Le tipologie dei plugin sono diverse e per darne un’indicazione possiamo vedere i seguenti:

- Plugin per la semplice aggiunta di un nuovo menù nella tool bar.
- Plugin per la gestione di più finestre di editor.
- Plugin per la gestione di nuovi sottomenu e popup menù.

- Plugin per la gestione di properties relative ad un processo di elaborazione.
- Plugin per la gestione di finestre all'interno del Workbench.
- Plugin per la gestione di editor specializzati per xml.
- Plugin per la gestione di help.

Le classi che costituiscono la struttura del progetto plugin, possono variare leggermente in base alla tipologia del plugin stesso; in linea di massima in un normale plugin si hanno sempre classi base per l'attivazione e la definizione delle funzionalità aggiunte, in figura 1.13 e 1.14, possiamo vedere la struttura semplificata della classi Main (Activator.java) per l'attivazione del plugin e la classe che estende i thread, per la gestione delle funzionalità del plugin.

```
package de.vogella.osgi.firstbundle;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

import de.vogella.osgi.firstbundle.internal.MyThread;

public class Activator implements BundleActivator {
        private MyThread myThread;

        public void start(BundleContext context) throws Exception {
                System.out.println("Starting de.vogella.osgi.firstbundle");
                myThread = new MyThread();
                myThread.start();
        }


        public void stop(BundleContext context) throws Exception {
                System.out.println("Stopping de.vogella.osgi.firstbundle");
                myThread.stopThread();
                myThread.join();
        }

}
```

Figura 1.13 Activator.java, classe main per l'attivazione del plugin.

```
package de.vogella.osgi.firstbundle.internal;

public class MyThread extends Thread {
	private volatile boolean active = true;

	public void run() {
		while (active) {
			System.out.println("Hello OSGI console");
			try {
				Thread.sleep(5000);
			} catch (Exception e) {
				System.out.println("Thread interrupted " + e.getMessage());
			}
		}
	}

	public void stopThread() {
		active = false;
	}
}
```

Figura 1.14 Classe per la definizione delle funzionalità del plugin da gestire.

Nella classe “Activator.java” abbiamo i metodi per l’avvio e l’arresto del plugin, mentre nella classe che attiva il codice di estensione, sotto il controllo di un thread, abbiamo la “run” del plugin.

1.7 DEPLOY E DIPENDENZE

L’operazione di deploy non è altro che l’installazione delle risorse che definiscono il plug-in (manifes, file jar ed eventuali altre risorse), all’interno delle directory strutturali di Eclipse (in genere *Plugins*).

Nel momento in cui un plug-in viene chiamato per assolvere alla specifica funzione che gestisce, si attiva, mentre nel caso opposto si disattiva; tale meccanismo è implementabile attraverso l’estensione della classe “org.eclipse.core.runtime.Plugin” (indicata dal tag “class” all’interno del plugin.xml) che appunto definisce l’infrastruttura per l’attivazione/disattivazione dei metodi di una classe (rispettivamente “*startup*” e “*shutdown*”) ed include la definizione di attributi descritti nel rispettivo manifest.

Nel manifest, ad esempio di JUnit di Figura 1.6, possiamo notare che non è specificata nessuna classe per la gestione del plug-in, in questi casi la gestione dei plug-in avviene tramite una classe di default.

Nel modello di Eclipse, ogni plug-in può essere messo in collegamento con altri attraverso due tipologie di relazione:

- *Dependency:* sono indicati i prerequisiti di un plug-in, che devono essere gestiti dagli altri per implementare le estensioni.
- *Extension:* sono indicate le estensioni, per lo sviluppo delle funzionalità.

Questo tipo di relazioni vengono dichiarate all'interno del manifest tramite i tag "*requires*" e "*extension*", che definiscono quindi le dipendenze tra i vari plug-in (figura 1.15).

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="com.bolour.sample.eclipse.demo"
   name="Extension Processing Demo"
   version="1.0.0">
   <runtime>
      <library name="demo.jar"/>
   </runtime>
   <requires>
      <import plugin="org.eclipse.ui"/>
   </requires>
</plugin>
```

Listing 2.2. Specifying Plug-in Dependencies.

Figura 1.15 plugin.xml

In questo esempio, la classe plug-in "com.bolour.sample.eclipse.demo" è dichiarato essere dipendente dalla classe base di Eclipse UI plug-in "org.eclipse.ui".

1.8 INSTALLAZIONE DEI PLUG-IN

Sono due i modi di installazione dei plug-ins:

1) Installazione di un plug-in non gestito (unmanaged senza l'uso dell'updateManager.

In questo caso il plug-in viene copiato nella cartella dei plug-ins di Eclipse e viene rilevato in fase di startup dal Platform Runtime. L'update manager non è cosciente della sua presenza.

2) Installazione di un plug-in mediante updateManager definendo un "Feature".

In questo caso va definita una "Feature". L'update manager organizza le funzionalità di Eclipse mediante le features, la quale si definisce in modo analogo per quanto fatto per i

plu-ins. In questo caso si parla di feature manifest (feature.xml) in cui si specificano quali plug-ins fanno parte della feature assieme con i dati statici di informazione relativi alla feature stessa.

Riassumendo quanto visto sulla struttura della piattaforma di Eclipse, ed il concetto di plug-in, è evidente l'obiettivo di costruire uno strumento modulare, strutturalmente dinamico anche a tempo di esecuzione, facendo uso delle risorse del sistema solo quando richiesto dalle funzionalità che vengono impiegate.

Attraverso al meccanismo dei plug-in si hanno diversi vantaggi, tra cui facilità d'integrazione con gli extension-point, flessibilità e dinamicità grazie alla disponibilità a run-time del plug-in registry, la quale utilità si manifesta nella possibilità, primo di verificare se un aggiornamento o aggiunta di un nuovo plug-in possa causare problemi o conflitti, e secondo di migliorare le prestazioni grazie al caricamento selettivo dei singoli plug-in.

2. SODA Societies in Open and Distributed Agent Spaces

[6] L’ingegneria del software si propone lo studio degli agenti (AOSE – Agent Oriented Software Engineering) per definire modelli software che possano gestire sistemi di elevata complessità; da qui nascono i sistemi multi-agente (MAS – Multi Agent System), caratterizzati da due concetti base:

- L’ambiente in cui si trovano gli agenti.
- Le società che si vengono a creare dai gruppi di agenti che interagiscono.

Nel primo punto si evidenziano le caratteristiche dell’ambiente, le risorse presenti e le modalità con cui gli agenti interagiscoano con esse. Il secondo punto specifica la sintesi delle regole sociali che condizionano l’evoluzione del sistema multi-agente ed i ruoli assunti da ciascun agente all’interno del sistema-società.

SODA (Societies in Open and Distributed Agent spaces) rappresenta una metodologia agent-oriented per la gestione di sistemi complessi, tramite l’impiego di sistemi multi-agente basato sul meta-modello “Agenti & Artefatti”.
La caratteristica della programmazione ad agenti si può mettere in evidenza dal rapporto tra l’architettura multi-agente a quella object-oriented.
Nella object-oriented gli oggetti sono entità passive che rispondono solamente a stimoli esterni e l’ambiente non viene in alcun modo modellato in quanto tutto risulta essere un oggetto; mentre per l’architettura multi-agente gli agenti, le società, l’ambiente e le iterazioni sono entità di prima classe nella modellazione del sistema.

In figura 2.1 possiamo vedere schematizzato il concetto appena evidenziato.

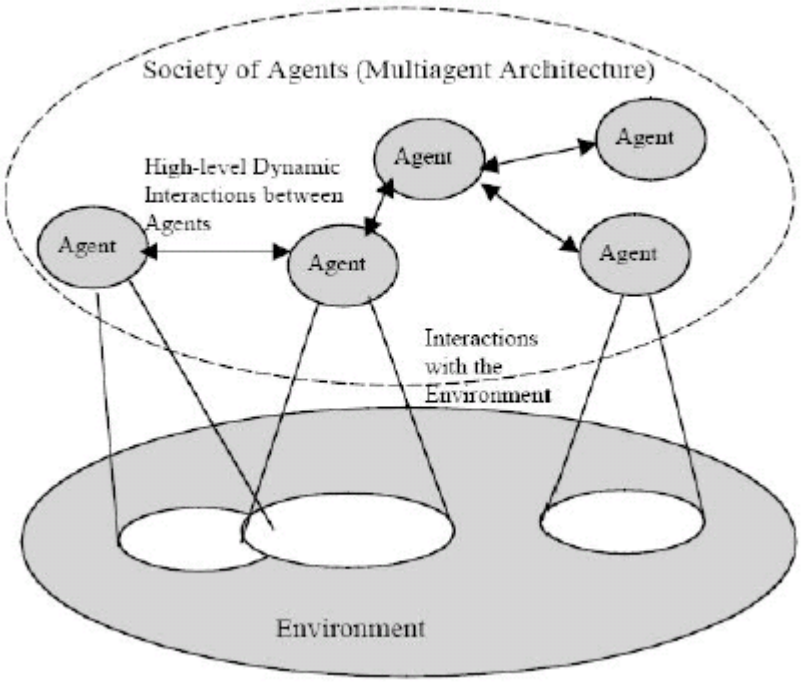


Figura 2.1 Architettura del sistema multi-agente

2.1 METODOLOGIA SODA

Rappresenta una metodologia per la definizione degli aspetti sociali di sistemi MAS, definendo l'aspetto "sociale di agenti" e "dell'ambiente"; quindi un ruolo centrale rivestono:

- *L'agente*: un'entità computazionale autonoma ed atomica, collocata in un ruolo preciso con aspetti sociali definiti, per il raggiungimento di obiettivi (goal-oriented).
- *L'artefatto*: tutte quelle entità che definiscono l'ambiente circostante, vale a dire le risorse, oggetti e strumenti che devono essere utilizzati dall'agente.

Gli *artefatti* sono quindi risorse a disposizione degli *agenti*, e per essere impiegati sono dotati di:

- Interfaccia: che contiene le operazioni disponibili, per l'utilizzo da parte dell'agente.
- Istruzioni operative: descrivono le azioni che l'agente deve eseguire per poter agire correttamente con l'artefatto.
- Descrizione funzionale: utile all'agente per aiutare nell'operazione di selezione, infatti da indicazioni di cosa si può ottenere dall'artefatto.

SODA quindi mette in evidenza il comportamento degli agenti e del ruolo rivestito nel sistema, tralasciando gli aspetti intra-agente; questo implica che la metodologia porta ad un modello specifico del sistema e la struttura degli agenti possa essere definita da una metodologia qualsiasi senza influire sull'architettura del sistema.

SODA opera tre diverse tipologie di astrazioni per descrivere le entità coinvolte:

- Per modellazione della parte attiva del sistema.
- Per la specifica della parte reattiva.
- Per la definizione delle interazioni e delle regole organizzative.

Il processo SODA è suddiviso in 2 fasi: *Analisi* e *Design* (Figura 2.2). Entrambe sono composte a loro volta da due sotto-fasi (chiamate step) ognuna, infatti l'Analisi si divide in *Analisi dei Requisiti* e *Analisi*, mentre il sotto-processo di Design comprende *Design Architetturale* e *Design di Dettaglio* [6].

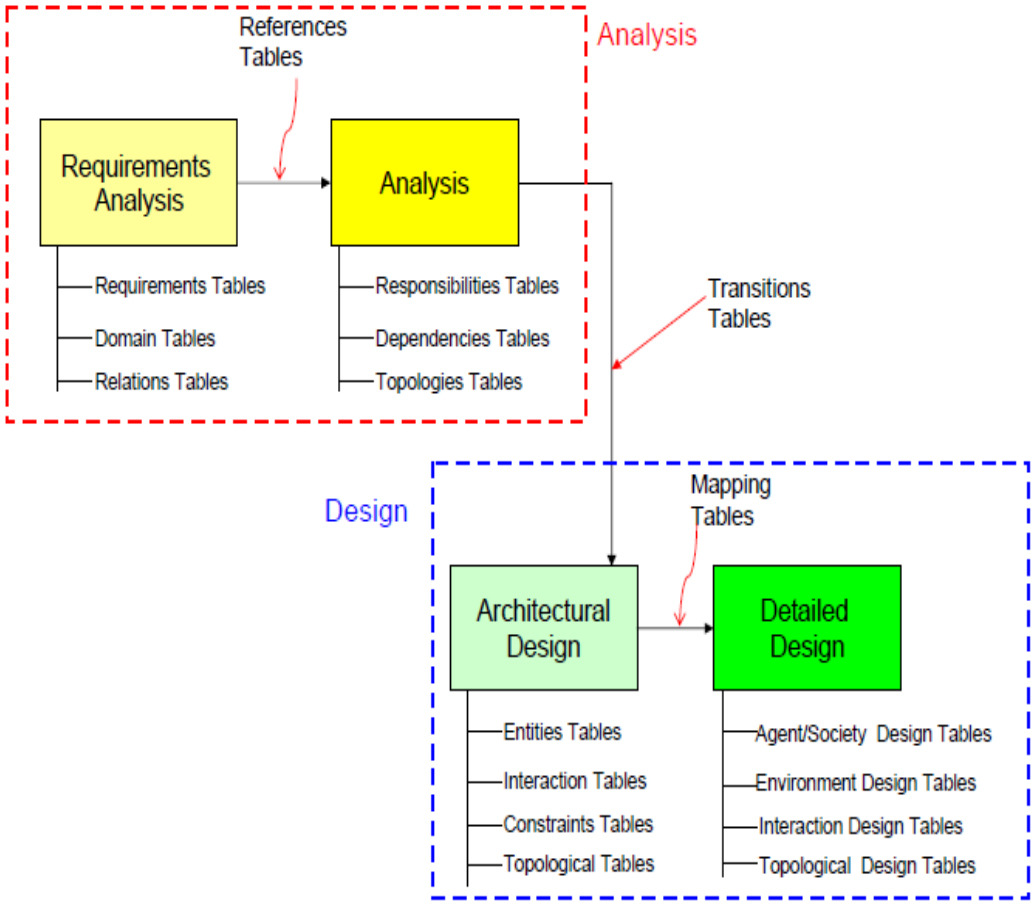


Figura 2.2 Fasi del processo SODA

L'adozione del meta-modello A&A consente di modellare sia le entità sia le risorse del sistema come agenti, attraverso astrazioni concettuali ben definite. Gli agenti definiscono attività autonome, mentre gli artefatti incapsulano operazioni di mediazione tra gli agenti ed il mondo esterno al sistema o modellare l'ambiente che supporta le attività degli agenti stessi.

Gli agenti possono utilizzare gli artefatti in diversi modi, che condizionano la modalità di selezione e controllo, in base alle necessità rendere più o meno dinamici i criteri di scelta, avremo quindi cinque modi d'uso possibili:

- *Uso inconsapevole:* gli agenti ed i loro designer non conoscono a priori gli artefatti, quindi questi ultimi sono usati senza riferimenti espliciti alle loro funzionalità.
- *Uso programmato:* il progettista dell'agente stabilisce a priori l'utilizzo dell'artefatto, programmandolo nel codice dell'agente; le caratteristiche dell'artefatto devono quindi essere note al designer, senza necessità di esplorarle da parte dell'agente.

- *Uso cognitivo:* il progettista programma la selezione di un particolare agente, ma il modo in cui deve essere utilizzato e le sue caratteristiche devono essere ottenute dinamicamente dall'agente; questo implica la necessità di formalizzare la struttura e le funzioni dell'artefatto.
- *Selezioni ed uso cognitivi:* il progettista delega all'agente sia la scelta degli artefatti da usare, sia l'esplorazione delle funzioni esposte e del loro modo di utilizzarle.
- *Costruzione e manipolazione:* gli agenti sono in grado di capire il funzionamento degli artefatti e soprattutto di progettarne o cambiarne il comportamento in base alle proprie necessità. Questo può aumentare la complessità degli agenti, ma è possibile semplificarne la struttura, infatti il designer può definire uno schema di comportamento da eseguire.

2.2 GLI ARTEFATTI

Gli artefatti possiamo vederli come “wrappers” [6],[7] che inglobano ed astraggono le risorse presenti nell’ambiente, consentendo di modellare la parte reattiva del sistema. Gli artefatti si manifestano agli agenti esplicitando i propri servizi attraverso interfacce d’uso (operazioni fornite), istruzioni operative (descrizione della procedura d’iterazione) e descrizione formale (descrizione di cosa si può ottenere dall’artefatto; serve nella selezione da parte dell’agente). In figura 2.3 viene riportata la struttura generale dell’artefatto.

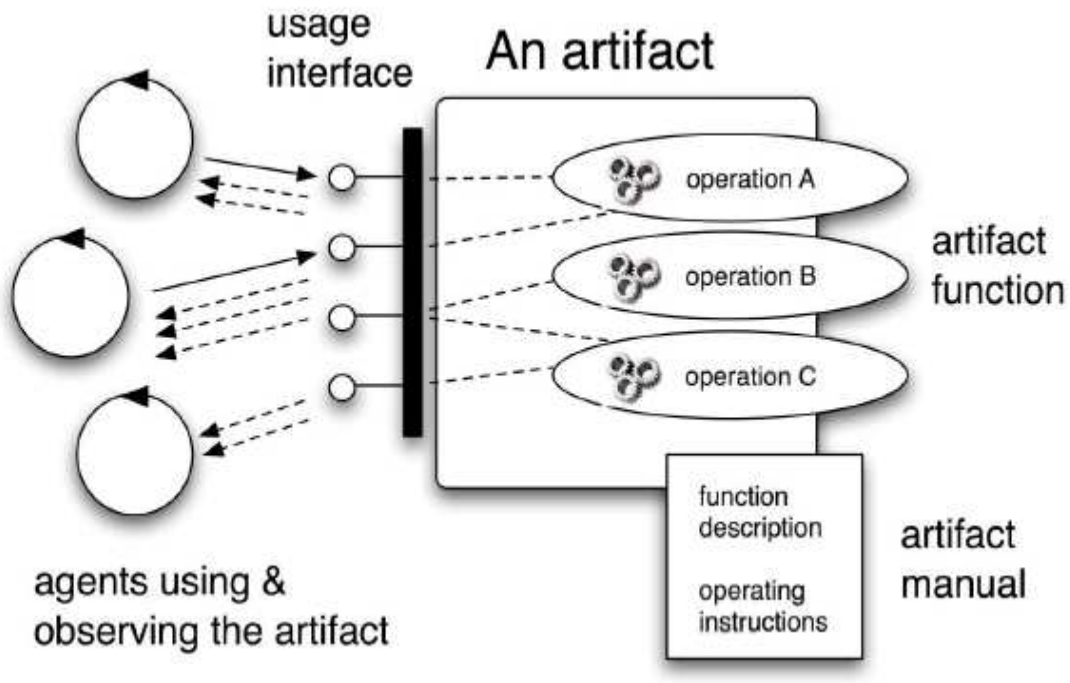


Figura 2.3 Modello di artefatto: interfaccia, istruzioni operative e descrizione

In base al ruolo di mediazione svolto, possiamo distinguere tre diverse tipologie:

- *Artefatti individuali:* sono utilizzati da un unico agente, mediano con un solo agente e l’ambiente. Questi artefatti non sono influenzati dagli altri agenti con cui non interagiscono, ma possono comunicare con altri artefatti (linkability).
- *Artefatti sociali:* sono utilizzati da più agenti, e di conseguenza forniscono un servizio finalizzato a raggiungere un obiettivo sociale del sistema.
- *Artefatti risorsa:* mediano tra una risorsa esterna ed il sistema.

Altre proprietà importanti degli artefatti sono:

- *Ispezionalità*.
- *Controllabilità* (estensione della precedente): monitoraggio del funzionamento a run-time.
- *Malleabilità:* modificabilità a run-time.
- *Predicibilità* del comportamento (in base alle istruzioni operative).
- *Formalizzazione del comportamento* con un modello semantico.
- *Linkability:* link incrementali per consentire di mettere in relazione più artefatti per condividere le funzionalità.
- *Distribuzione.*

Gli artefatti condizionano l'aspetto sociale degli agenti all'interno del sistema, strutturandolo attraverso i permessi, politiche e protocolli che possono essere rappresentati come entità di prima classe oppure incapsulati negli artefatti.

Sia gli agenti che gli artefatti sono contenuti all'interno di un contenitore che prende il nome di "workspace".

2.3 WORKSPACE

Rappresenta il contenitore aperto e dinamico, attraverso cui si definisce la topologia dell'ambiente del sistema multi-agente.
All'interno del workspace agenti ed artefatti possono aumentare o diminuire a seconda delle esigenze.

2.4 LAYERING

Uno dei concetti importanti e fondamentali della metodologia SODA è quello di progettare e sviluppare un sistema attraverso vari livelli di astrazione indipendenti ma correlati, il tutto prende il nome di "layering". Le entità che siano agenti o artefatti, collocate ad un dato livello di astrazione, sono l'aggregazione di altre entità appartenenti ad un livello inferiore e di maggior dettaglio. E' possibile passare da un livello ad un altro attraverso l'operazione di *in-zoom* (maggior dettaglio. Figura 2.4) oppure *out-zoom* (minore dettaglio. Figura 2.5). Altra operazione è quella di *proiezione* (figura 2.6) che porta entità da un livello ad un altro oppure le colloca direttamente ad un dato livello, senza eseguirne alcun tipo di scomposizione/aggregazione.

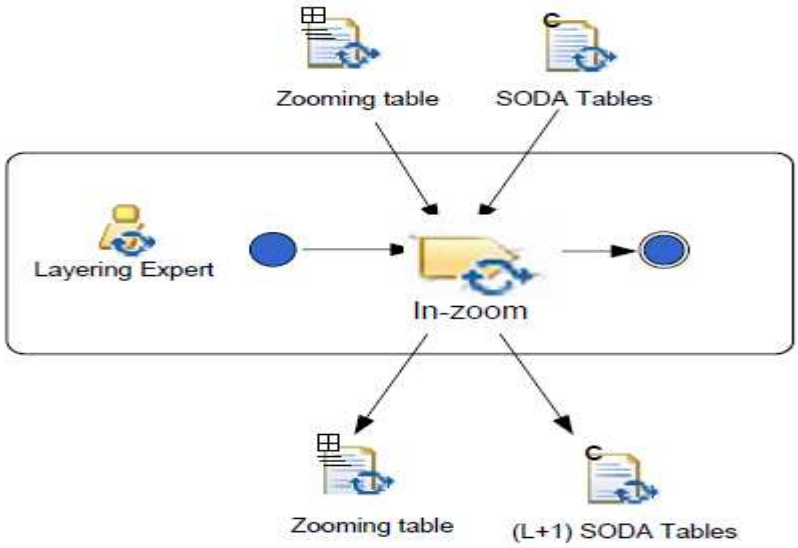


Figura 2.4 In-Zoom table

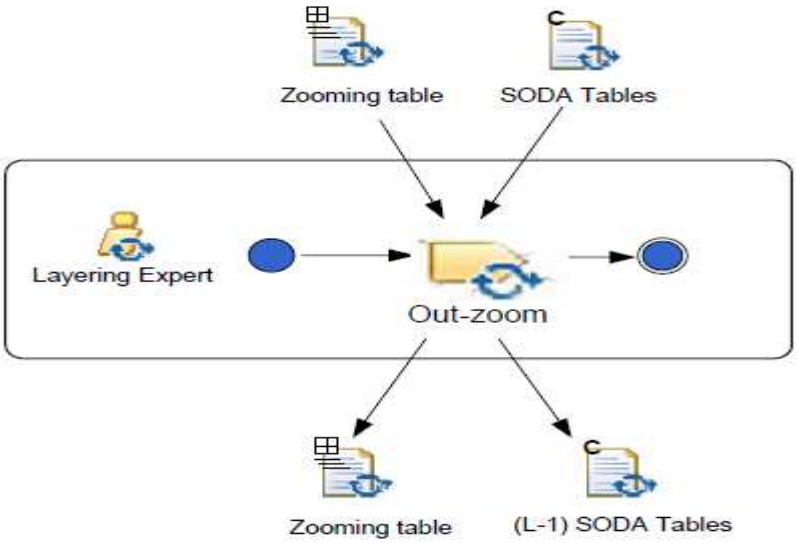


Figura 2.5 Out-Zooming table

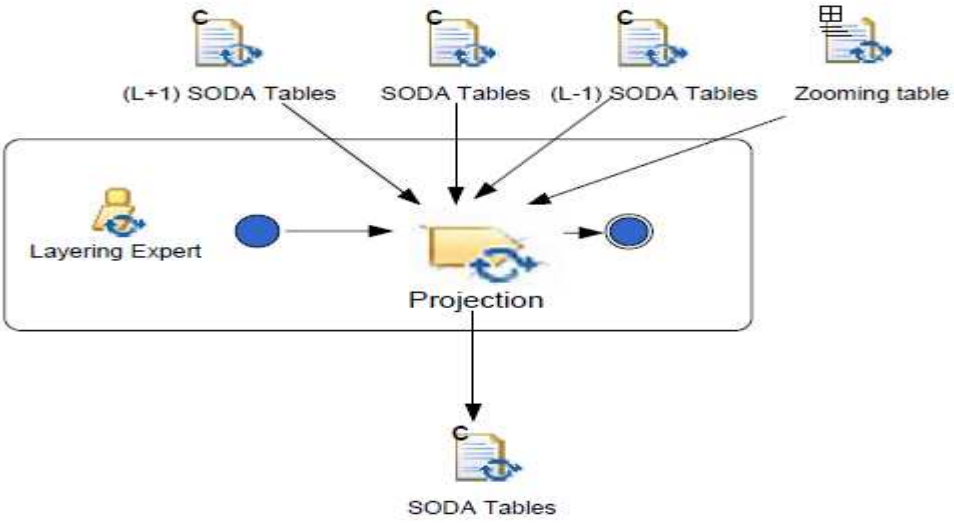


Figura 2.6 Projection table

In figura 2.7 viene rappresentato il modello di layering, in cui si vede come ottenere lo sviluppo dei livelli, semplicemente ripetendo ciclicamente le medesime operazioni di zoom. Sempre in figura viene rappresentata la struttura concettuale delle operazioni descritte.

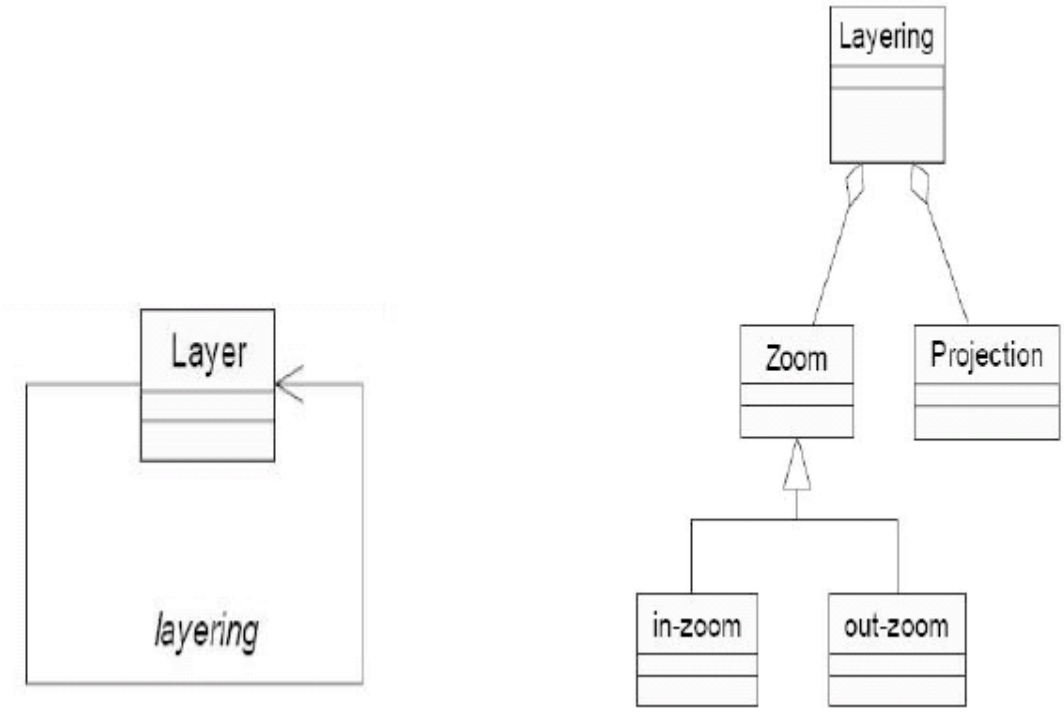


Figura 2.7 Modello di layering.

Quando si applica la metodologia SODA, si individua un livello di partenza, che rappresenta i *core* (C); tale livello deve risultare completo, vale a dire contenere tutte le entità richieste. Tutti gli altri livelli si distinguono dal core e vengono denominati C+1,

C+2, … C-1…(Figura 2.8), questi livelli possono anche non riportare tutte le entità e quindi essere non completi, ma evidenziare solo le entità che vengono raffinate in maggior/minor dettaglio. Sempre in figura si possono notare entità di uno stesso livello, che stanno in relazione, ed entità che passando a livelli maggiormente dettagliati danno luogo ad una separazione un più entità specifiche.

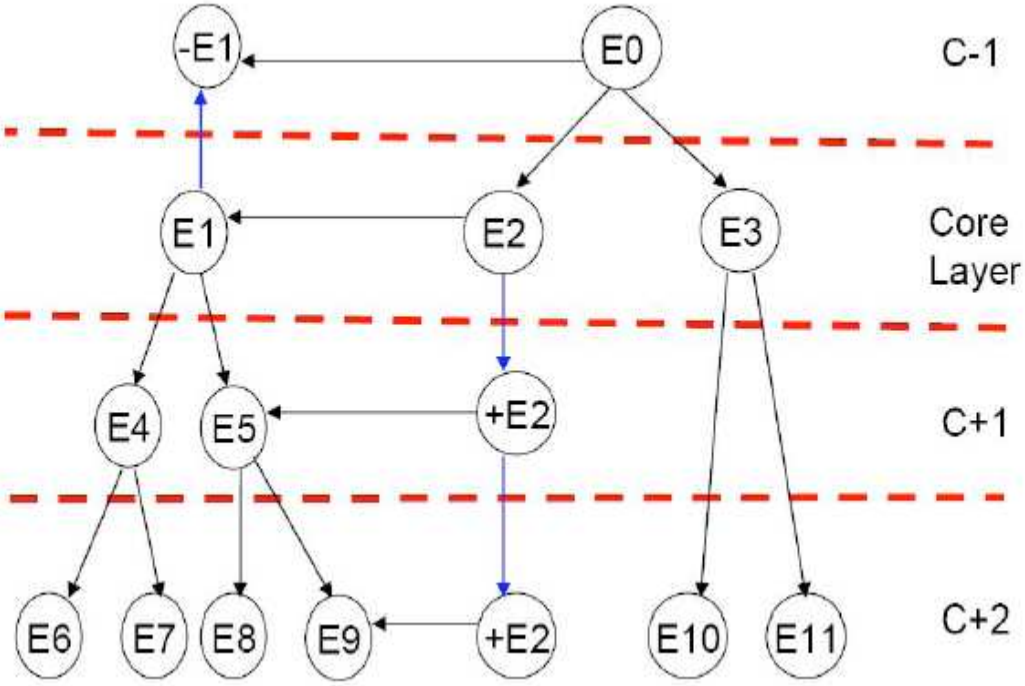


Figura 2.8 Layers di SODA

Dalla figura vediamo ad esempio entità che riportano un segno + ed altre il -, esse sono le entità che vengono proiettate attraverso i livelli. Il layering viene rappresentato mediante una "*Zooming-table*" (C) Zt che indica C il livello, Zt la sigla della tabella ed il pedice 't' indica che si tratta di una tabella.

In figura 2.9 è riportato il flusso del processo di layering (la parte atomica), che determina lo sviluppo dei livelli.

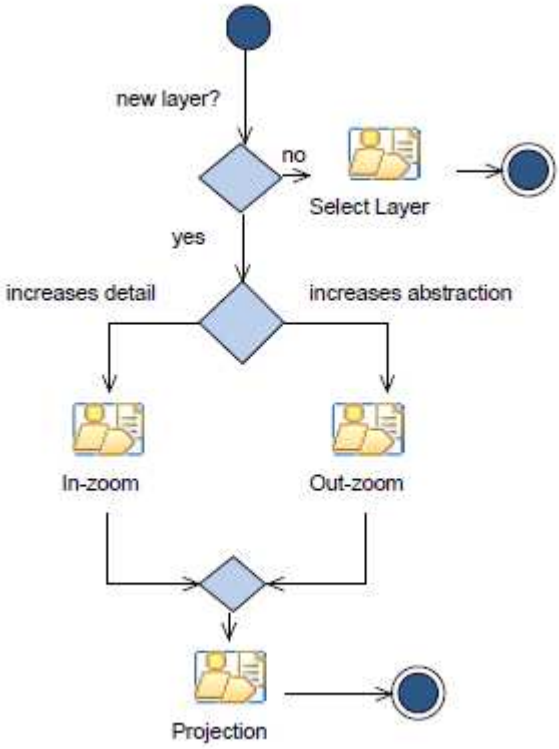


Figura 2.9 il processo di Layering

Nel dettaglio:

- *SELECT LAYER:* gli input sono rappresentati dallo Zooming table e dalle tabelle SODA relative alla fase del processo in cui si applica il layering. In output vengono prodotte altre tabelle del processo SODA.
- *IN-ZOOM:* ha come documenti iniziali la Zooming table e le tabelle SODA a livello L; produce a sua volta una Zooming table e le tabelle SODA a livello L+1.
- *OUT-ZOOM:* come l'attività precedente solo vengono create le tabelle di livello L-1.

Mentre in figura 2.10 vengono schematizzate le attività che compongono il layering:

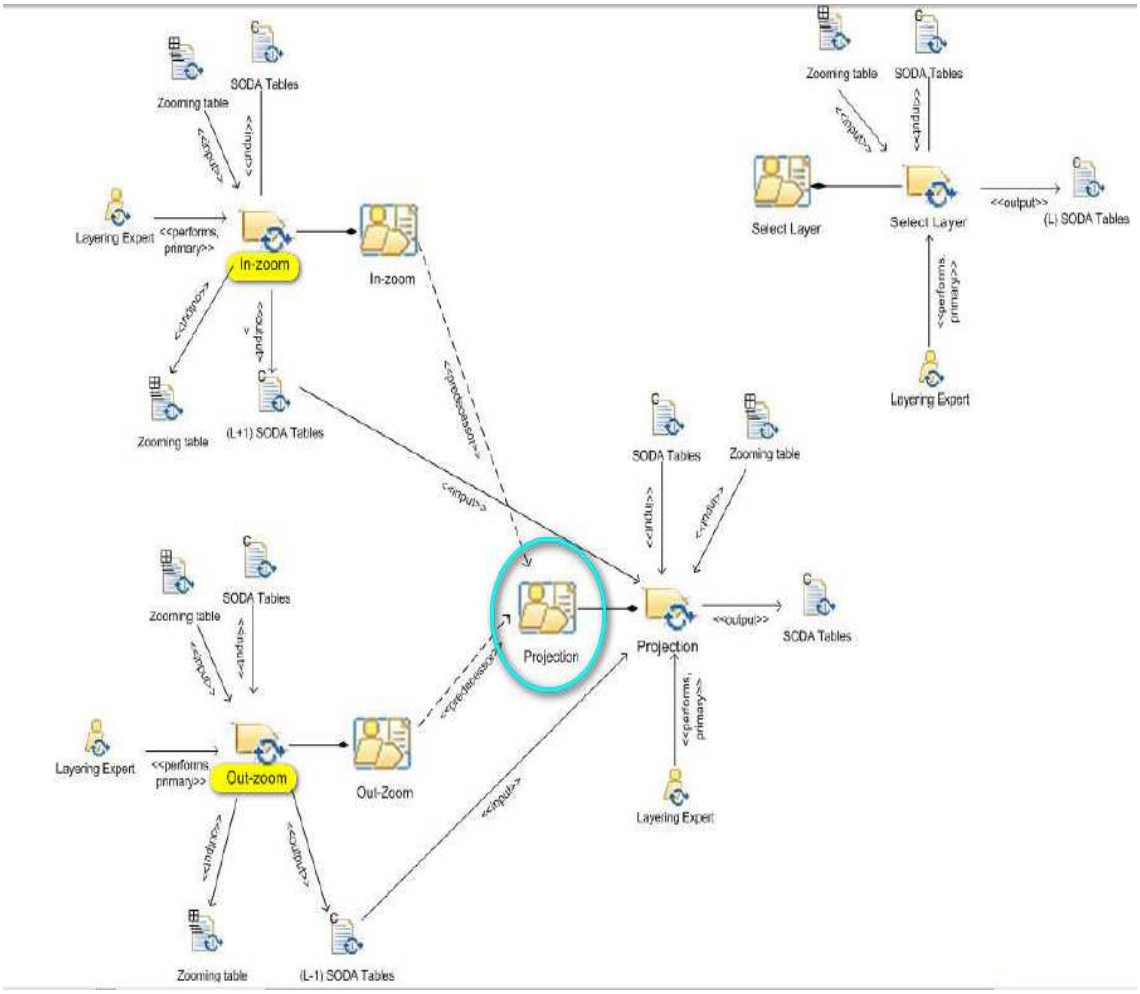


Figura 2.10 Flusso delle attività del Layering.

Dove per:

- *PROIEZIONE:* essendo sempre conseguente ad una operazione di zoom, ha come input le tabelle SODA di livello L-1, L, L+1, oltre che la Zooming table. I documenti prodotti sono tabelle del processo SODA a livello L.

Importante sottolineare che la transizione tra i livelli può portare alla definizione di nuove entità oppure al raffinamento delle esistenti.

Il principio di layering può essere applicato durante le fasi di analisi dei requisiti, analisi e design architetturale; concetti che vedremo successivamente.

2.5 IL METAMODELLO DELLE ASTRAZIONI SODA

Il meta-modello SODA si compone di quattro fasi, ognuna delle quali individua precise entità, che vengono passate da una fase all'altra con una corrispondenza diretta (linee nere) oppure con traduzione delle entità (linee blu). In figura 2.11, sono indicate le quattro fasi SODA, con l'indicazione dall'applicabilità dell'operazione di layering o meno.

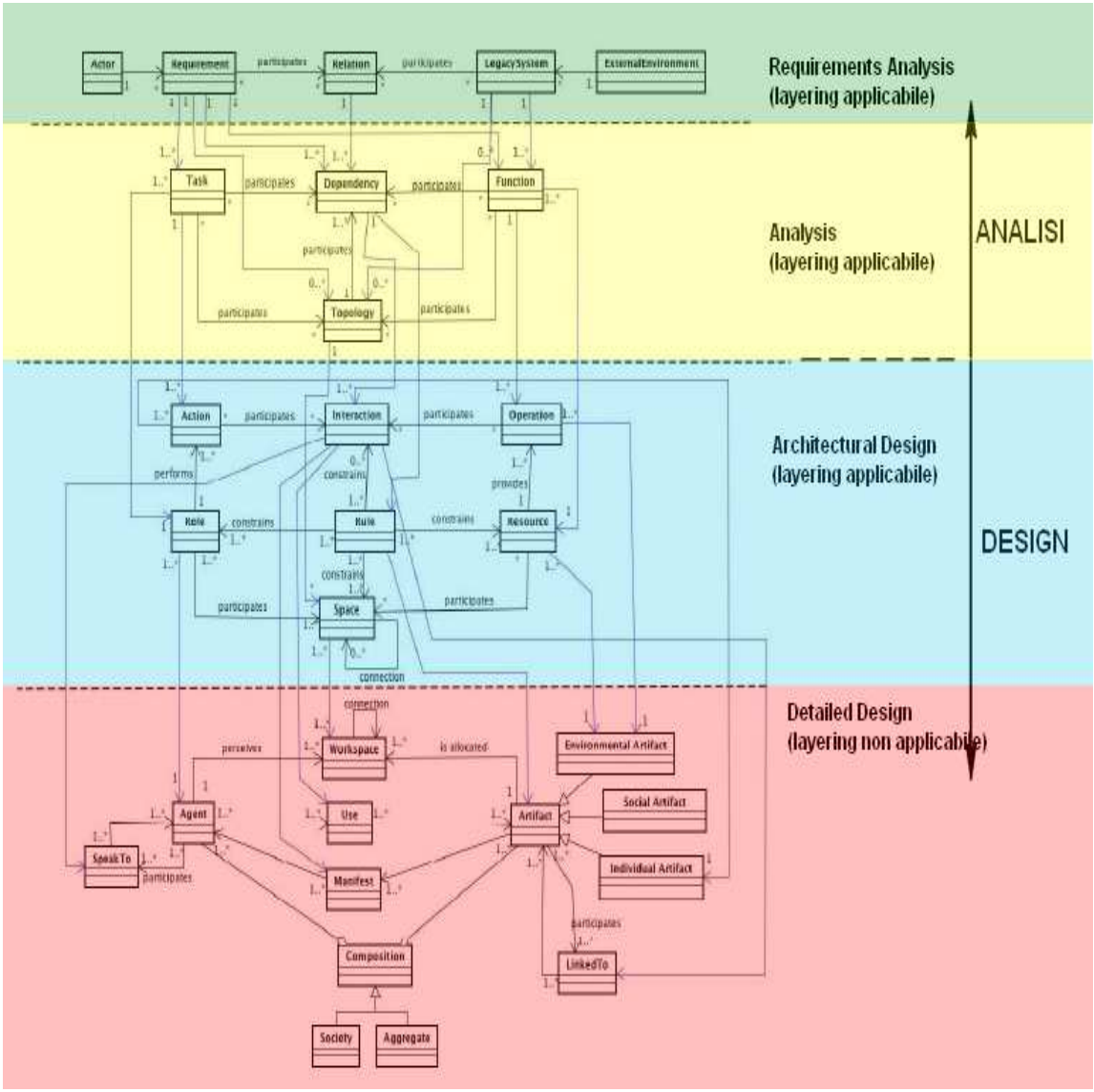


Figura 2.11 Meta-modello delle astrazioni SODA.

Diamo ora una descrizione delle fasi:

ANALISI DEI REQUISITI (Requirements Analysis): Vengono utilizzate entità astratte per modellare i requisiti del sistema. Vengono messi in evidenza i rapporti tra i requisiti ed i sistemi legacy.

- *REQUIREMENT* e *ACTOR:* usate rispettivamente per rappresentare i requisiti del cliente fornendo una descrizione funzionale, non funzionale o di dominio dei servizi del sistema, e le sorgenti del requisito ovvero gli utenti finali del sistema .
- *EXTERNALENVIRONMENT:* rappresenta l'astrazione usata per definire l'ambiente in cui è immerso il sistema.
- *LAGACYSYSTEM:* enumera i contesti di altri sistemi presenti nell'ambiente.
- *RELATION:* modella il legame tra i requisiti ed i sistemi legacy.

ANALISI: in questa fase si concretizzano le astrazioni fatte al passo precedente, definendo i *task* e le *function* (attività che supportano i task). Dalle relazioni della fase precedente, derivano:

- *DEPENDENCY:* vengono indicate le dipendenze tra le entità.
- *TOPOLOGY:* per modellare l'ambiente, definendo diverse topologie, spesso derivate dalle funzioni, ma che possono anche riguardare il raggiungimento di un task.

DESIGN ARCHITETTURALE: vengono introdotte le entità:

- *ROLE:* contiene la responsabilità del completamento dei task.
- *RESOURCE:* forniscono le funzioni necessarie.

Le *role* devono essere in grado di compiere azioni (*Action*) e le *resource* di eseguire operazioni mettendo a disposizione una o più funzioni.

- *ITERACTION:* mappano le dipendenze della fase precedente e rappresentano i collegamenti tra ruoli, risorse, ruoli e risorse.
- *RULE:* che abilitano e regolamentano il comportamento delle entità.
- *SPACE:* entità che consentono di astrarre e modellare la struttura dell'ambiente in cui si trova il sistema.

DESIGN DI DETTAGLIO: in questa fase le entità attive sono rappresentate da:

- *AGENT:* entità autonome in grado di svolgere diversi ruoli.
- *AGENT SOCIETIES:* insieme di agenti ed artefatti che interagiscono adottando un comportamento globale di tipo autonomo e proattivo.

mentre quelle passive vengono tradotte in:

- *ARTIFACT:* rendono più concrete le corrispondenti astrazioni delle risorse individuate nel passo precedente.
- *AGGREGATE:* hanno un comportamento complessivo funzionale e reattivo.

Le possibili iterazioni tra entità vengono definite in questa parte del processo tramite i concetti di:

- *Use:* indica l'uso di un artefatto da parte di un agente.
- *Manifest:* utilizzata dagli artefatti per segnalare la propria presenza agli agenti.
- *SpeakTo:* identifica una iterazione tra due agenti.
- *LikedTo:* caratterizza un legame tra due artefatti.

2.6 ZOOMING TABLE E GLI ELEMENTI DEL METAMODELLO SODA

La zooming table è il risultato prodotto dalla procedura di layering, strutturalmente tale documento ha la struttura di figura 2.12, in cui si vedono riportati i livelli e le entità per ciascuno di essi.

Layer L	Layer L+1
out-zoomed entity	in-zoomed entities

$(L)Z_t$

Figura 2.12 zooming table structure

In figura 2.13, vengono evidenziate la relazione tra zooming table e gli elementi del meta modello SODA.

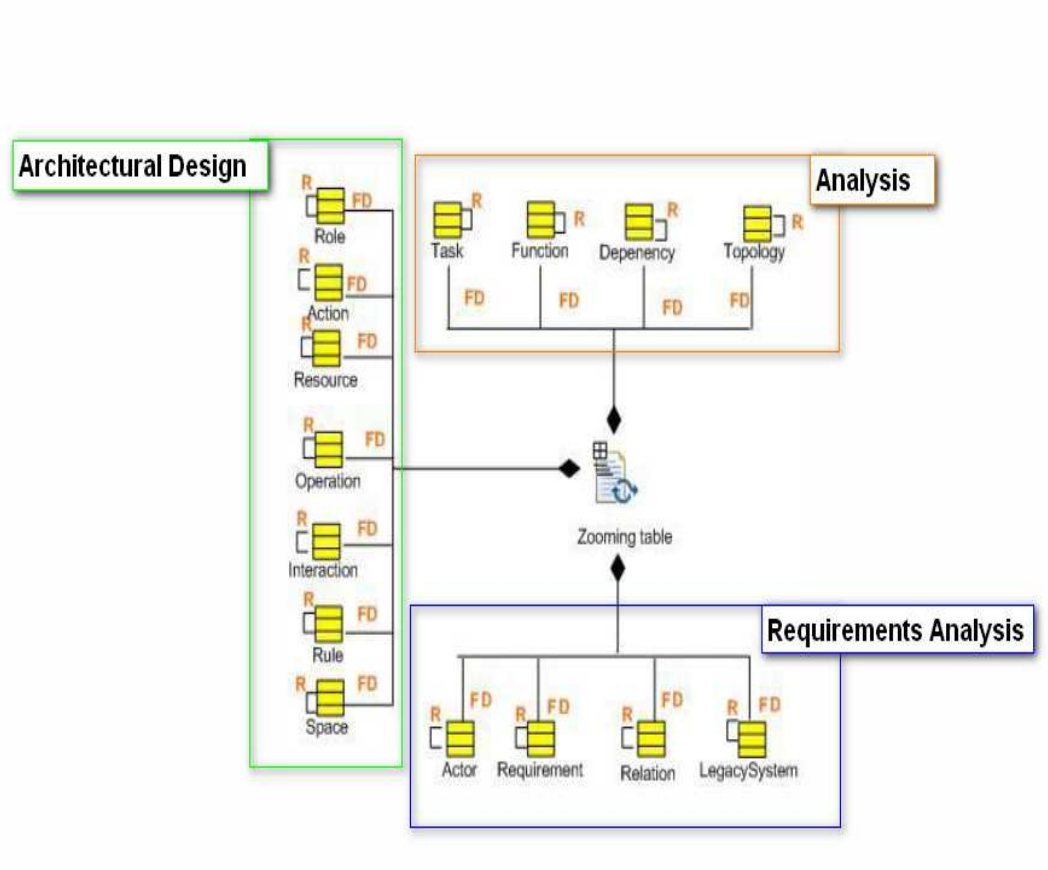


Figura 2.13 Relazioni tra zooming table ed elementi del meta modello

A seconda dell'etichetta ogni elemento può essere:

- (D) DEFINITO: l'elemento è introdotto per la prima volta (viene istanziato).
- (F) RIFINITO: viene aggiunta una caratteristica, che può essere ad esempio un nuovo attrivuto.
- (R) RELAZIONATO: un oggetto già esistente viene posto in relazione ad un altro, viene di conseguenza istanziata nella zooming table una nuova relazione.
- (Q) QUOTATO: un elemento già definito viene riportato nella zoomin table per completare la struttura.
- (RQ) RELAZIONE QUOTATA: una relazione definita in un altro punto del processo viene riutilizzata.

2.7 IL PROCESSO SODA

Dalle fasi che definiscono il meta modello SODA, deriviamo il processo SODA, come indicato in figura 2.14:

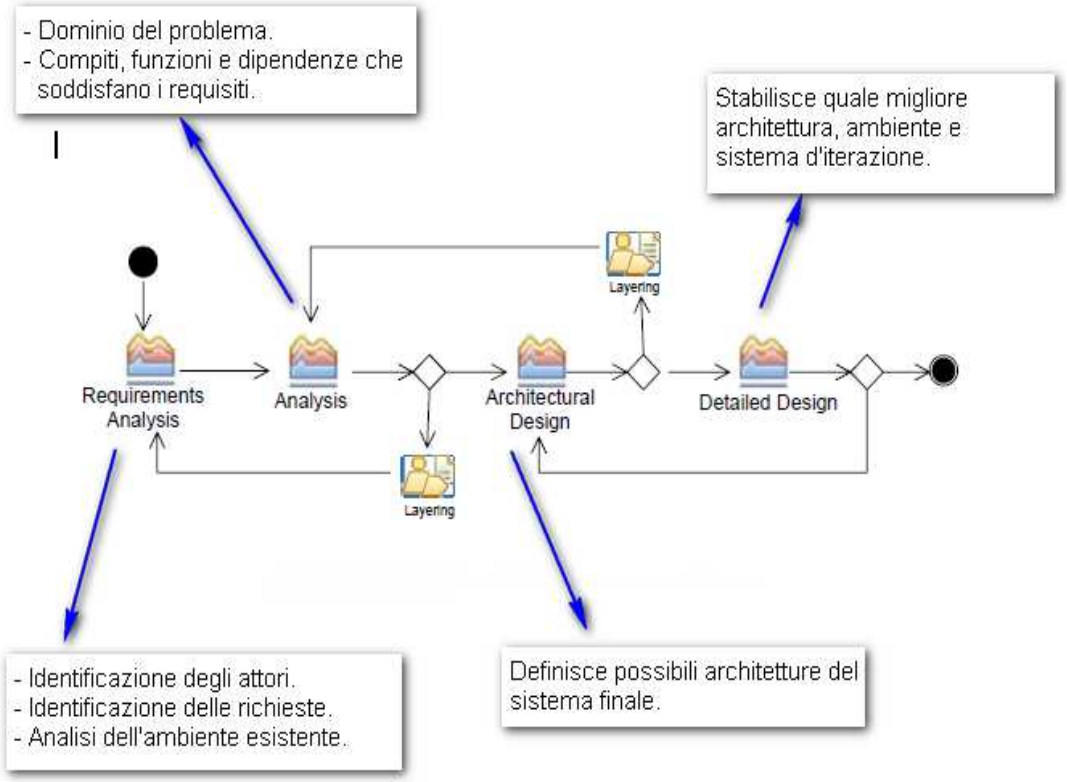


Figura 2.14 Ciclo di vita del processo SODA

Ogni passo produce un insieme di tabelle relazionali, ognuna della quali descrive uno specifico elemento del meta modello e le relazioni con gli altri elementi coinvolti.

2.7.1 ANALISI DEI REQUISITI

Il processo è definito in figura 2.15:

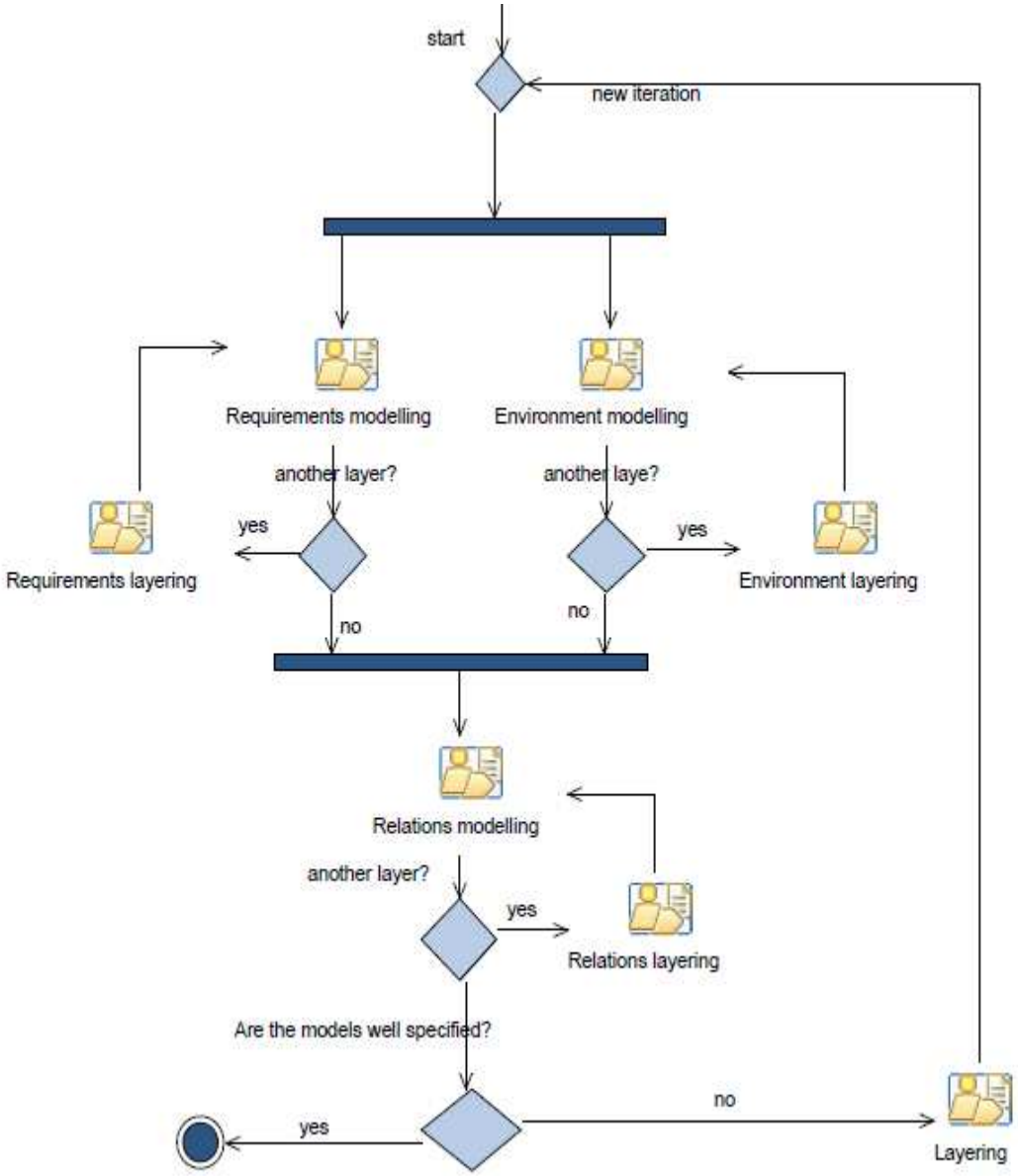


Figura 2.15 Processo della fase di analisi dei requisiti

Le prime attività che vengono svolte sono quelle della modellazione dei requisiti e dell'ambiente, che possono essere più volte reiterate utilizzando il layering.

Segue successivamente la modellazione delle relazioni che intercorrono tra le entità individuate nelle attività precedenti; anche questa sotto-fase può essere reiterata tramite il layering.

Il modello termina, verificando se il livello di astrazione è quello desiderato e nel caso ripete tutto il processo dall'inizio applicando il layering.

Analizzando la *modellazione dei requisiti* possiamo dire che sia suddivisa in due parti:

- La descrizione degli attori.
- La descrizione dei requisiti.

Entrambe prevedono come input il documento di specifica dei requisiti e la Zooming table.
In output si hanno i documenti relativi alla descrizione degli attori, nello specifico la *Actor table,* la quale diventa input per la parte successiva, mentre da quella dei requisiti si ottengono la Requirement table e la Actor-Requirement table, che mette in evidenza le figure legate ad ogni singolo requisito del sistema.

Dalla *modellazione dell'ambiente* si hanno tutti i sistemi legacy presenti appunto nell'ambiente in cui opera il sistema. In ingresso abbiamo sempre la Zooming table, mentre in uscita vengono prodotte la Legacy-system table che contiene la descrizione dei sistemi legacy che sono all'interno del sistema, e la External-environment-legacy table che contiene la descrizione dei sistemi legacy all'esterno del sistema.

Dalla *modellazione delle relazioni* vengono formulate le relazioni intercorrenti tra le entità individuate in precedenza. Come output si ottengono la Relation table, la Requirement-relation table e la LegacySystem-relation table. In figura 2.16 si evidenziano gli output prodotti dal flusso di analisi dei requisiti.

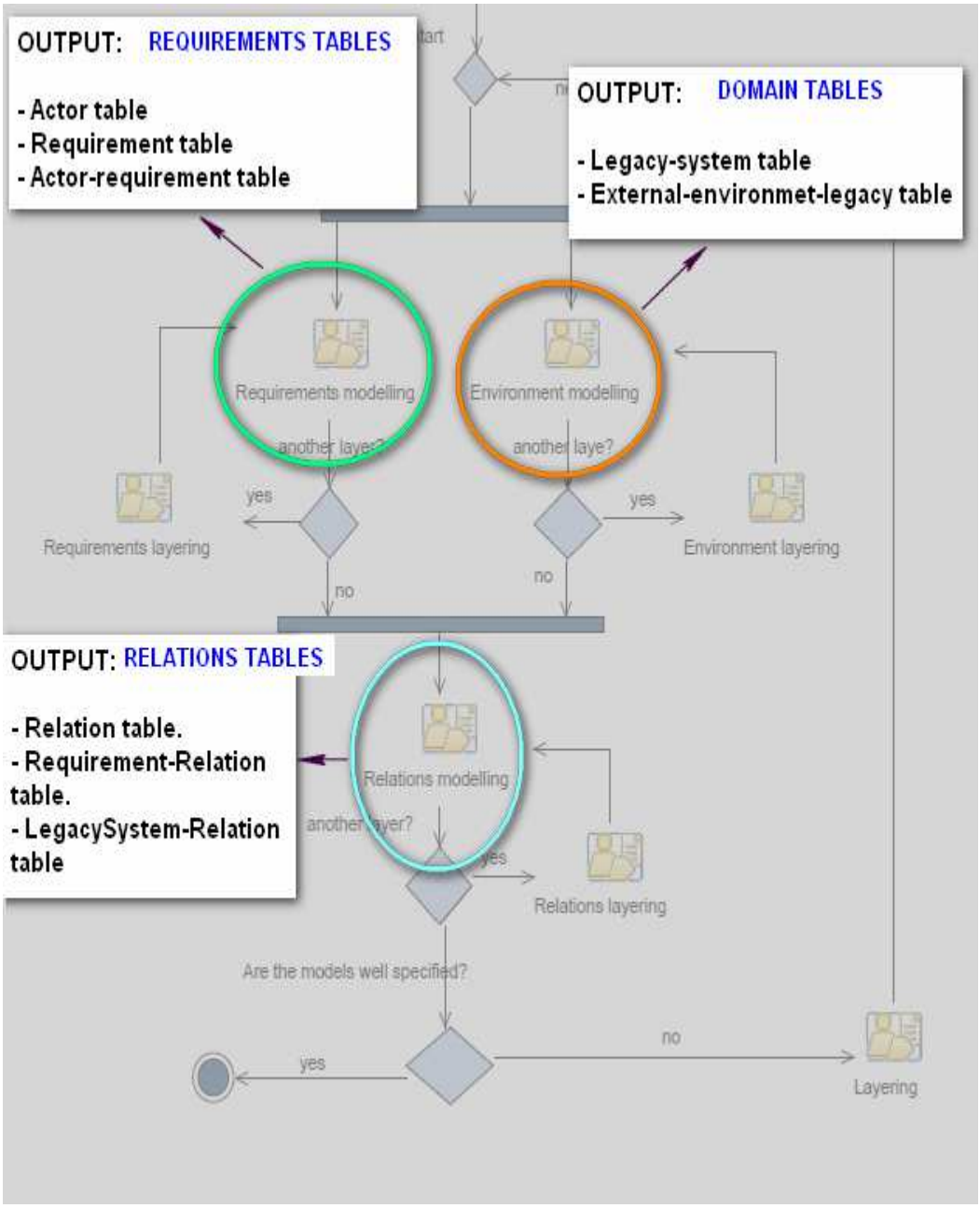


Figura 2.16 Output del processo di analisi dei requisiti

Per dare una visione d'insieme del flusso delle attività che compongono l'analisi dei requisiti con i relativi documenti di input e output, vediamo quanto riportato in figura 2.17.

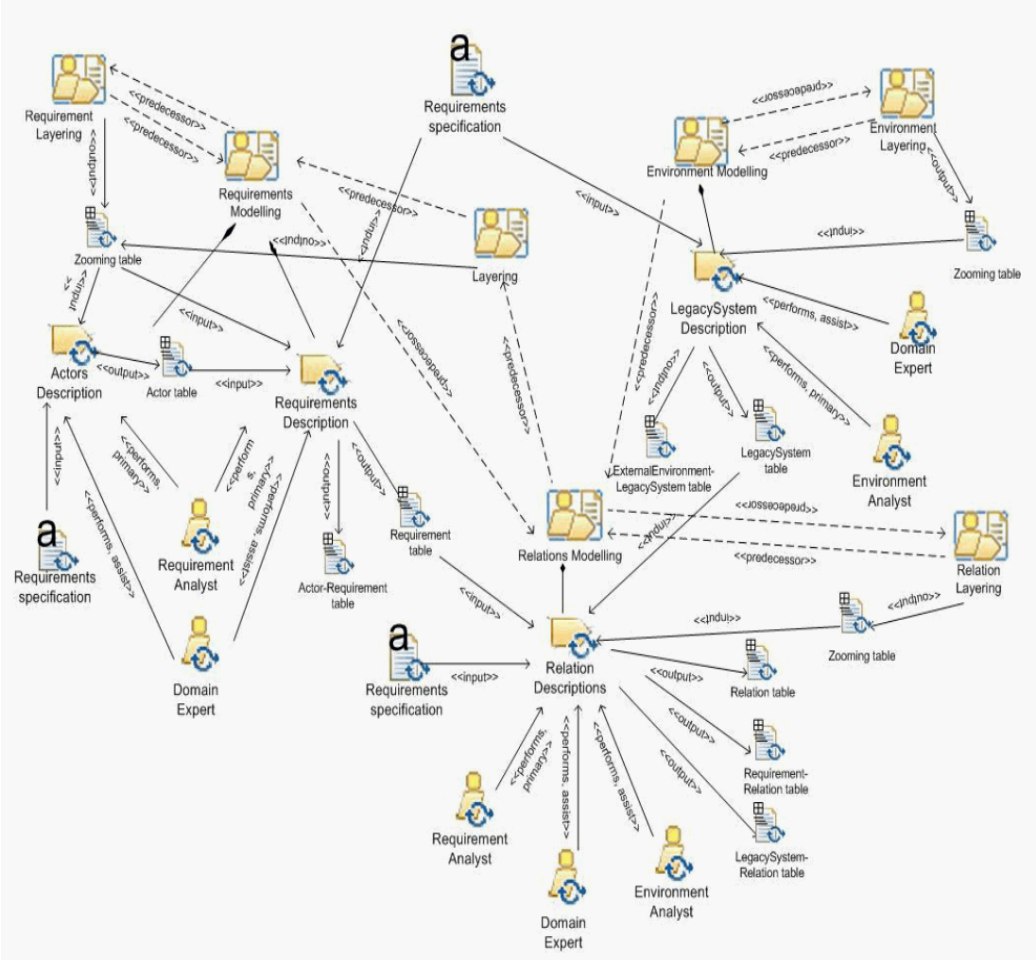


Figura 2.17 Flusso delle attività nell'analisi dei requisiti

Per prima cosa si ha che il documento di specifica dei requisiti è in input comune a tutte le attività.

Importante è notare la sequenza temporale in cui si ha prima la modellazione d'ambiente e dei requisiti, e successivamente la modellazione delle relazioni.

La Zooming table usata per la modellazione dei requisiti, può derivare dal Requirement layering o dal layering successivo alla modellazione delle relazioni.

In conclusione *dall'analisi dei requisiti* si ottengono tre categorie di tabelle (come evidenziato anche in figura 2.16):

REQUIREMENT TABLES: contiene le tabelle che definiscono le entità astratte legate al concetto di requisito:

- Actor table ((L)Ac_t) descrive ogni singolo attore del sistema;
- Requirement table ((L)Re_t) specifica i singoli requisiti;
- Actor-Requirement table ((L)AR_t) per ogni attore elenca i requisiti a cui è legato.

Actor	Description
Actor name	Actor description

(L)Ac_t

Requirement	Description
Requirement name	Req. description

(L)Re_t

Actor	Requirement
Actor name	Req. names

(L)AR_t

DOMAIN TABLES: contiene le tabelle che riguardano la definizione delle entità riguardanti l'ambiente esterno, quello in cui il sistema progettato è immerso:

- External-Environment Legacy System table ((L)EELS$_t$), che esplicita le associazioni tra sistemi legacy e ambiente esterno;
- Legacy system table ((L)LS$_t$) fornisce la descrizione dei singoli sistemi Legacy coinvolti.

External-Environment	Legacy System
Ext-env. name	Legacy-System names

(L)EELS$_t$

Legacy System	Description
Legacy-System name	Legacy-system description

(L)LS$_t$

RELATIONS TEBLES: contiene le tabelle che riportano i collegamenti tra le entità astratte:

- Relation table ((L)Rel_t) riporta le relazioni tra le singole entità astratte;
- Requirement-Relation table ((L)RR_t) elenca tutte le relazioni in cui è coinvolto ogni singolo requisito.
- LegacySystem-Relation table ((L)LSR_t) fornisce l'elenco delle relazioni in cui compare ogni sistema legacy.

Relation	Description
Relation name	relation description

(L)Rel_t

Requirement	Relation
Req. name	Relation names

(L)RR_t

Legacy System	Relation
Legacy-System name	Relation names

(L)LSR_t

2.7.2 RELAZIONI TRA LE TABELLE PRODOTTE DALL'ANALISI DEI REQUISITI E GLI ELEMENTI DEL META-MODELLO

Prendendo in esame la figura 2.18:

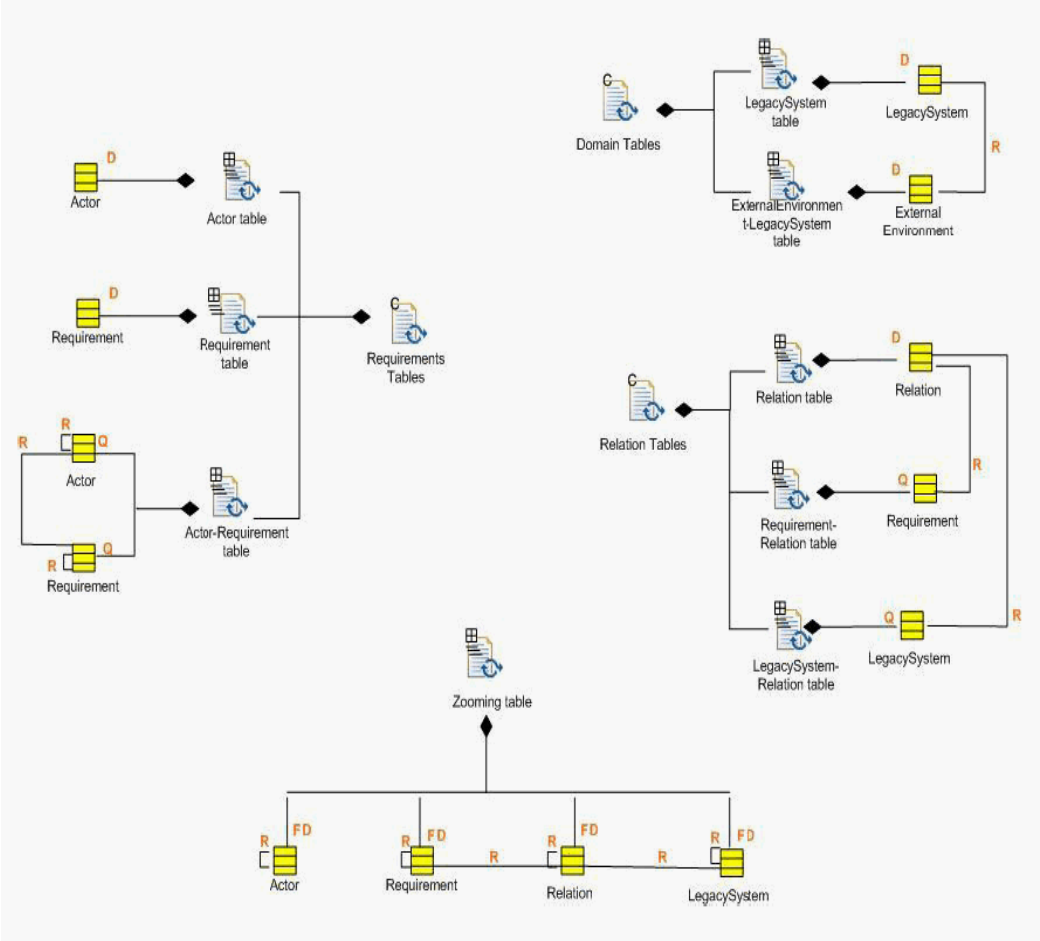


Figura 2.18 Relazioni tra work products dell'Analisi dei Requisiti ed elementi del meta-modello

Per quanto riguarda la Zooming table ha associazioni di definizione o rifinitura con gli elementi Actor, Requirement, Relation, LegacySystem; inoltre Relation è in relazione con Requirement e LegacySystem.

Infine ognuno degli elementi può essere messo in relazione con entità dello stesso tipo (legame di tipo R riflessivo).

Per le *Requiremets tables* si verifica che gli elementi Actor e Requirement sono legati rispettivamente alla Actor table ed alla Requirement table da vincoli di tipo D, quindi possono essere istanziate nuove entità di questi elementi.

La Actor-Requirement table invece lega Actor e Requirement, per cui riutilizza entità già definite; Actor e Requirement possono essere in questo caso in relazione tra loro ed anche con entità dello stesso tipo.

Per le *Domain Tables* si hanno legami di definizione con elementi LegacySystem ed External-Environment, rispettivamente per LegacySystem table ed ExternalEnvironmet-LegacySystem table; le entità appartenenti ai due tipi possono poi essere messe in relazione.

Considerando le associazioni delle Relation tables troviamo che la Relation table può definire entità di tipo Relation, mentre la Requirement-Relation table può quotare Requirement e stessa cosa può avvenire per la LegacySystem-Relation table con elementi di tipo LegacySystem. In fine Relation può essere in relazione sia con Requirement, sia con LegacySystem.

2.7.3 ANALISI

Il processo è definito in figura 2.19:

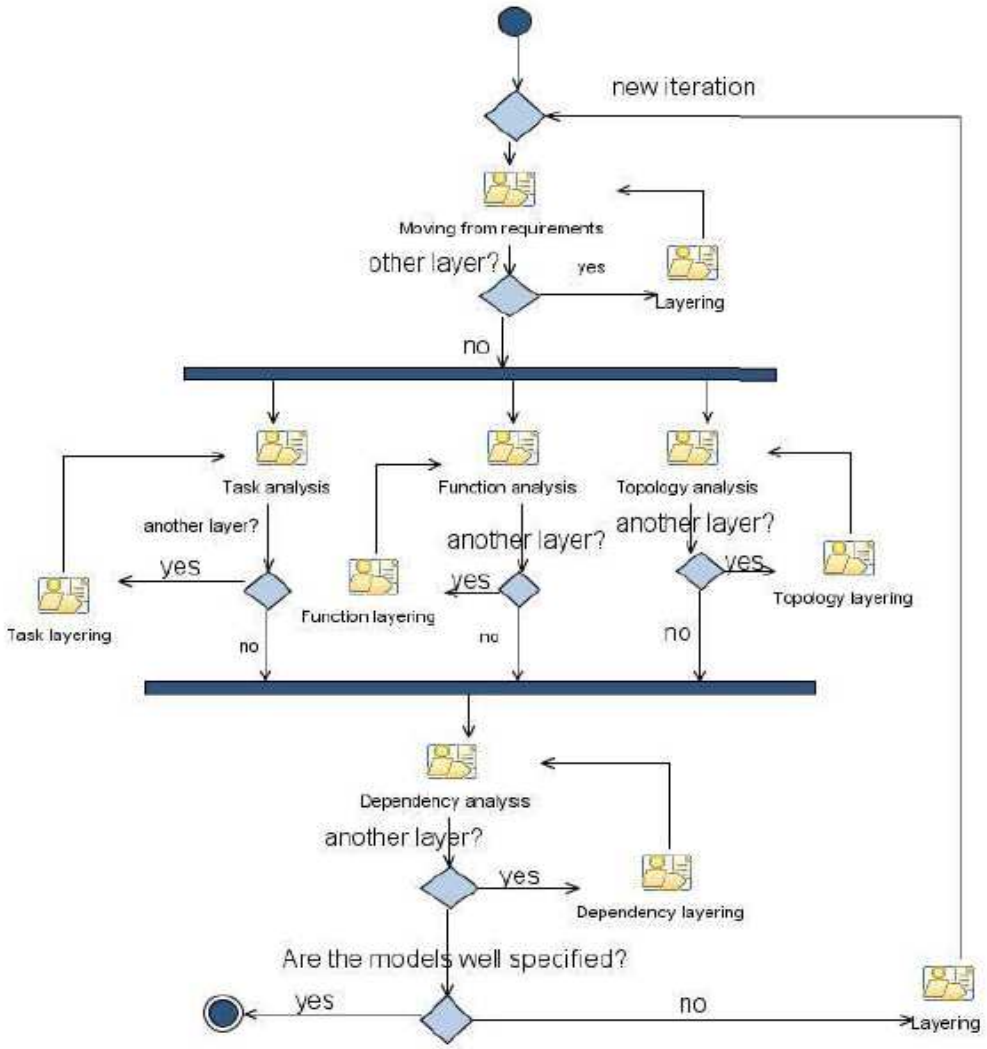


Figura 2.19 Processo della fase di Analisi

La fase di Analisi segue quella di Analisi dei Requisiti, per cui è necessario mettere in corrispondenza le entità delle due fasi.

Partendo ad analizzare il flusso dell'attività di analisi, in primo luogo è necessario compiere la *Migrazione dei requisiti* ("Moving from requirements" in Figure 2.14). Questo primo step va ripetuto per ognuno dei layer che erano stati definiti nella fase di Analisi dei Requisiti, utilizzando l'operazione di layering. Come risultato finale si ottengono le mappature delle entità individuate durante la fase precedente di Analisi dei Requisti.

I documenti in ingresso a questa attività sono:

- Requirements tables
- Relation tables
- Domain tables
- Zooming table

In uscita si ottengono:

- Reference Requirement-Function table
- Reference Requirement-Task table
- Reference Requirement-Dependency table
- Reference Requirement-Topology table
- Reference LegacySystem-Function table
- Reference LegacySystem-Topology table
- Reference Relation-Dependency table

Tutte queste tabelle formano l'insieme REFERENCE TABLES (figura 2.20).

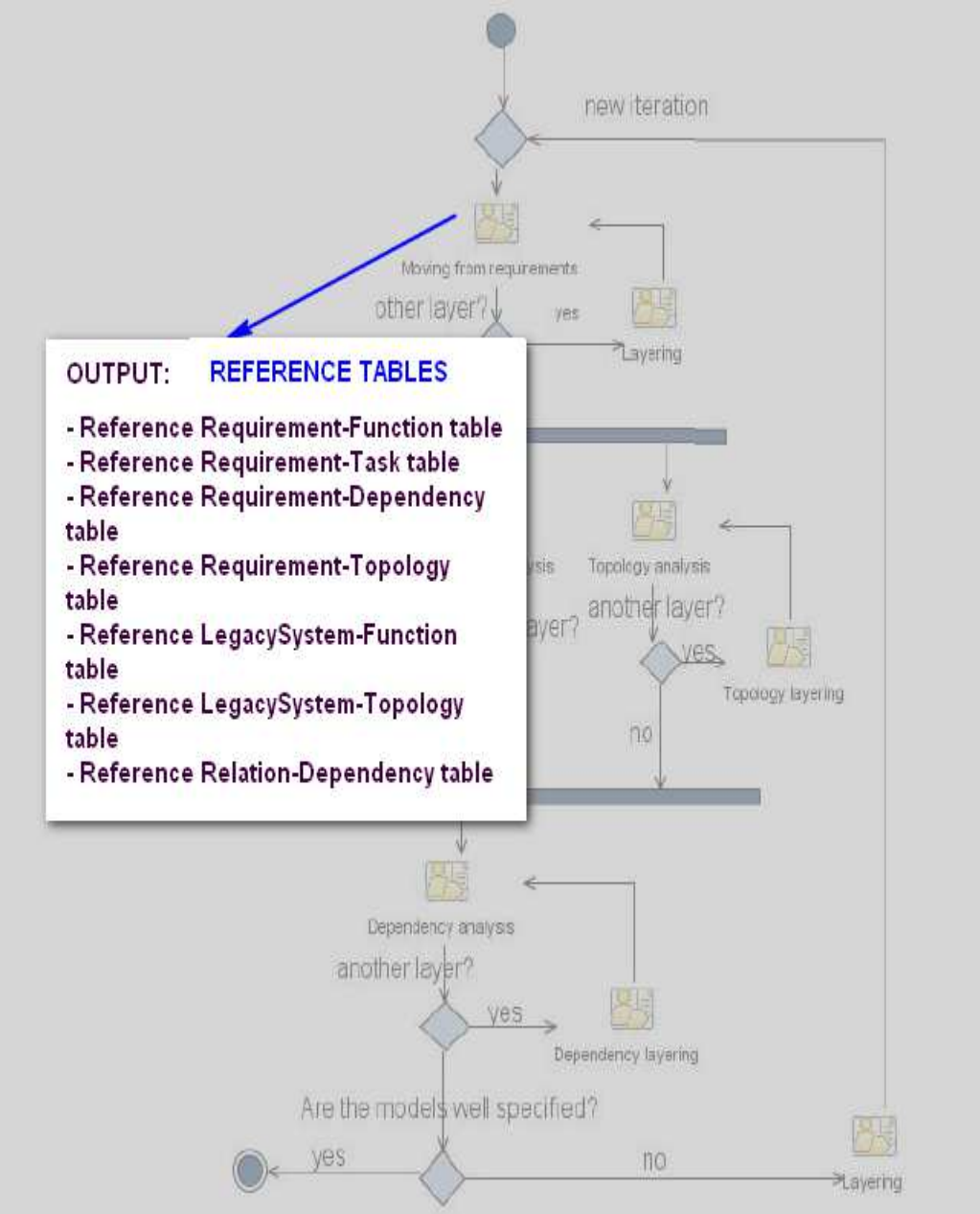


Figura 2.20 Output del primo step della fase di Analisi.

L'*Analisi dei task* formula una descrizione dei singoli task individuati nel sistema, partendo dalla Reference Requirement-Task table (dell'attività precedente) e dalla Zooming table, producendo in uscita la Task table.

L'*Analisi delle Funzioni* descrive le funzioni necessarie per portare a compimento i task, prendendo in ingresso la Reference Requirement-Function table, ottenuta nella migrazione dai requisiti, e dalla Zooming table, dando come risultato finale la Function table.

L'*Analisi delle Topologie* prevede di analizzare, oltre alla Zooming table, la Reference LegacySystem-Topology table e la Reference Requirement-Topology table.
I risultati di output sono la Function-Topology table, la Task-Topology table e la Topology table che descrive le singole topologie trovate e le descrive.

L'*Analisi delle Dipendenze* che mappa le Relazioni individuate durante l'Analisi dei Requisiti in Dipendenze.
Tra le tabelle in ingresso abbiamo la Reference Relation-Dependency table e la Reference Requirement-Dependency table (dalla migrazione dai requisiti), assieme a Topology, Task e Function table (ottenute mediante le tre fasi di analisi dei Task, Funzioni e Topologie) ed alla Zooming table.
Le tabelle prodotte elencano le dipendenze individuate e le mettono in relazione a task, funzioni e topologie:

- Dependency table
- Topology-Dependency table
- Function-Dependency table
- Task-Dependency table

In figura 2.21 viene riportato un dettaglio delle tabelle prodotte in output.

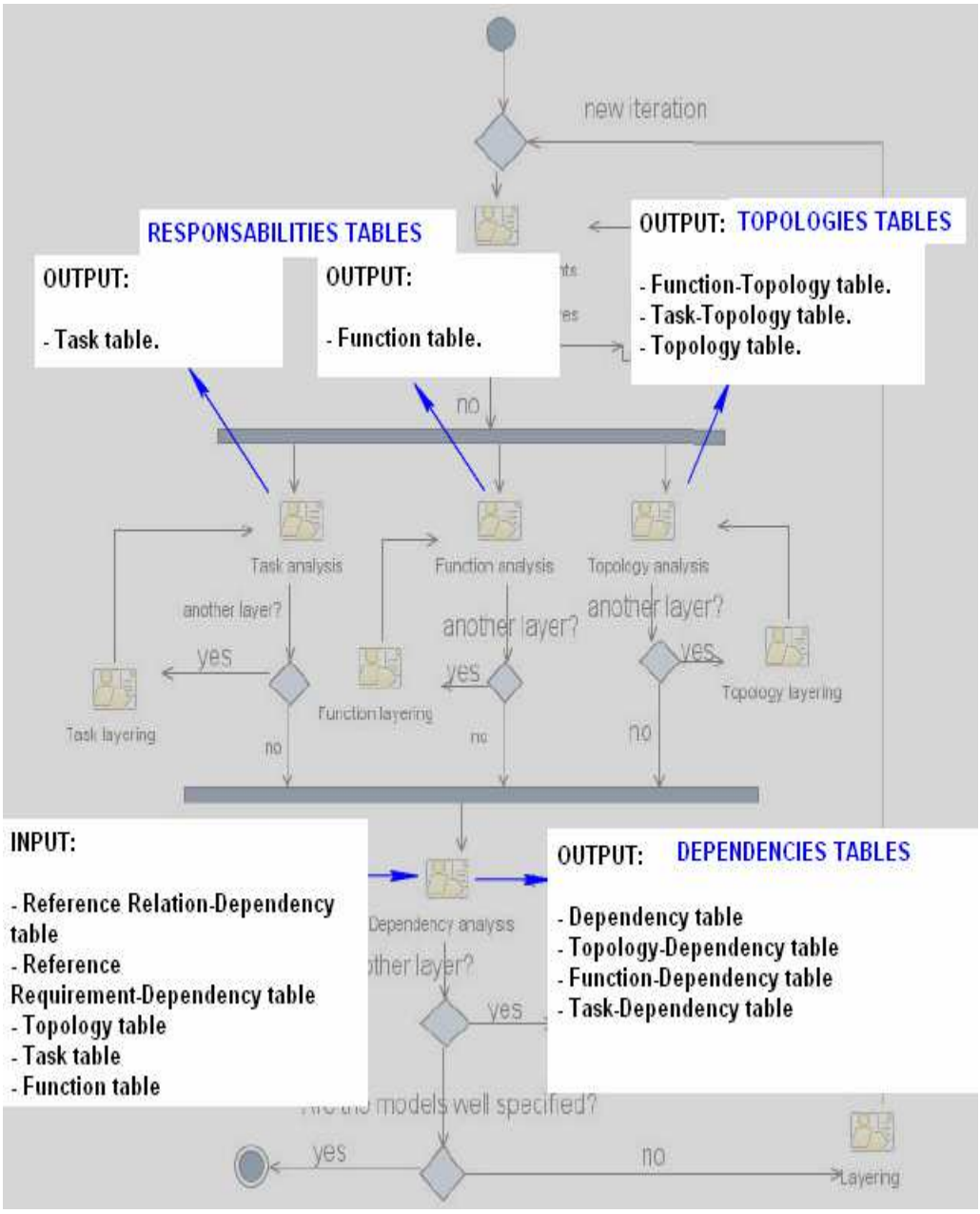


Figura 2.21 Tabelle prodotte in output in fase di analisi.

Per dare una visione d'insieme del flusso delle attività che compongono l'analisi con i relativi documenti di input e output, vediamo quanto riportato in figura 2.22.

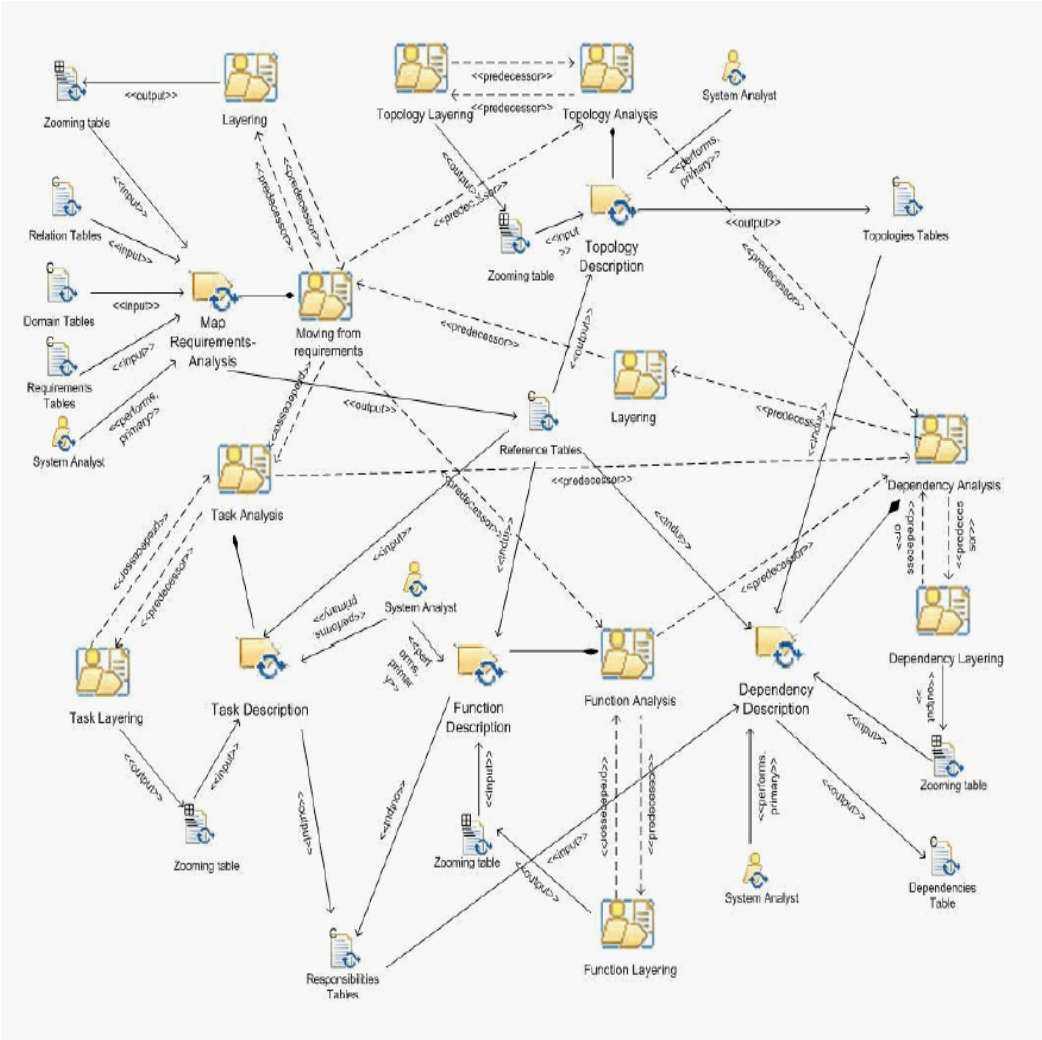


Figura 2.22 Flusso delle attività nella fase di Analisi.

In figura possiamo vedere raffigurato il flusso di analisi con tutte le tabelle ingresso/uscita, che intervengono ed in che momento. Da notare la Refernce Tables, prodotte della Migrazione dai Requisiti, costituiscono l'input per tutte le attività successive, è quindi importante il ruolo della mappatura tra le entità delle fasi di Analisi dei Requisiti e Analisi.

In conclusione *dall'analisi* si ottengono quattro categorie di tabelle (come evidenziato anche in figura 1.20 e 1.21):

REFERNCE TABLES: che consentono di associare le astrazioni utilizzate nell'Analisi dei Requisiti con quelle usate nella fase di Analisi:

- Reference Requirement-Task table ((L)RRT$_t$) associa ogni requisito ai task generati;

- Reference Requirement-Function table ((L)RRF$_t$) mette in relazione ogni requisito con le corrispondenti funzioni;

- Reference Requirement-Topology table ((L)RRTo$_t$) esplicita il legame tra ogni requisito e le relative topologie;

- Reference Requirement-Dependency table ((L)RReqD$_t$) mappa le associazioni tra i requisiti e le dipendenze generate;

- Reference LegacySystem-Function table ((L)RLSF$_t$) specifica le relazioni tra i sistemi legacy e le funzioni corrispondenti;

- Reference LegacySystem-Topology table ((L)RLSF$_t$) associa sistemi legacy e topologie;

- Reference Relation-Dependency table ((L)RRelD$_t$) stabilisce la corrispondenza tra le Relazioni individuate nell'Analisi dei Requisiti e le Dipendenze presenti nell'Analisi.

Requirement	Task
Requirement name	Task names

(L)RRT$_t$

Requirement	Function
Requirement name	function names

(L)RRF$_t$

Requirement	Topology
Requirement name	topology names

(L)RRTo$_t$

Requirement	Dependency
Requirement name	dependency names

(L)RReqD$_t$

Legacy System	Function
Legacy-System name	function names

(L)RLSF$_t$

Legacy System	Topology
Legacy-System name	topology names

(L)RLST$_t$

Relation	Dependency
Relation name	Dependency names

(L)RRelD$_t$

Le *Dependencies Tables* esplicitano le relazioni tra i task e le funzioni:

- Dependency table ((L)D$_t$) descrive le dipendenze tra le entità astratte;

- Task-Dependency table ((L)TD$_t$) per ogni task elenca tutte le dipendenze in cui è coinvolto;

- Function-Dependency table ((L)FD$_t$) contiene la lista delle dipendenze in cui compare ogni singola funzione;

- Topology-Dependency table ((L)TopD$_t$) specifica per ogni topologia tutte le dipendenze in cui compare.

Dependency	Description
Dependency name	Dep. description

$(L)D_t$

Task	Dependency
Task name	Dependency names

$(L)TD_t$

Function	Dependency
function name	Dependency names

$(L)FD_t$

Topology	Dependency
topology name	Dependency names

$(L)TopD_t$

Le *Responsibilities Tables* definiscono le entità, quali task e funzioni, che rappresentano il concetto di "centro di responsabilità":

- Task table ($(L)T_t$) fornisce l'elenco di tutti i Task individuati e la relativa descrizioni;

- Function table ($(L)F_t$) descrive le singole Funzioni presenti nel sistema.

Task	Description
Task name	task description

$(L)T_t$

Function	Description
Function name	Func. description

$(L)F_t$

Le *Topologies Tables* consentono di esprimere i vincoli topologici sull'ambiente:

- Topology table ((L)Top_t) descrive i singoli vincoli topologici;

- Task-Topology table ((L)$TTop_t$) associa ogni task ai rispettivi vincoli;

- Function-Topology table ((L)$FTop_t$) elenca per ogni funzione i vincoli topologici in cui è coinvolto.

Topology	Description
topology name	Top. description

(L)Top_t

Task	Topology
task name	topology names

(L)$TTop_t$

Function	Topology
function name	topology names

(L)$FTop_t$

2.7.4 RELAZIONI TRA LE TABELLE PRODOTTE DALL'ANALISI E GLI ELEMENTI DEL META-MODELLO

Prendendo in esame la figura 2.23, vediamo le relazioni tra le tabelle coinvolte nella fase di Analisi con gli elementi del meta-modello.

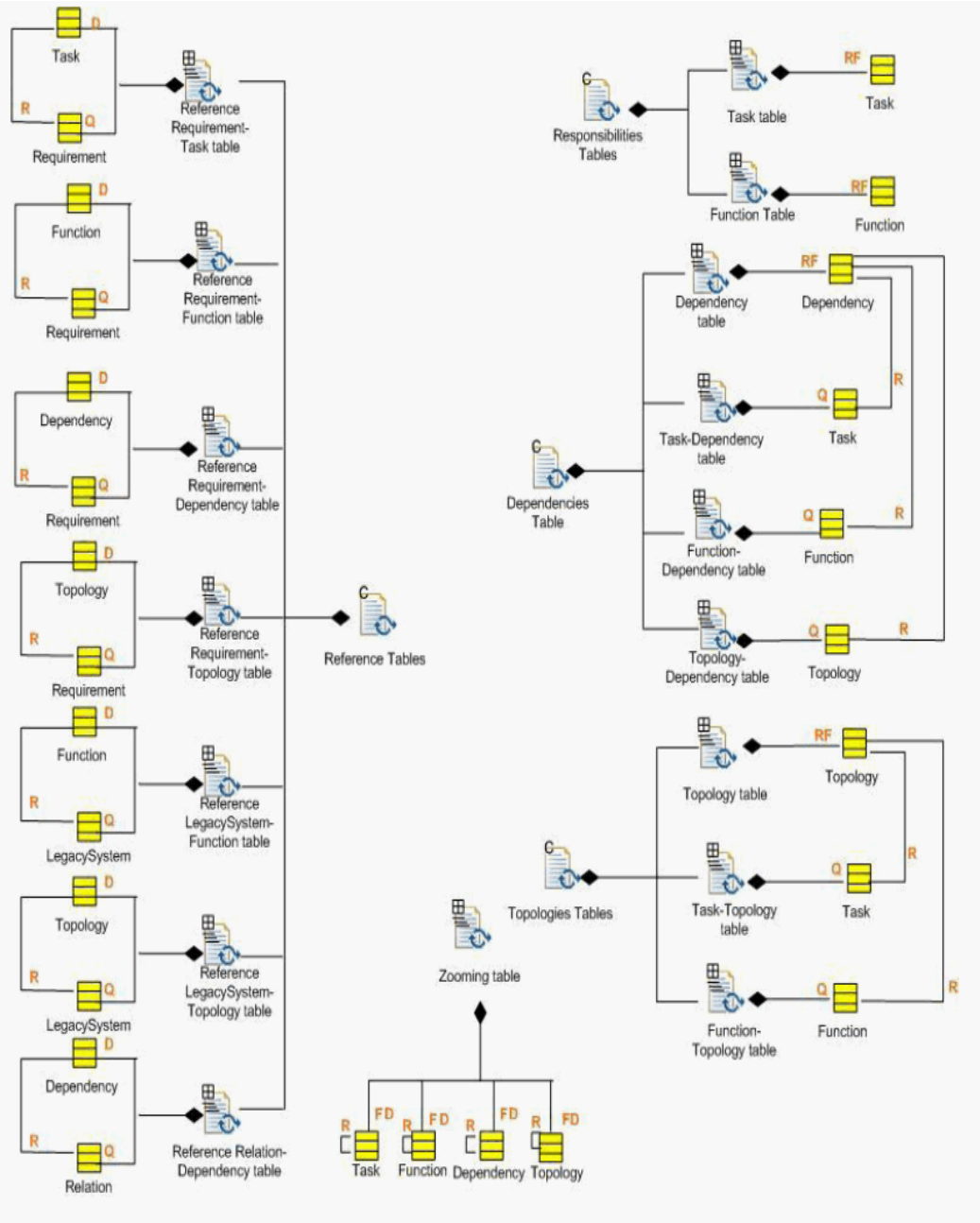


Figura 2.23 Relazioni tra work products dell'Analisi ed elementi del meta-modello

La *Zooming table* definisce gli elementi Task, Function, Dependency e Topology, queste a loro volta sono relazionate a loro stesso.

La *Reference tables* hanno tutte un legame comune con l'elemento Requirement, che di volta in volta è in relazione con l'elemento che viene definito in base al tipo di tabella (es. la Reference Requirement Task table definisce nuove entità di tipo Task, che sono in relazione con Requirement già definito in precedenza, qui riutilizzato).

La *Responsabilities tables* mette in relazione Task table e Function table, associate rispettivamente a Task e Function con legami di rifinitura (F) e relazione (R).

La *Dependencies Tables* riutilizza i Task, Function e Topology già definite (relazione di tipo Q), mentre la Dependecy table rifinisce gli elementi di tipo Dependecy, che è in relazione (R) con gli elementi Task, Function e Topology per quel che riguarda le tabelle Task-Dependency, Function-Dependency e Topology-Dependency.

Analogo discorso per le *Topologies Tables*, infatti Topology table può rifinire Topology, che è in relazione con Task e Function, quotati rispettivamente da Task-Topology table e Function-Topology table.

2.7.5 DESIGN ARCHITETTURALE

Il processo è definito in figura 2.24:

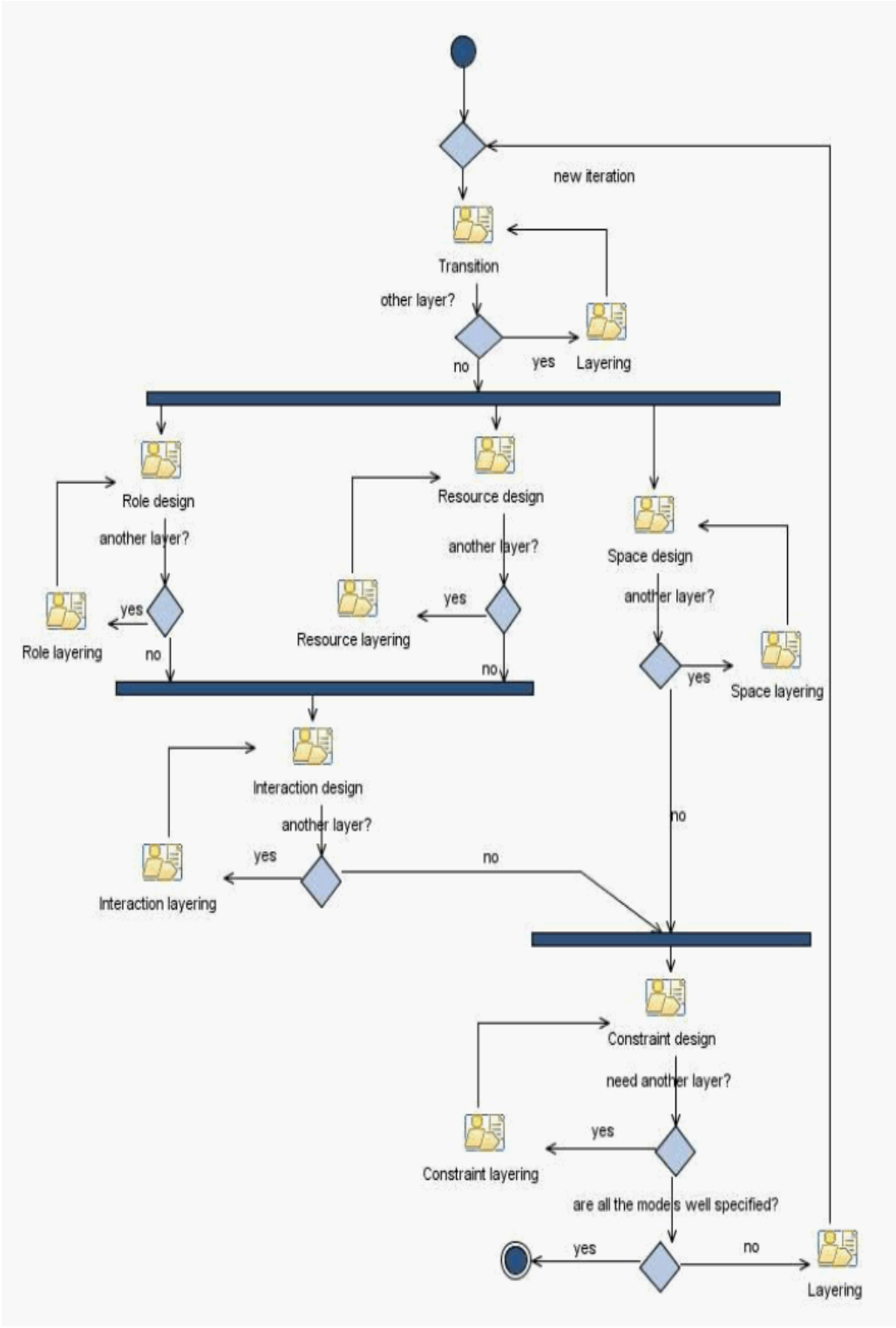


Figura 2.24 Processo della fase di Architectural Design

Questa è la terza fase del processo SODA, in cui si portano le entità precedenti ad un livello meno astratto.

Come si vede, nel processo di Design architetturale (figura 2.24) si ha come primo step:

la *Transition*, che come la Migrazione dai Requisiti, qui si ha la congiunzione tra l'Analisi ed il Design. Il passaggio dalla seconda a questa terza fase è reiterabile tramite il layering, per ogni livello di layering presente nella fase di Analisi.

Successivamente segue:

in *Role Design* vengono definiti i ruoli, vale a dire le entità attive in grado di compiere azioni e quali.

In *Resource Design* si definiscono le risorse coinvolte, vale a dire le entità passive in grado di fornire servizi e quali.

In *Space Design* viene definita la struttura dell'ambiente

Sia Role Design, Resource Design che Space Design sono iterabili tramite layering.

Role e Resource convergono al passo successivo che è rappresentato dall'*Iteraction Design*, in cui vengono specificati i legami ruoli e risorse; anche in questo step abbiamo la presenza del layering.

Come step finale del processo abbiamo il *Constraint Design* in cui vengono individuati i vincoli relativi alle entità ruoli, risorse, interazioni e spazi individuati.

Terminata questa operazione si verifica se sia necessario specificare meglio il modello complessivo ottenuto durante tutto il processo della fase di Design Architetturale e, in caso positivo, si applica nuovamente l'operazione di Layering e si riparte dalla Transizione, ripetendo tutti i passi.

In figura 2.25 è possibile vedere le tabelle prodotte durante tutto il processo del Design Architetturale:

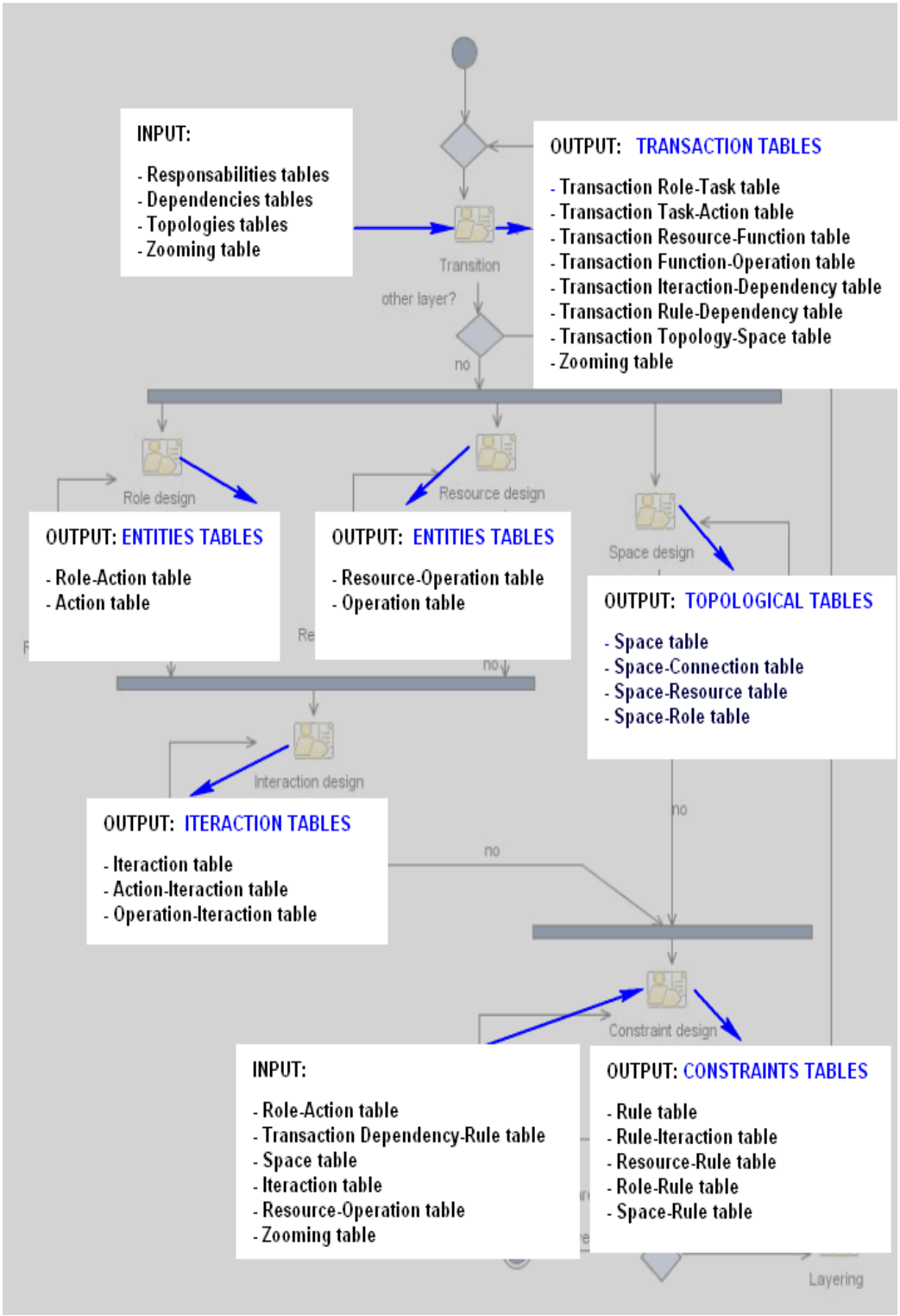


Figura 2.25 Tabelle prodotte in output nel processo di Design Architetturale

Per dare una visione d'insieme del flusso delle attività che compongono l'Architectural Design con i relativi documenti di input e output, vediamo quanto riportato in figura 2.26.

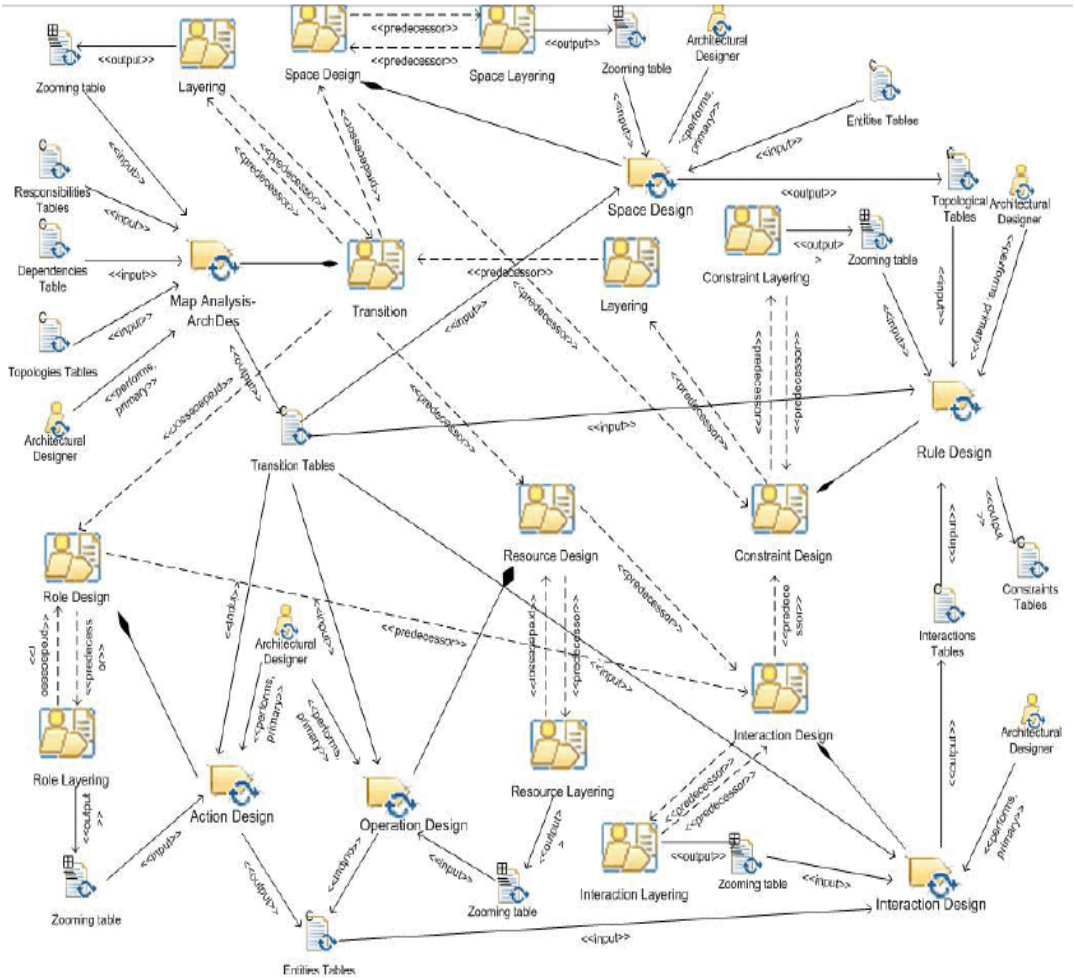


Figura 2.26 Flusso delle attività nella fase di Architectural Design

I vincoli di sequenza delle attività sono esplicitati anche in base agli input e agli output; possiamo notare che le Transitions Tables sono utilizzate da tutte le attività successive alla Transizione, quindi essa rappresenta un'operazione fondamentale per il Design. Inoltre le Entities Tables compaiono come input anche nel Design degli Sazi, quindi questa azione dovrà seguire il Design delle Risorse e quello dei Ruoli.

In conclusione *dall'Architectural Design* si ottengono cinque categorie di tabelle (come evidenziato anche in figura 2.25):

Le *Transition Tables* realizzano il collegamento tra la fase di Analisi e quella di Design Architetturale, mappando le astrazioni della prima in quelle della seconda:

- Transition Role-Task table ((L)TRT_t) associa i task ai ruoli;

- Transition Task-Action table ((L)TTA_t) mette in relazione task e azioni;

- Transition Resource-Function table ((L)TRF_t) indica a quali funzioni è legata ogni risorsa;

- Transition Function-Operation table ((L)TFO_t) esprime i legami tra funzioni e operazioni;

- Transition Interaction-Dependency table ((L)TID_t) traduce le dipendenze, stabilite nella fase di Analisi, in Interazioni utilizzate dal Design;

- Transition Rule-Dependency table ((L)$TRuD_t$) mappa le dipendenze in regole;

- Transition Topology-Space table ((L)$TTopS_t$) realizza la corrispondenza tra topologie e spazi per quel che riguarda l'ambiente.

Role	Task
Role name	Task names

(L)TRT_t

Task	Action
Task name	action names

(L)TTA_t

Resource	Function
Resource name	function names

(L)TRF_t

Function	Operation
function name	operation names

(L)TFO_t

Dependency	Interaction
Dependency name	interaction names

$(L)TID_t$

Dependency	Rule
Dependency name	rule names

$(L)TRuD_t$

Topology	Space
topology name	space names

$(L)TTopS_t$

Le *Entities Tables*, prodotte da Design dei Ruoli e Design delle Risorse, consentono di specificare le entità attive e quelle passive del sistema; le prime sono accompagnate dalle azioni che sono in grado di svolgere, mentre con le seconde sono descritti i servizi che esse forniscono.

- Action table ($(L)A_t$) elenca e descrive le azioni eseguibili da qualche ruolo;

- Operation table ($(L)O_t$) indica le operazioni che una risorsa espone al sistema;

- Role-Action table ($(L)RA_t$) per ogni ruolo fornisce le azioni che può eseguire;

- Resource-Operation table ($(L)RO_t$) stabilisce per ogni risorsa quali operazioni può svolgere.

Action	Description
action name	action description

$(L)A_t$

Operation	Description
operation name	action description

$(L)O_t$

Role	Action
role name	action names

(L)RA_t

Resource	Operation
resource name	operation names

(L)RO_t

Le *Interaction Tables* derivano dal Design delle Interazioni, quindi descrivono le interazioni tra ruoli e risorse:

- Interaction table ((L)I_t) descrive ogni singola interazione;

- Action-Interaction table ((L)AcI_t) elenca per ogni azione le interazioni in cui compare;

- Operation-Interaction table ((L)OpI_t) associa ogni operazione alle interazioni che la riguardano.

Interaction	Description
interaction name	interaction description

(L)I_t

Action	Interaction
action name	interaction names

(L)AcI_t

Operation	Interaction
operation name	interaction names

(L)OpI_t

Le *Constraints Tables*, prodotte dal Design dei Vincoli, riportano i vincoli che caratterizzano il comportamento delle entità:

- Rule table ((L)Ru_t) elenca e descrive tutte le regole;

- Rule-Interaction table ((L)IRu_t) esprime i vincoli sulle interazioni;

- Resource-Rule table ((L)ReRu$_t$) stabilisce le regole in cui ogni risorsa è coinvolta;

- Role-Rule table ((L)IRoRu$_t$) elenca le regole in cui compare ogni ruolo;

- Space-Rule table ((L)SRu$_t$) specifica le regole che caratterizzano ogni spazio.

Rule	Description
rule name	description

(L)Ru$_t$

Interaction	Rule
Interaction name	Rule names

(L)IRu$_t$

Resource	Rule
resource name	Rule names

(L)ReRu$_t$

Role	Rule
role name	Rule names

(L)RoRu$_t$

Space	Rule
space name	Rule names

(L)SRu$_t$

Le *Topological Tables* sono il risultato del Design delle Topologie e caratterizzano la struttura logica dell'ambiente:

- Space table ((L)S$_t$) contiene la descrizione di tutti gli spazi;

- Space-Connection table ((L)SC$_t$) all'interno di un livello specifica i collegamenti tra i diversi spazi (per avere una visione gerarchica degli spazi si deve utilizzare la Zooming table);

- Space-Resource table ((L)SRe$_t$) elenca per ogni risorsa gli spazi in cui è utilizzata;

- Space-Role table ((L)SRo$_t$) stabilisce per ogni ruolo gli spazi in cui compare.

Space	Description
space name	description

(L)S$_t$

Space	Connection
space name	ronnection names

(L)SC$_t$

Space	Resource
space name	resource names

(L)SRe$_t$

Space	role
space name	role names

(L)SRo$_t$

2.7.6 RELAZIONI TRA LE TABELLE PRODOTTE DAL DESIGN ARCHITETTURALE E GLI ELEMENTI DEL META-MODELLO

Prendendo in esame la figura 2.27, vediamo le relazioni tra le tabelle coinvolte nella fase di Architectural Design con gli elementi del meta-modello.

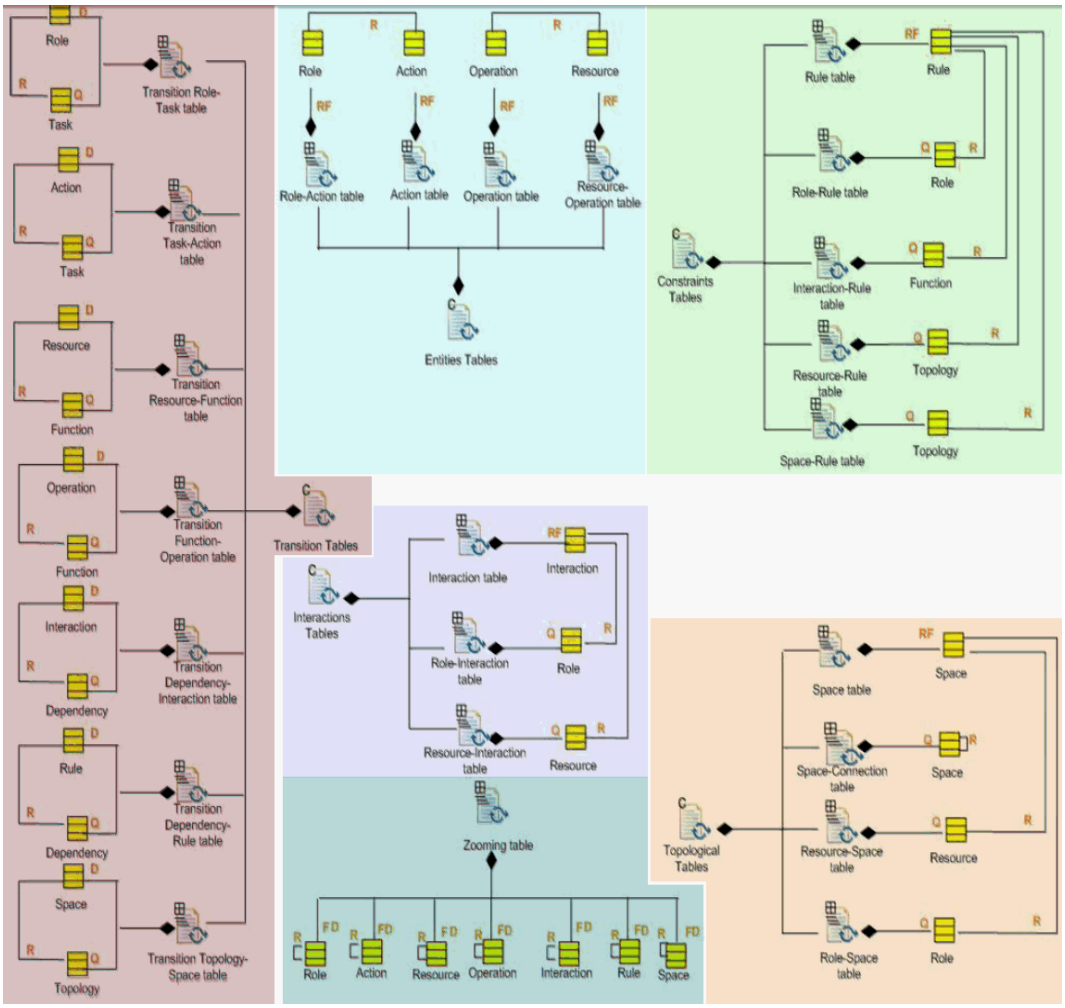


Figura 2.27 Relazioni tra work products dell'Architecture Design ed elementi del meta-modello

La *Zooming table* ha ralazioni di definizione (D), rifinitura (F) e relazione semplice (R) con: Role, Acton, Resource, Operation, Iteraction, Rule e Space.

Le *Entities tables* hanno legami di rifinitura (F) e relazione semplice (R) con Role, Action, Operation e Resource. Le corrispondenti tabelle sono: Role-Action table, Action table, Operation table e Resource-Operation table.

Le *Transaction tables* hanno una relazione semplice (R), uno di definizione (D) e uno di riutilizzo (Q) nel dettaglio:

- Trans. Role-Task t. e Trans. Task-Action t. quotano Task e definiscono rispettivamente Role e Action, che a loro volta sono in relazione con Task;

- Trans. Resource-Function t. e Trans. Function-Operation t. quotano Function e definiscono Resource e Operation, in relazione con Function;

- Trans. Dependency-Interaction t. e Trans. Dependency-Rule t. quotano Rependency e definicono Interaction e Rule, posti in relazione con Dependency;

- Trans. Topology-Space t. definisce Space e riutilizza Topology; i due elementi sono in relazione tra loro.

Nelle *Interaction Tables* osserviamo che:

- *Interaction table* è in relazione con Interaction e la può rifinire; questa è in realzione con Role e Resource, quotate rispettivamente da Role-Interaction table e Resource-Interaction table.
- *Constraints Tables*, infatti Rule table rifinisce ed è in relazione con Rule. A sua volta Rule è in relazione con: Role (riutilizzato da Role-Rule t.), Function (riutilizzato da Interaction-Rule t.), Resource (quotato da Resource-Rule t.) e Topology (quotato da Space-Rule t.).
- Le *Topological Tables* riguardano gli spazi, per cui Space table è in relazione con Space e lo può rifinire; questo è poi in relazione con Resource e Role, quotati da Resource-Space table e Role-Space table. La Space-Connection table invece riutilizza Space (Q), che in questo caso è in relazione con se stesso.

In conclusione, si può osservare che tutti gli elementi del meta-modello utilizzati nel Design Architetturale vengono definiti inizialmente nell‘attività di Transizione. In seguito soltanto il Layering può definire nuove entità, quindi questo risulta nella Zooming table, mentre le altre attività riutilizzano entità già definite, o eventualmente ne possono modificare le proprietà.

2.7.7 DESIGN DI DETTAGLIO

Rappresenta l'ultima fase del processo SODA, in cui viene scelta la struttura definitiva del sistema, individuando il livello di rappresentazione di ognuna delle entità architetturali definite nel processo di progettazione. Il processo è definito in figura 2.28:

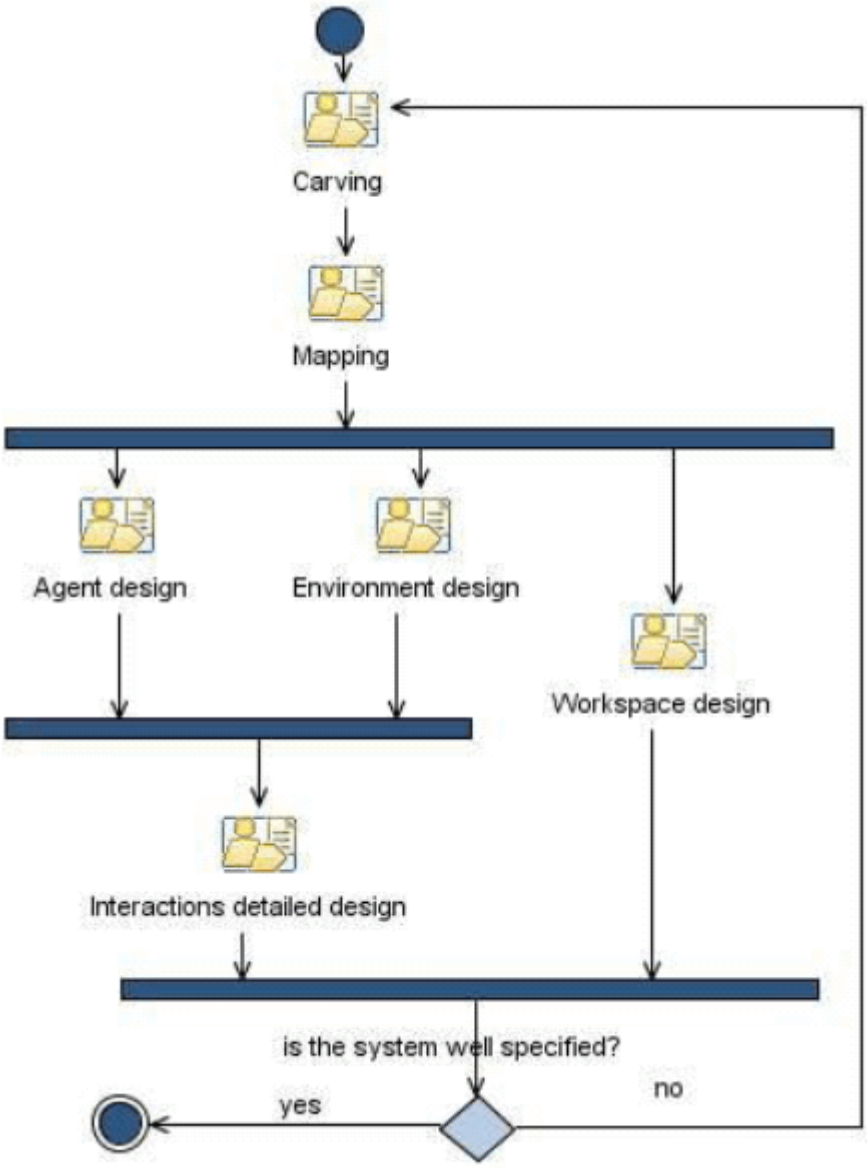


Figura 2.28 Processo della fase di Design Destail

Dalla figura, analizziamo i vari step:

Il *Carving* in cui partendo dal livello principale (core layer) si scelgono le entità soggette a in-zooming, stabilendo quali prendere a livello base e quali a livello sottostante, determinando la scelta del livello di astrazione e

architetturale che rappresenta il sistema.
Una volta fatto questo, l'insieme delle entità viene estratto dal design del sistema (operazione di carving-out).

Per le entità ambientali, si mappano le risorse non soggette a zoom in artefatti (entità che forniscono servizi), mentre quelle ottenute da in-zoom sono tradotte in aggregati di artefatti.

Gli ruoli sono messi in corrispondenza con gli agenti (se non zoomati) o con società di agenti (se sottoposti a in-zoom).

Questo passaggio sostituisce il layering nella fase di Design di Dettaglio, in quanto a questo punto potrebbero essere presenti tutti i possibili livelli individuati nelle fasi precedenti ed ognuno, se completo, rappresenterebbe un'architettura indipendente, rendendo impossibile implementare il sistema; per questo motivo nel Carving si opera una scelta della struttura da adottare.

E' possibile reiterare la fase di Carving, nel caso il sistema non fosse completamente specificato.

Il *Mapping* che serve a mettere in corrispondenza le entità della fase precedente, vale a dire il design architetturale, con le entità di quest'ultima.

L'Agent Design una delle tre attività che si hanno dopo il *Mapping*, serve a definire il design dei singoli agenti, artefatti e società si agenti che derivano dall'operazione di *Carving.*

Environment Design si sviluppa in parallelo alla precedente attività, definisce il design degli artefatti e dei loro aggregati, derivanti anch'essi dal *Carving.*

L'Agent Design e l'Environmet Design convergono nell'attività di: *Iteractions Detailed Design* in cui vengono definite le tipologie di comunicazione tra le entità:

- "*Use":* che indica l'uso di un artefatto da parte di un agente.
- "*SpeakTo":* con cui un agente interagisce con un altro.
- "*Manifest":* attraverso cui un artefatto segnala la propria presenza ad un agente.
- "*LinkedTo":* che esplicita il collegamento tra due artefatti.

WorkSpace Design in cui si determina la struttura finale dell'ambiente in cui si trova il sistema, descrivendo i singoli workspace, le connessioni tra workspace diversi e le relazioni tra workspace con agenti e artefatti.

Una volta ottenuto il modello complessivo del sistema, si verifica se è ben specificato, valutando il caso di ripetere o meno il processo di Design di Dettaglio. Al termine , si conclude anche tutto il procedimento SODA.

Si può notare la totale assenza del layering, in quanto si ha bisogno in questa fase, di portare le entità verso il concreto, scegliendo il livello definitivo che rappresenti il sistema tra i differenti livelli identificati durante tutto il procedimento SODA.

In figura 2.29 è possibile vedere le tabelle prodotte durante tutto il processo del Design di Dettaglio, avendo un dettaglio delle singole attività, poste in relazione ai rispettivi input e output:

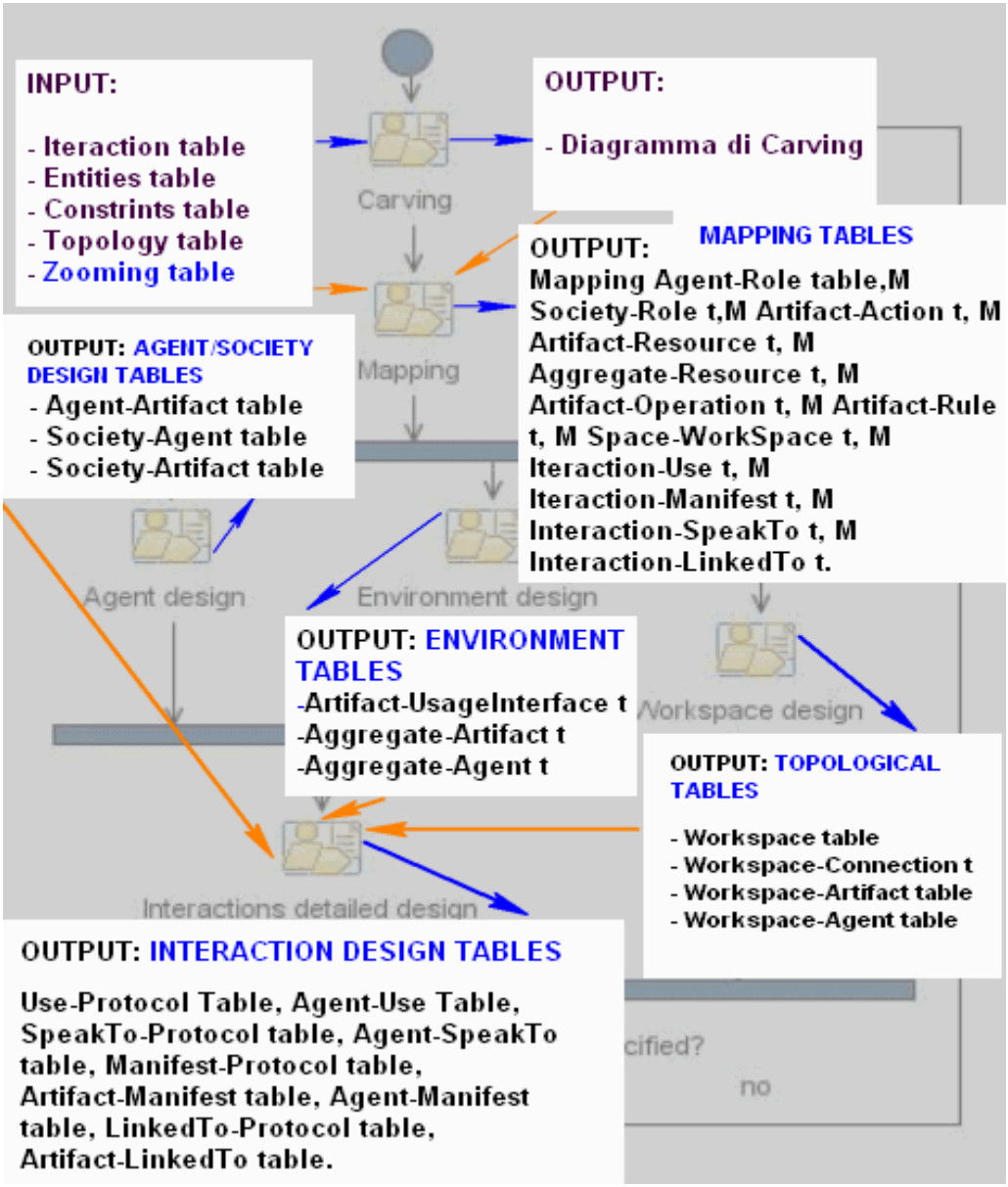


Figura 2.29 Tabelle prodotte in output nel processo di Design di Dettaglio

Per dare una visione d'insieme del flusso delle attività che compongono il Design di Dettaglio con i relativi documenti di input e output, vediamo quanto riportato in figura 2.30.

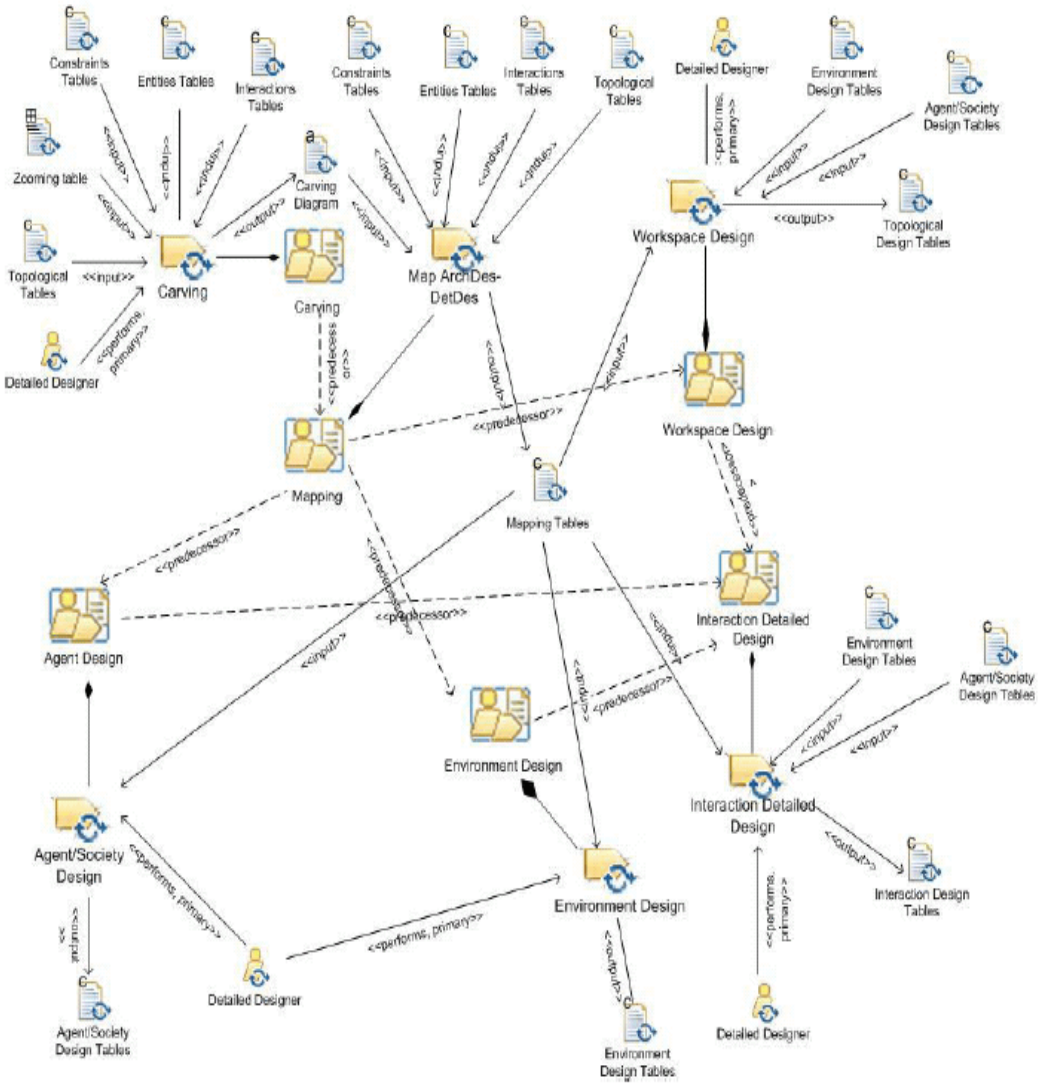


Figura 2.30 Flusso delle attività nella fase di Design di Dettaglio

L'ultima fase di SODA, produce i seguenti insiemi di tabelle:

Le *Mapping Tables*, prodotte dall'attività di Mapping, mettono in corrispondenza le entità utilizzate nel Design Architetturale con quelle del Design di Dettaglio:

- Mapping Agent-Role table(MAR_t) traduce i ruoli in agenti;

- Mapping Society-Role t (MSR_t) mette in corrispondenza ruoli e società di agenti;

- Mapping Artifact-Action t ($MAAc_t$) mappa le azioni in singoli artefatti;

- Mapping Artifact-Resource t ($MArR_t$) associa gli artefatti alle risorse definite in precedenza;

- Mapping Aggregate-Resource t ($MAggR_t$) traduce le operazioni in aggregati di artefatti;

- Mapping Artifact-Operation t ($MArOp_t$) associa artefatti ambientali alle operazioni;

- Mapping Artifact-Rule t ($MArRu_t$) mappa le regole stabilite nella fase precedente in artefatti che le implementano;

- Mapping Space-Workspace t (MSW_t) mappa gli spazi in workspaces;

- Mapping Interaction-Use t (MIU_t) esplicita le interazioni di tipo Use;

- Mapping Interaction-Manifest t (MIM_t) esplicita le interazioni di tipo Manifest, cioè quelle in cui un artefatto indica ad un agente la propria presenza;

- Mapping Interaction-SpeakTo t ($MISp_t$) esplicita interazioni di tipo SpeakTo;

- Mapping Interaction-LinkedTo t (MIL_t) esplicita interazioni di tipo LinkedTo.

Agent	Role
agent name	Role names

$(L)MAR_t$

Society	Role
society name	Role names

$(L)MSR_t$

(Individual) artifact	Action
artifact name	action names

(L)MAAc$_t$

(Env.) artifact	Resource
artifact name	resource names

(L)MArR$_t$

Aggregate	Resource
aggregate name	resource name

(L)MAggR$_t$

(Env.) artifact	Operation
artifact name	operation names

(L)MArOp$_t$

Rule	Artifact
rule name	artifact names

(L)MArRu$_t$

Workspace	Space
workspace name	space names

(L)MSW$_t$

Interaction	Use
interaction name	use names

(L)MIU$_t$

Interaction	Manifest
interaction name	manifest names

(L)MIM$_t$

Interaction	SpeakTo
interaction name	speakTo names

(L)MISp$_t$

Interaction	LinkedTo
interaction name	linkedTo names

(L)MIL$_t$

Le *Agent/Society Design Tables* sono il risultato del Design degli Agenti e descrivono le società di agenti e i loro componenti:

- Agent-Artifact t (AA$_t$) indica per ogni agente gli artefatti ad esso collegati;

- Society-Agent t (SA$_t$) contiene l'elenco degli agenti appartenenti ad ogni società;

- Society-Artifact t (SAr$_t$) fornisce la lista degli artefatti compresi in ogni società.

Agent	(Individual) Artifact
agent name	artifact names

(L)AA_t

Society	Agent
society name	agent names

(L)SA_t

Society	Artifact
society name	artifact names

(L)SAr_t

Le *Environment Design Tables* derivano dal Design dell'Ambiente e specificano la composizione degli aggregati di artefatti:

- Artifact-UsageInterface t ($AUIr_t$) espone le singole operazioni fornite dagli artefatti;
- Aggregate-Artifact t ($AggArt_t$) elenca per ogni aggregato gli artefatti che lo compongono;
- Aggregate-Agent t ($AggAge_t$) specifica quali agenti appartengono agli aggregati;

In queste tabelle non viene presa in considerazione la distinzione tra le diverse tipologie di artefatti, in quanto gli aspetti che vengono trattati sono comuni a tutti i tipi.

Artifact	Usage interface
artifact name	List of operations

(L)AUI$_t$

Aggregate	Artifact
aggregate name	artifact names

(L)AggArt$_t$

Aggregate	Agent
aggregate name	agent names

(L)AggAge$_t$

Le *Interaction Design Tables*, definite dal Design delle Interazioni, riguardano i diversi tipi di relazione che ci possono essere tra le entità che compongono il sistema:

- Use-Protocol t (UP$_t$) descrive i protocolli per ogni interazione di tipo ―Use;
- Agent-Use (AgeU$_t$) specifica le Use in cui ogni agente è coinvolto;
- Artifact-Use (ArtU$_t$) specifica le Use in cui ogni artefatto è coinvolto;
- SpeakTo-Protocol t (SP$_t$) descrive i protocolli per ogni interazione di tipo SpeakTo;
- Agent-SpeakTo (AgeSP$_t$) specifica le SpeakTo in cui ogni agente è coinvolto;
- Manifest-Protocol t (MP$_t$) descrive i protocolli per ogni interazione di tipo Manifest;
- Artifact-Manifest (ArtM$_t$) specifica le Manfest in cui ogni artefatto è coinvolto;
- Agent-Manifest (AgeM$_t$) specifica le Manfest in cui ogni agente è coinvolto;
- LinkedTo-Protocol t (LP$_t$) descrive i protocolli per ogni interazione di tipo LinkedTo;
- Artifact-LinkedTo (ArtL$_t$) specifica le LinkedTo in cui ogni artefatto è coinvolto;

Use	Protocol
use name	Protocol descr.

(L)UP_t

Agent	Use
agent name	use names

(L)$AgeU_t$

Artifact	Use
artifact name	use names

(L)$ArtU_t$

SpeakTo	Protocol
speak name	Protocol descr.

(L)SP_t

Agent	SpeakTo
agent name	speak names

(L)$AgeSP_t$

Manifest	Protocol
manifest name	Protocol descr.

(L)MP_t

Artifact	Manifest
artifact name	manifest names

(L)$ArtM_t$

Agent	Manifest
agent name	manifest names

(L)AgeM$_t$

LinkedTo	Protocol
linked name	Protocol descr.

(L)LP$_t$

Artifact	LinkedTo
artifact name	linked names

(L)ArtL$_t$

Le *Topological Tables*, output del Design dei Workspaces, descrivono la struttura dell'ambiente del sistema:

- Workspace t ((L)W$_t$) descrive ogni singolo workspace;

- Workspace-Connection t ((L)WC$_t$) riporta le connessioni tra workspace;

- Workspace-Artifact t ((L)WArt$_t$) specifica l'allocazione dei singoli artefatti all'interno dei workspace;

- Workspace-Agent t ((L)WAge$_t$) elenca per ogni agente i workspace che può percepire.

Workspace	Description
workspace name	description

$(L)W_t$

Workspace	Connection
workspace name	workspace names

$(L)WC_t$

Workspace	Artifact
workspace name	artifact names

$(L)WArt_t$

Agent	Workspace
agent name	Workspace names

$(L)WA_t$

2.7.8 RELAZIONI TRA LE TABELLE PRODOTTE NEL DESIGN DI DETTAGLIO E GLI ELEMENTI DEL META-MODELLO

Prendendo in esame la figura 2.31, vediamo le relazioni tra le tabelle coinvolte nella fase di Design di Dettaglio con gli elementi del meta-modello.

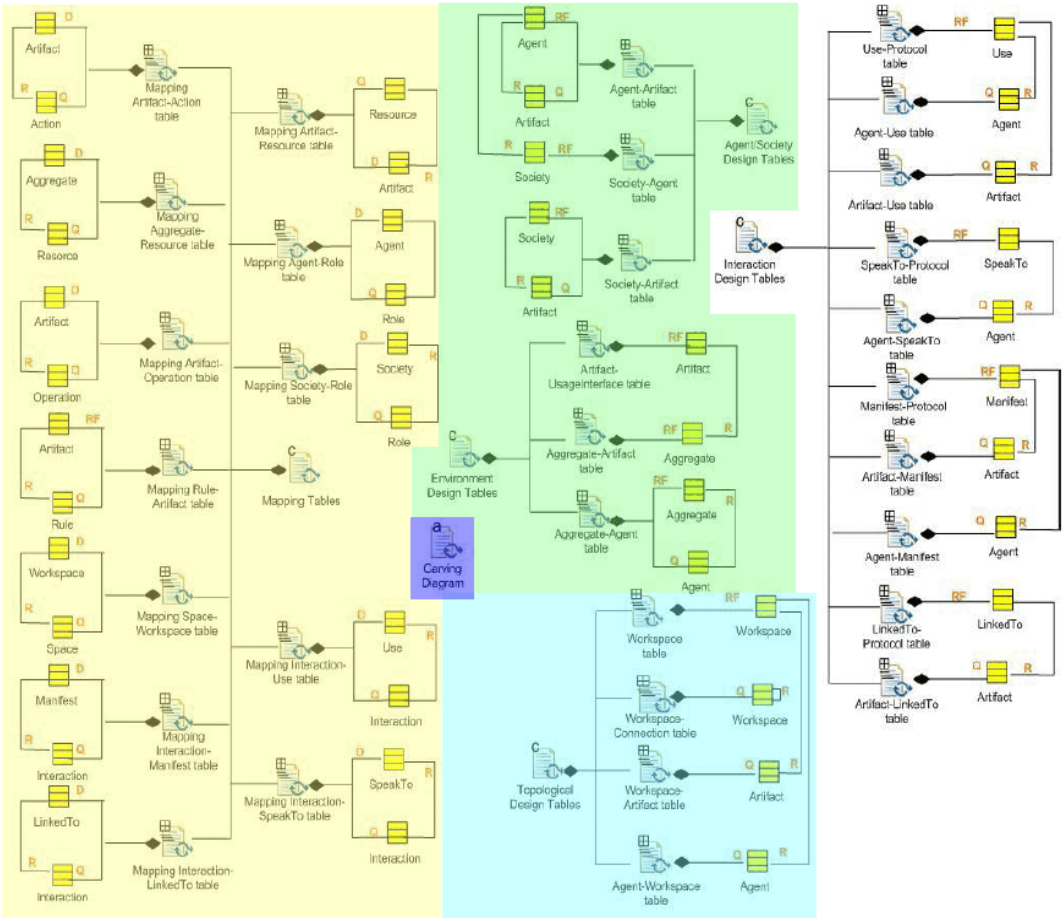


Figura 2.31 Relazioni tra work products del Design di Dettaglio ed elementi del meta-modello

Primo elemento da mettere in rilievo è il *diagramma di Carving* che non appare legato a nessun altro elemento; questo perché non modifica nessuna entità, ma estrae dal modello "oggetti" già definiti.

Per quanto riguarda le *Mapping Tables*, per prima cosa vediamo che la M Artifact-Rule table ha un'associazione di tipo RF con Artifact (in quanto gli artefatti vengono definiti dalle tabelle Mapping Artifact-Action, M Artifact-Operation e M Artifact-Resource) e di tipo Q con Rule; i due elementi sono in relazione tra loro. Tutte le altre tabelle di questa famiglia hanno un legame di definizione con un elemento; le due categorie coinvolte sono poi in relazione tra loro. In particolare le corrispondenze sono:

- Mapping Agent-Role table: (D) Agent, (Q) Role;
- Mapping Society-Role t: (D) Society, (Q) Role;
- Mapping Artifact-Action t: (D) Artifact, (Q) Action;
- Mapping Artifact-Resource t: (D) Artifact, (Q) Resource;
- Mapping Aggregate-Resource t: (D) Aggregate, (Q) Resource;
- Mapping Artifact-Operation t: (D) Artifact, (Q) Operation;
- Mapping Space-Workspace t : (D) Space, (Q) Workspace;
- Mapping Interaction-Use t : (D) Use, (Q) Interaction;
- Mapping Interaction-Manifest t: (D) Manifest, (Q) Interaction;
- Mapping Interaction-SpeakTo t: (D) SpeakTo, (Q) Interaction;
- Mapping Interaction-LinkedTo t: (D) LinkedTo, (Q) Interaction.

Si noti come tutte le entità dei diversi elementi vengano definite da queste tabelle, quindi durante l'attività di Mapping. Questo sottolinea ulteriormente la centralità di questa operazione nel processo di Design di Dettaglio, infatti osservando le associazioni degli altri insiemi di tabelle si può verificare che non vi sono altre definizioni di entità, ma solo riutilizzo, rifinitura e relazione.

Vediamo le Agent-Society Design Tables:

- Society-Artifact t: (RF) Society, poiché questa tabella viene utilizzata per stabilire l'appartenenza degli artefatti alle società e implica la modifica della struttura della società stessa.
  Per quanto riguarda Artifact il legame è di tipo (Q) , mentre i due elementi sono in relazione tra loro;
- Agent-Artifact : (RF) Agent, (Q) Artifact, (R) Artifact-Agent;
- Society-Agent t: (RF) Society, (R) Society-Agent.

Simili sono le relazioni che coinvolgono le *Environment Design Tables*: Artifact-UsageInterface table e Aggregate-Artifact table sono associate in modo (RF) rispettivamente ad Artifact e Aggregate, che sono a loro volta in relazione (R); Aggregate-Agent tale invece ha un legame (RF) con Aggregate e di tipo (Q) con Agent, in quanto questa tabella consente di definire quali agenti appartengano ad ogni aggregato, quindi di modificarne la struttura (da cui l'associazione di tipo F). Agent e Aggregate sono anche in relazione tra loro.
Analizziamo i legami delle *Topological Design Tables*:

- Workspace table è in relazione (R) con Workspace e lo può rifinire (F);

- Workspace-Connection table riutilizza Workspace, che è in relazione con se stesso;

- Workspace-Artifact table quota Artifact, che è in relazione con Workspace della prima tabella;

- Agent-Workspace table riusa Agent, in relazione a Workspace.

Le associazioni che coinvolgono le *Interaction Design Tables* sono varie:

- Use-Protocol: (RF) Use;

- Agent-Use: (Q) Agent, (R) Agent/Use (della tabella precedente);

- Artifact-Use: (Q) Artifact, (R) Agent/Use (della prima tabella);

- SpeakTo-Protocol: (RF) SpeakTo;

- Agent-SpeakTo: (Q) Agent, (R) Agent/SpeakTo (della tabella precedente);

- Manifest-Protocol: (RF) Manifest;

- Agent-Manifest: (Q) Agent, (R) Agent/Manifest (della tabella precedente);

- Artifact-Manifest: (Q) Artifact, (R) Artifact/Manifest (della tabella Manifest-Protocol);

- LinkedTo-Protocol: (RF) LinkedTo;

- Artifact-LinkedTo: (Q) Artifact, (R) Artifact/LinkedTo (della Artifact-inkedTo tabella precedente).

Queste ultime relazioni sono in linea con la natura delle Interaction Design Tables, infatti definiscono i tipi di interazioni che coinvolgono di volta in volta agenti o artefatti, o entrambi, quindi questi elementi sono riutilizzati, mentre le entità dei tipi corrispondenti alle comunicazioni vengono rifinite.

2.7.9 DIPENDENZE TRA WORK-PRODUCTS

In Figura 2.32 sono riportati tutti gli insiemi di tabelle derivanti dall'esecuzione delle fasi del processo SODA, ponendo in risalto le relazioni di dipendenza che derivano dalla sequenza delle diverse attività.

Da questo schema risalta il ruolo delle tabelle che scaturiscono dalle attività di transizione tra una fase e l'altra: References, Transitions e Mappings Tables. Appare infatti evidente che queste tabelle dipendono da tutti gli insiemi prodotti dalla fase che li precede, ma allo stesso tempo tutti i set di tabelle della fase corrente (cioè quella a cui gli insiemi di transizione appartengono) dipendono dall'attività di passaggio.

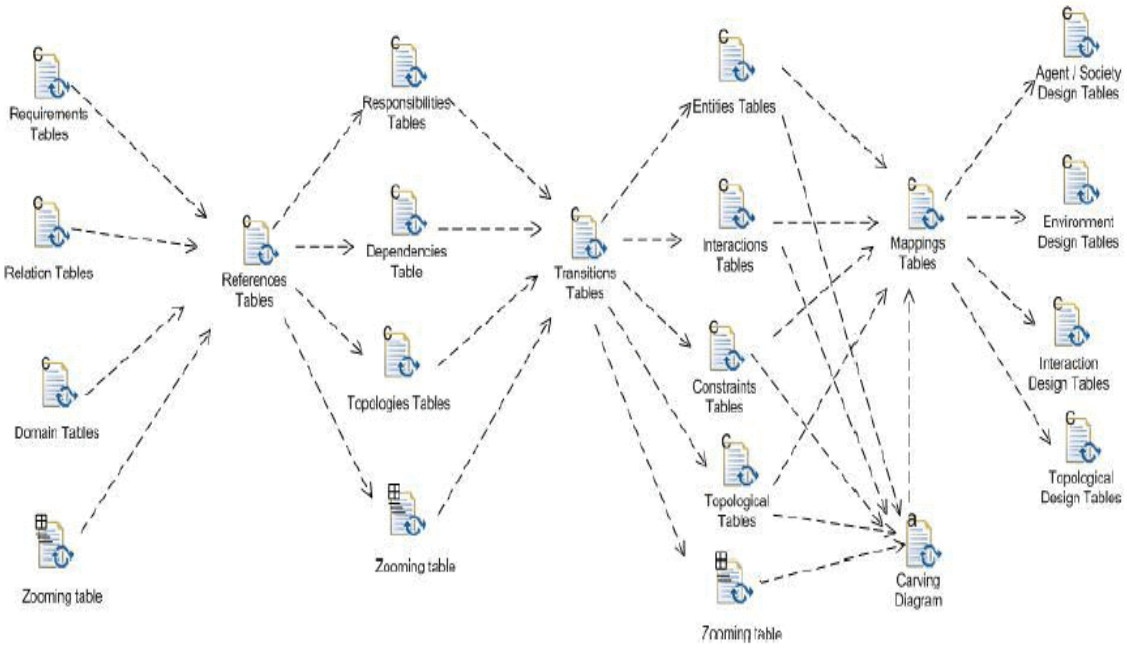


Figura 2.32 Relazioni tra work-products

Scendendo nel dettaglio si ha che le References Tables dipendono da: Requirements, Elements e Domain Tables, oltre che dalla Zooming table, prodotte dall'Analisi dei Requisiti.

Dalle References Tables derivano a loro volta Responsibilities, Topologies e Dependencies Tables, più la Zooming table, risultanti dall‘Analisi.
Da queste dipendono le Transitions Tables, che a loro volta precedono: Entities, Interactions, Constraints e Topological Tables, unite alla Zooming table, derivate dal processo di Design Architetturale.
Le dipendenze che ne seguono sono quella delle Mapping Tables e del Carving Diagram, da cui dipendono le stesse Mapping Tables. Da queste ultime dipendono le tabelle generate dal Design di Dettaglio: Agent/Societies Design, Environment Design, Topological Design e Interaction Design Tables.

3 Check&SODA: CONTROLLO E BUSINESS LOGIC.

Nell'ambito di sistemi software, una delle esigenze è quella di ottenere sistemi veloci, versatili, e facilmente adattabili ai requisiti applicativi; per questo motivo il campo della ricerca studia nuove metodologie atte a creare software per organizzare sistemi sempre più complessi.
Le nuove metodologie devono consentire di sviluppare in modo organizzato le strutture software, che sono richieste avere caratteristiche sempre più dinamiche.

La metodologia SODA (Capitolo 2), si sviluppa attraverso passi precisi, ciascuno dei quali genera tabelle per la descrizione di tutte le entità appartenenti al sistema e delle relazioni che intercorrono tra esse.

In questo capitolo, saranno affrontate:

a) le logiche di controllo del processo di sviluppo SODA.
b) le verifiche dell'integrità dei dati e della documentazione finale prodotta dal modulo Check&SODA.

Il punto di partenza è l'analisi del meta-modello su cui si basa il processo SODA (Sezione 3.1), verrà poi definita una grammatica formale (Sezione 3.2) e successivamente ne verrà illustrato un esempio di utilizzo, da parte delle classi che costituiscono l'unità di controllo (Sezione 3.3). Verranno poi indicate le funzionalità che saranno implementate (Sezione 3.4) ed il modo in cui queste saranno sviluppate nella logica di verifica che viene evidenziata con l'ausilio di diagrammi di flow chart (Sezione 3.5), mentre per la descrizione dell'architettura delle classi di controllo, si fa uso di diagrammi UML (Sezione 3.6).

3.1 ANALISI DEL META-MODELLO SODA

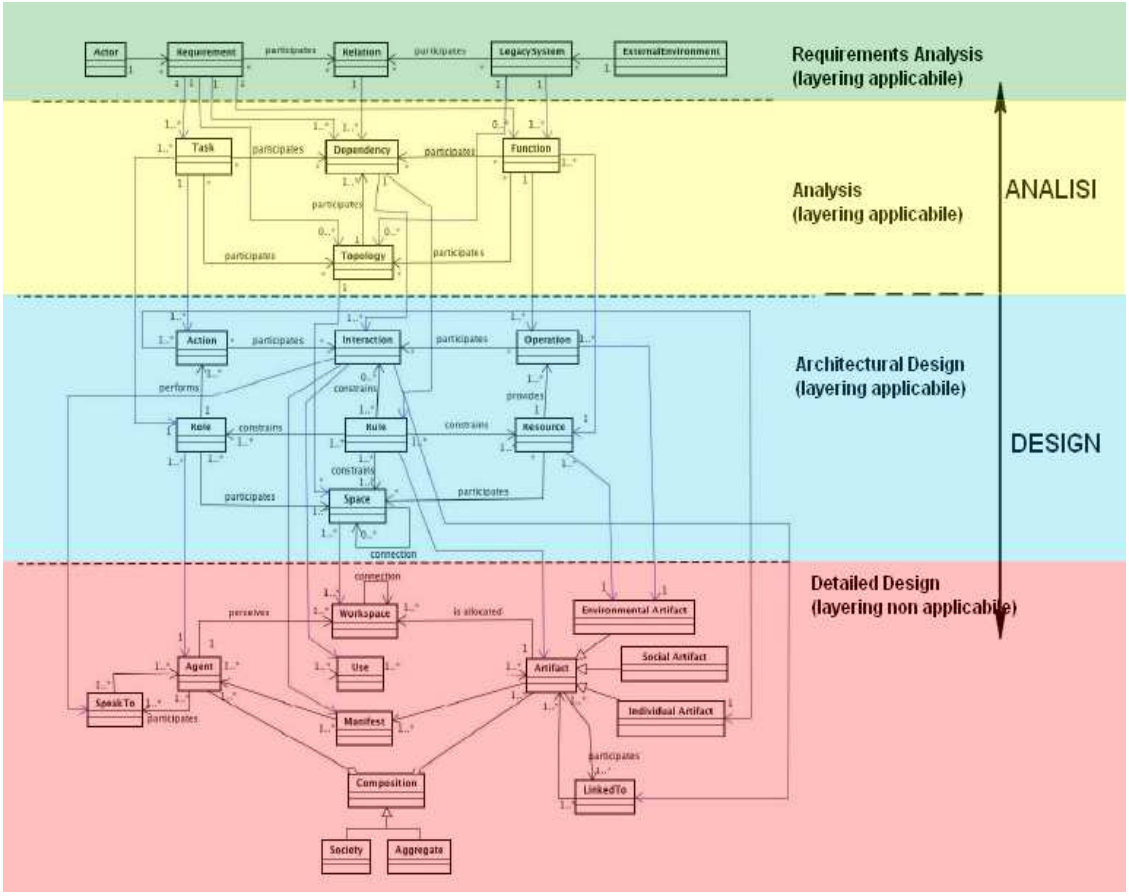


Figura 3.1 Meta-modello delle astrazioni SODA.

Riprendendo lo schema raffigurante il meta-modello SODA (Figura 3.1), analizziamo la metodologia, che sviluppandosi su quattro fasi, individua per ciascuna di esse le entità e le relazioni che servono a definire il sistema multi-agente finale. Di seguito sono riportate le relazioni principali che riguardano gli elementi del meta modello , precedentemente indicati nella tesi relativa al modello grafico (Graph&SODA) [7], qui ripresi e dettagliati.

ANALISI DEI REQUISITI:

Figura 3.2:

- Un *Actor* può essere associato a zero o più *Requirement*.
- *ExternalEnviroment* può essere legato a zero o più *LegacySystem.*
- Un *Requirement* può partecipare a zero o più *Relation* e lo stesso vale per *LegacySystem*.
- Relation può mettere in corrispondeza:

- più entità di tipo *Requirement.*
- più elementi *LegacySystem.*
- *Requirement e LagcySystem.*

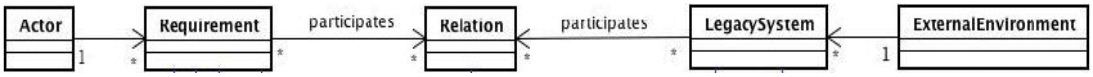


Figura 3.2 Meta-modello dell'Analisi dei Requisiti (fase 1).

Il passaggio tra l'Analisi dei Requisiti e Analisi prevede i seguenti legami (Figura 3.3):

- Ogni *Requirement* può partecipare alla definizione di:

  - Uno o più *Task.*
  - Una o più *Dependency*
  - Zero o più *Functions.*
  - Zero o più *Totpology.*

- Una *Relation* può essere tradotta in una o più *Dependency.*
- Ogni *LagacySystem* può essere associato a:

  - Una o più *Function.*
  - Zero o più *Totpology.*

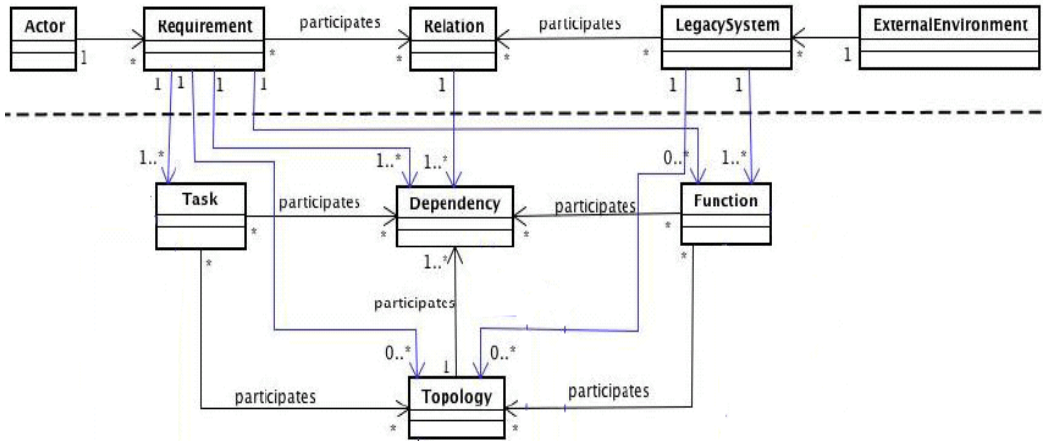


Figura 3.3 Rapporto Prima e seconda fase SODA.

ANALISI:

Si hanno le associazioni (Figura 3.4):

- Una *Dependency* consente la partecipazione di zero o più *Task* e zero o più *Function.*
- Un *Task* o una *Function* possono partecipare a zero o più *Dependency*.
- Per *Topology* valgono gli stessi legami che ricorrono per *Dependency.*
- Una *Topology* può caratterizzare una o più *Dependency.*

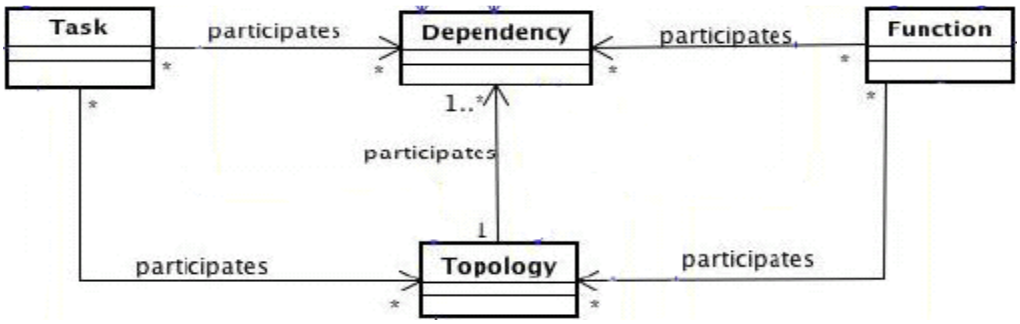


Figura 3.4 Meta-modello dell'Analisi (fase 2).

Le seguenti relazioni sono invece presenti passando dall'Analisi al Design Architetturale (Figura 3.5):

- Da un *Task* si possono definire uno o più *Action*.
- Un *Role* può essere specificato da uno o più Task.
- Una *Dependency* viene mappata in:

  - Una o più *Iteraction.*
  - Una o più *Rule.*
  - Una o più *Function* è legata ad una o più *Operation*.
  - Una o più *Function* definiscono una *Resource.*
  - Da una *Topology* vengono definiti zero o più *Space*.

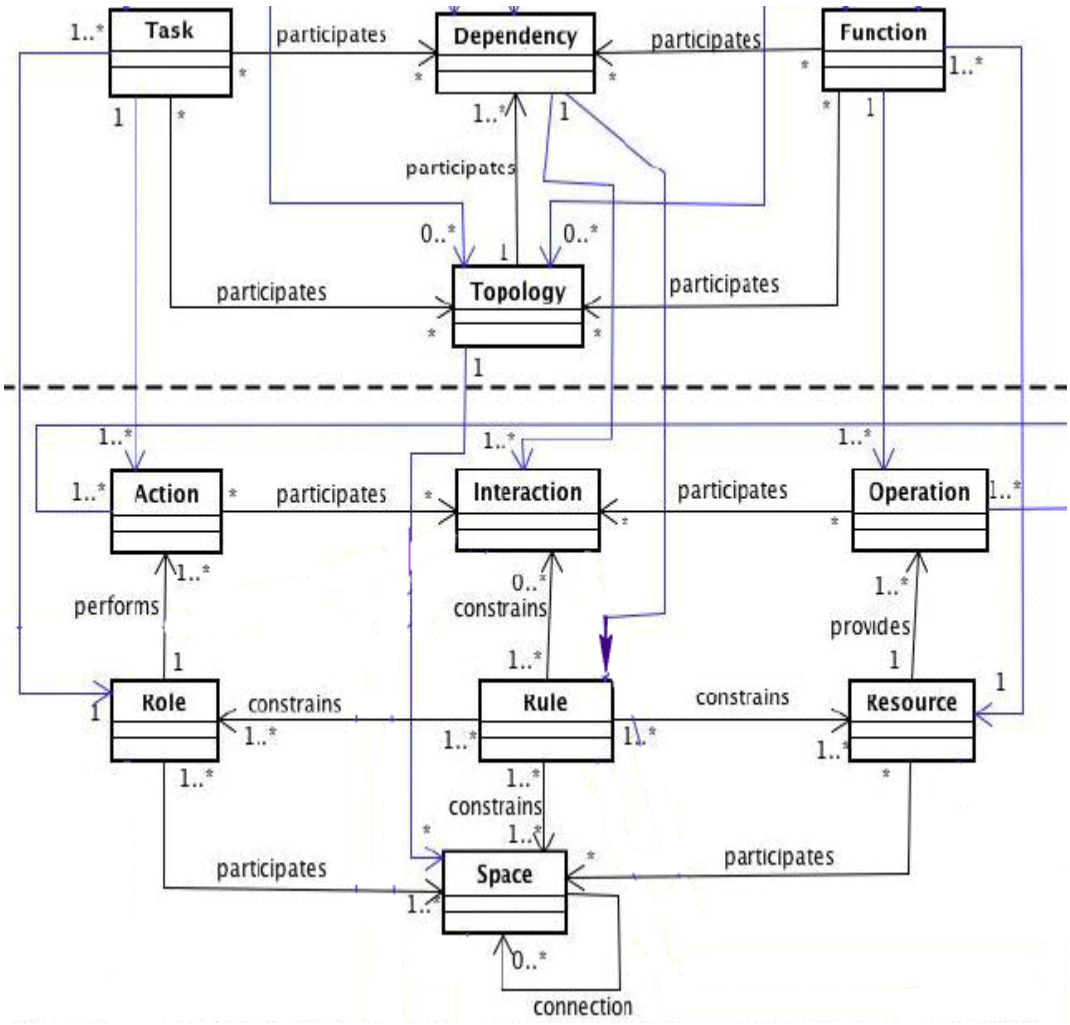


Figura 3.5 Rapporto seconda e terza fase SODA.

DESIGN ARCHITETTURALE:

Caratterizzata da varie associazioni (Figura 3.6):

- *Action* e *Operation* possono partecipare alla definizione di zero o più *Interaction;*

- *Interaction* può essere legata a zero o più *Action* e zero o più *Operation*;

- Uno o più *Role* possono partecipare alla definizione di uno o più *Space*;

- Zero o più *Resource* possono partecipare alla specifica di zero o più Space;

- Uno *Space* è legato sempre ad almeno un *Role*, ma non necessariamente ad una *Resource*

- Ogni *Space* può essere connesso con zero o più elementi dello stesso tipo;

- Un *Role* compie una o più *Action*;

- Una *Resource* fornisce una o più *Operation*;

- Una o più *Rule* possono vincolare:

  - Zero o più *Interaction*;
  - Uno o più *Role*;
  - Uno o più *Space*;
  - Una o più *Resource*.

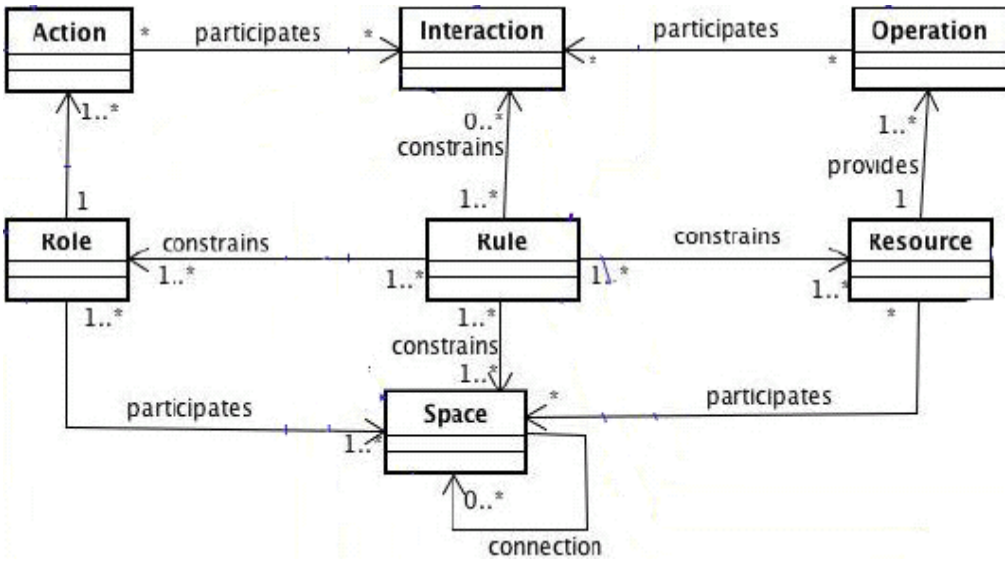


Figura 3.6 Meta-modello del Design architetturale (fase 3).

La mappatura degli elementi dalla fase di Design Architetturale a quella di Design di Dettaglio è realizzata in base alle seguenti relazioni (Figura 3.7):

- Da uno o più *Space* possono essere definiti uno o più *Workspace*;
- Uno o più *Role* sono associati a un *Agent*;
- Una *Rule* è associata a un *Artifact*;
- Una o più *Resource* e *Operation* vanno a specificare un *EnvironmentalArtifact*;
- Una o più A*ction* definiscono un *IndividualArtifact*;

- *Interaction* può essere specificata in uno dei diversi tipi di connessione tra agenti e/o artefatti: *SpeakTo, LinkedTo, Use, Manifest*.

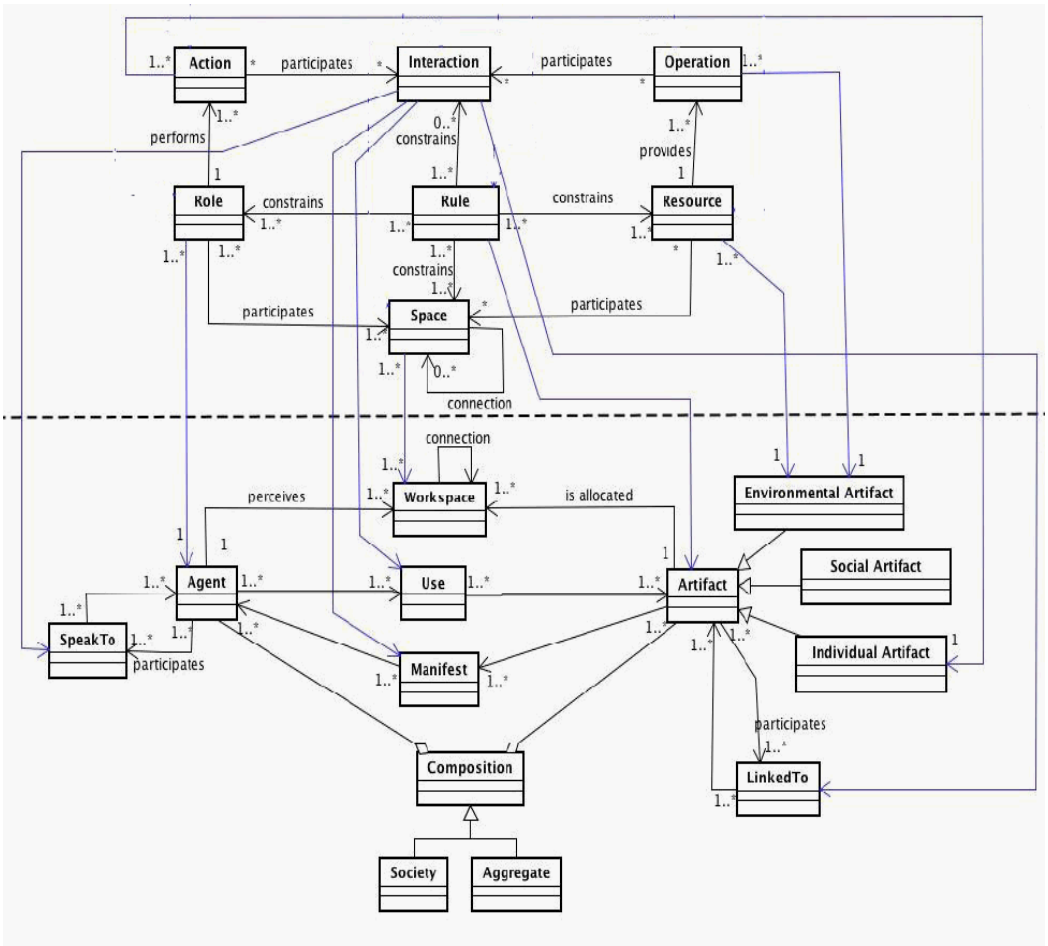


Figura 3.7 Rapporto terza e quarta fase SODA.

DESIGN DI DETTAGLIO:

Nello step conclusivo del processo, il Design di Dettaglio, si distinguono alcune associazioni particolari (Figure 3.8):

- Un elemento di tipo *Workpsace* può essere connesso a zero o più entità dello stesso tipo;
- Possono essere presenti delle *Composition* (specificate in *Society* o *Aggregate*) e ognuna può contenere zero o più *Agent* e zero o più *Artifact*;
- Un *Agent* percepisce uno o più *Workspace*;
- Un *Artifact* è allocato in uno o più *Workspace*;
- Un *Artifact* può essere di tipo *Individual*, *Social*, *Environmental*;
- Uno o più *Agent* partecipano a una o più interazioni *SpeakTo*;

- Uno o più *Artifact* partecipano alle interazioni *LinkedTo* (una o più);

- *SpeakTo* e *LinkedTo* devono mettere in corrispondenza almeno due entità, rispettivamente di tipo *Agent* e *Artifact*;

- Uno o più *Agent* possono essere connessi da interazioni di tipo *Use* a uno o più *Artifact*; - Uno o più *Artifact* possono essere associati da *Manifest* a uno o più *Agent*.

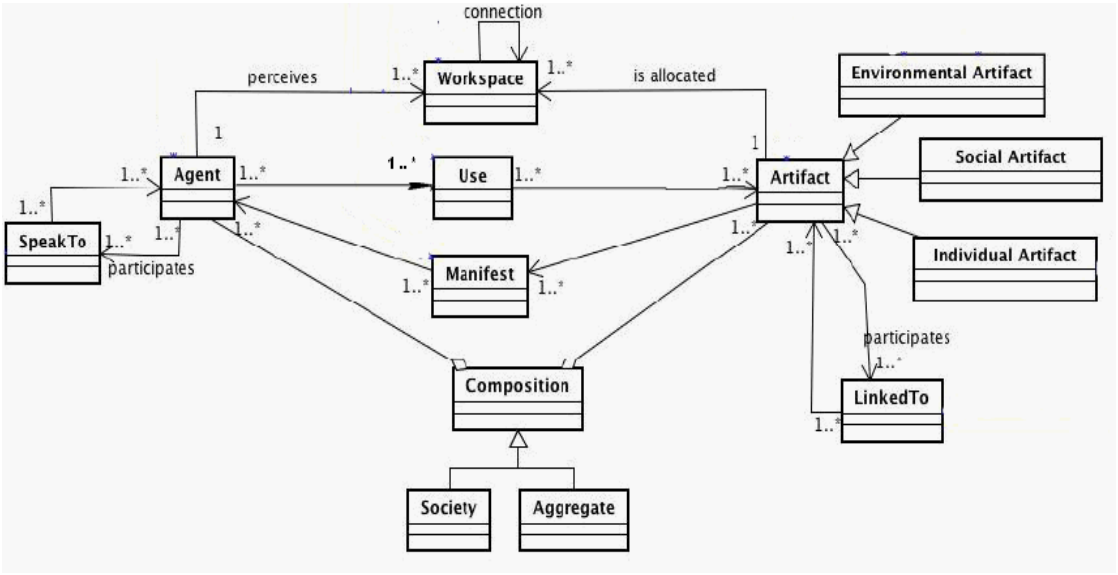


Figura 3.8 Meta-modello del Design di dettaglio (fase 4).

3.2 GRAMMATICA

Il linguaggio SODA è un linguaggio di tipo grafico, e per la sua verifica occorre definire una grammatica che ne imposti le regole, per i controlli e i vincoli da rispettare in fase di progettazione; per tale motivo è stata ripresa e ri-elaborata la grammatica relativa al linguaggio grafico precedentemente sviluppata [7].
Tale grammatica, è stata rivista e raffinata, per poter essere utilizzata dal modulo di controllo al fine di verificare sia, l'integrità dei dati sia la produzione dei documenti finali, delle fasi SODA.

A livello pratico, l'implementazione delle regole grammaticali, è stata definita mediante un XML Schema Definition (XSD) (Figura 3.9), di cui si fa uso per gestire più agevolmente le relazioni di una struttura a oggetti (xbean), che verrà generata automaticamente con l'ausilio del framework Castor. Sono le relazione tra gli oggetti xbean, che permettono alle classi di check, di capire quali oggetti (entità) esistono in una particolare fase del progetto SODA e di sapere quali oggetti (entità/relazioni) attendersi nelle migrazioni di fase.

Nella descrizione che segue, viene riportato in dettaglio la prima fase SODA, mentre per le altre fasi si può fare riferimento all'appendice A, in quanto la logica non cambia ma solo le entità e le relazioni in gioco.

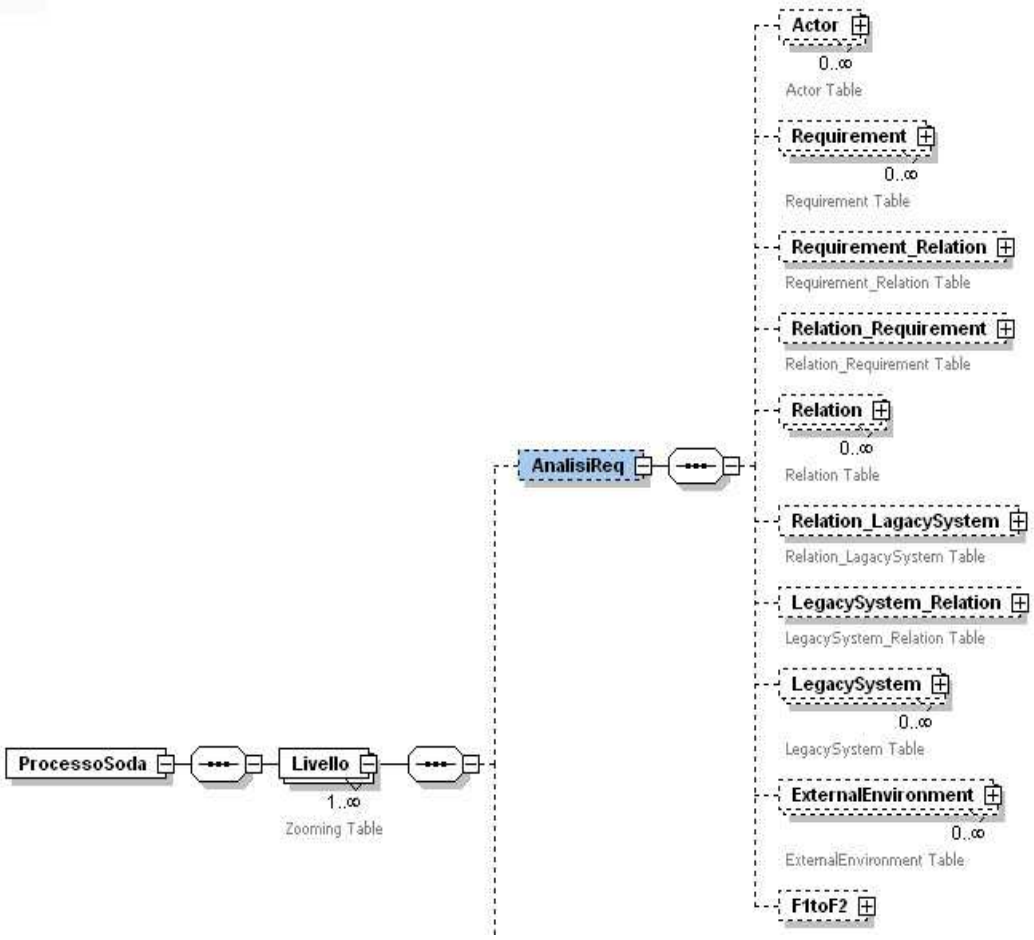


Figura 3.9 Struttura parziale della grammatica riferita alla prima fase SODA (analisi dei requisiti).

E' semplice capire a livello visivo, come gli oggetti xbean, vengono legati per identificare le entità della prima fase SODA. Dal grafico si evince che la prima fase SODA "AnalisiReq", che può contenere più entità "Actor", "Requirement", "Relation", "LegacySystem" ed "ExternalEnvironmet", e più relazioni di fase e migrazione, a sua volta sia replicabile all'interno dell'oggetto "Livello", che viene ad assumere il ruolo della Zooming table.
Le classi di controllo, navigano la struttura mediante l'utilizzo di parametri, che vedremo definiti successivamente come parametri di input.

Più in dettaglio in figura 3.10, viene riportata l'indicazione degli attributi che caratterizzano l'entità "AnalisiReq".

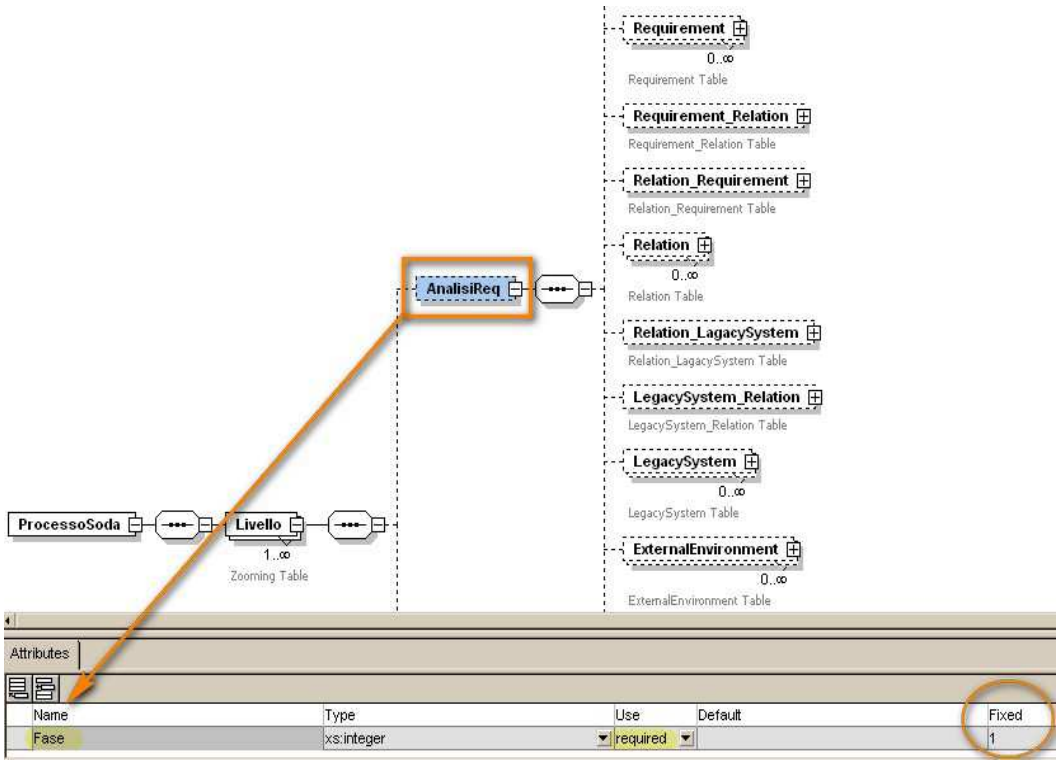


Figura 3.10 Dettaglio della prima fase SODA. Indicazione degli attributi.

Come già detto l'elemento "AnalisiReq" rappresenta un oggetto (xbean), quindi controllabile dalle classi "CheckSODA". L'indicazione della fase viene fornita in modo univoco con l'attributo "Fase", il quale assume un valore fisso e progressivo per ciascuna fase successiva alla prima.

La rappresentazione degli attributi di zooming, si vede in figura 3.11.

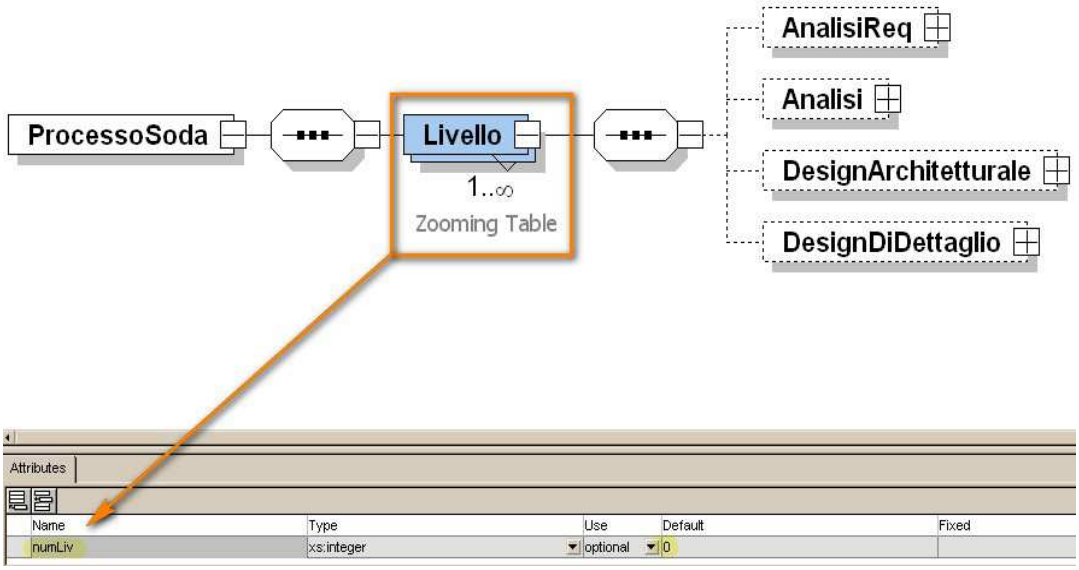


Figura 3.11 Dettaglio attributi della Zooming Table.

All'interno di ciascun livello, che di fatto rappresenta la Zooming Table, sono contenute le quattro fasi SODA con le rispettive entità e relazioni di fase e di migrazione, sempre rappresentate con oggetti xbean.

La corrispondente relazione nel modello grafico (GMF), della Zooming table, è rappresentata da una EClass che risulta l'aggregazione delle altre classi di fase.

Rappresentazione dettagliata di alcune entità della prima fase SODA (Figura 3.12).

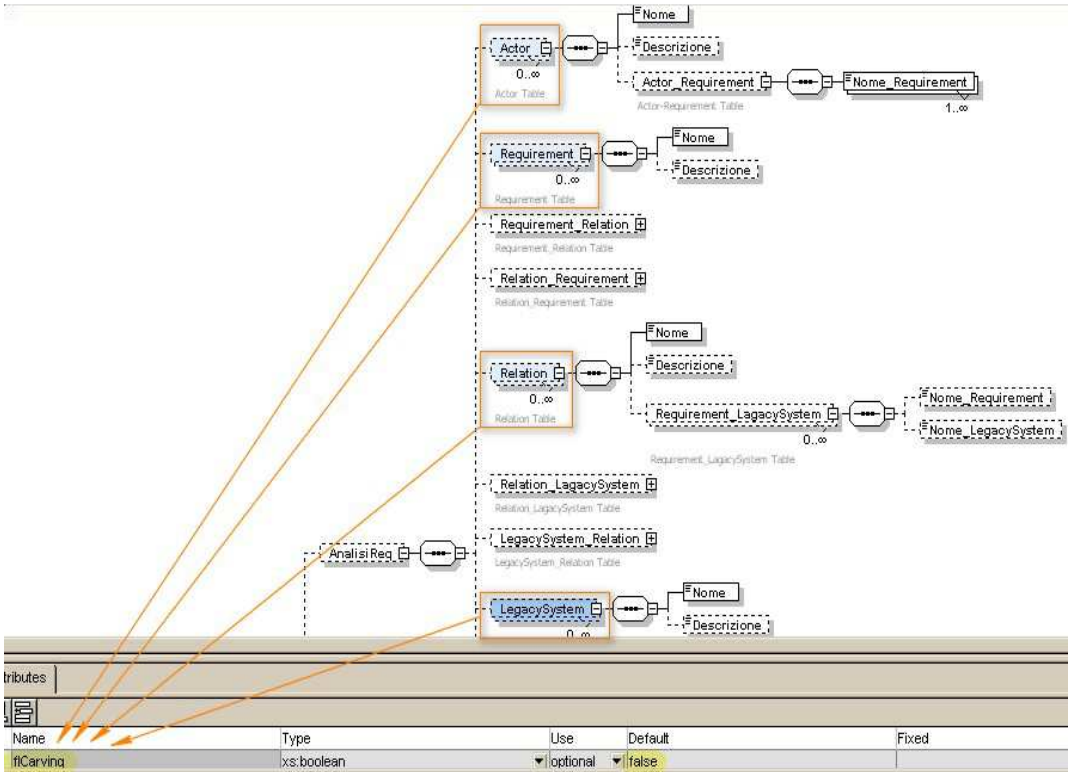


Figura 3.12 Dettaglio entità della fase. Attributo per la gestione del carving.

Come si vede le entità sono array di tabelle, infatti si possono inserire a livello grafico, per ciascuna fase, più entità dello stesso tipo contenenti i campi descrittivi “nome” e “descrizione”, più le relazioni che vedremo meglio di seguito. L’attributo “flCarving” delle entità di fase, serve a permettere la gestione della funzionalità di Carving, che sarà applicabile solamente nella quarta fase del processo SODA.

A questo punto, nel caso in cui i controlli del modulo di check vadano a buon fine, si ottengono le descrizioni ed i contenuti informativi, delle entità e relazioni, relative alla varie fasi SODA, sotto forma di file XML; divise per livelli (zooming) con l’attributo di carving per l’individuazione del modello ad agenti finale.

E’ possibile applicare XQuery per la manipolazione delle informazioni finali, da parte di altri tool.

E’ possibile il trasferimento degli XML ad altri sistemi.

Con le medesime logiche appena definite, vediamo di descrivere altri aspetti ed elementi che caratterizzano la prima fase SODA, figura 3.13.

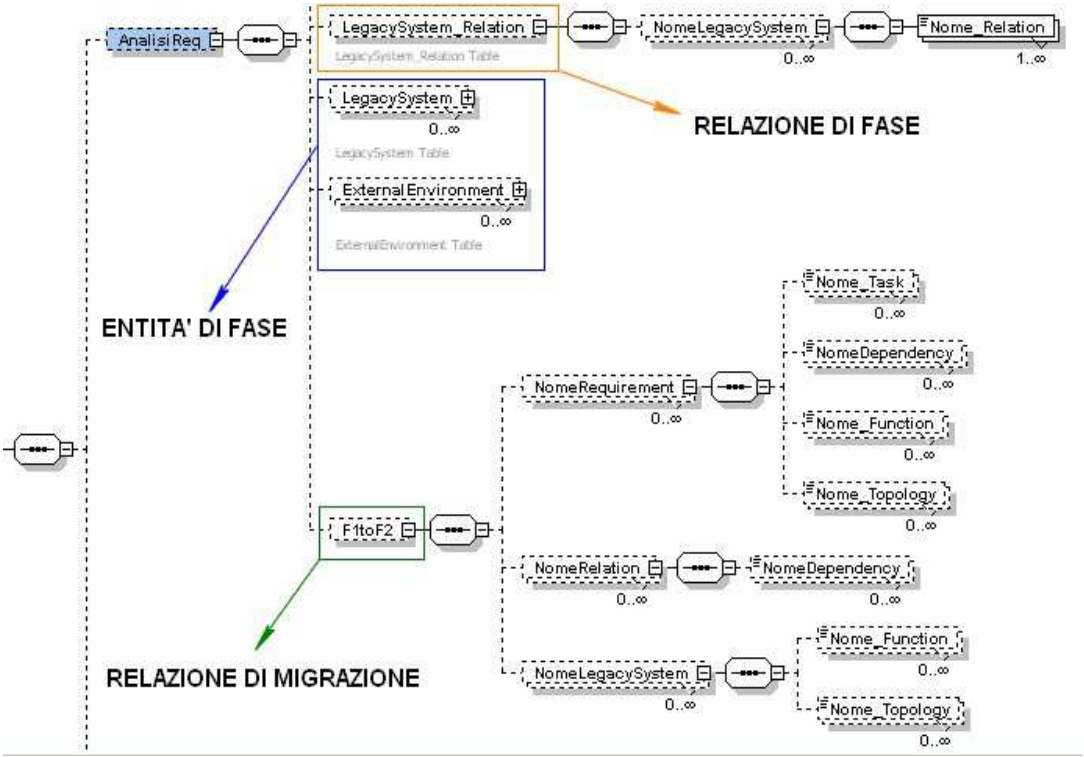


Figura 3.13 Struttura della fase di Analisi dei Requisiti (prima fase).

Dal dettaglio della figura, è possibile distinguere la tipologia degli elementi base che vanno a formare una fase SODA. A parte il tipo di oggetto "*entità di fase*" che è già stato descritto, abbiamo anche gli oggetti "*relazione di fase*" e "*relazione di migrazione*".

Le "*relazioni di fase*" rappresentano i legami di entità, tutte all'interno di ciascuna fase SODA, mentre le "*relazioni di migrazione*", identificano le relazioni tra le entità di due fasi SODA consecutive.

Per agevolare meglio la navigazione delle struttura grammaticale, sia le "relazioni di fase" che quelle di "migrazione" sono state collocate nel medesimo ramo delle entità di fase, fatta eccezione solo per alcune singole relazioni del tipo uno-a-molti, che sono ubicate direttamente all'interno delle entità di fase. Questa soluzione è il risultato ibrido, di una struttura che inizialmente era stata progettata con le relazioni tutte all'esterno delle entità di fase, poi in un'altra versione, con tutte le entità di relazione all'esterno delle entità di fase. Il risultato ibrido di questa struttura, sembra essere meglio rispondente alle necessità di agevolare la navigazione della grammatica, ai fini di gestire meglio l'integrità referenziale, che le classi di controllo devono garantire.

Il dettaglio della restante grammatica, viene riportato nell'appendice A.

3.3 IMPIEGO DELLA GRAMMATICA

In questo paragrafo, viene data una spiegazione più precisa del modo in cui le regole grammaticali, definite in precedenza, vengano implementate dal modulo di controllo Chek&SODA.

La soluzione proposta permette di creare una struttura ad oggetti tramite xbean, attraverso la quale le classi di controllo, muovendosi per le fasi SODA, sanno quali entità aspettarsi e quindi verificarne le caratteristiche, grazie agli oggetti trovati.

Il coincidere degli oggetti e delle loro caratteristiche, con le entità inserite attraverso i moduli di progettazione, vale a dire quello grafico e/o quello tabellare, permette di discriminare la correttezza dei dati in base alla grammatica.

Se l'esito dei controlli risulta positivo, la medesima struttura ad oggetti, può immagazzinare i dati che rappresentano le entità SODA. In caso di esito negativo, viene invece generata un'eccezione, con lo scopo di avvisare il progettista dell'errata operazione che sta eseguendo.

Da quanto decritto, è possibile evidenziare il doppio ruolo che ha la struttura degli xbean:

- rappresentazione delle regole grammaticali.
- contenitore per la memorizzazione delle informazioni che descrivono le entità SODA.

Come passo finale, una volta popolati gli oggetti della struttura grammaticale, attraverso la funzionalità di unmarshal, è facile ottenere il documento XML finale aggiornato, che grazie al suo formato, risulta facilmente elaborabile per le funzionalità di export o per le funzionalità relative agli altri moduli.
E' importante evidenziare, che le operazioni di inserimento o cancellazione di ogni singola entità a livello grafico o tabellare, vengono intercettate immediatamente, attraverso eventi rilevabili dal modulo di controllo, mediante gli extension-point, che lo stesso modulo estende. E' evidente che l'aggiornamento della struttura grammaticale e di conseguenza del documento XML, va di pari passo alle azioni che il progettista compie durante l'attività di progetto.

Fino a questo punto è stata descritta la struttura degli xbean, come unico elemento della grammatica per le operazioni di verifica; questo in realtà non è completamente esatto.
Infatti il tipo di relazione tra le entità in gioco, relative alle singole fasi SODA, del tipo uno-a-uno, uno-a-molti e molti-a-molti e l'integrità referenziale tra le entità, sono gestite dalla logica che si trova nelle classi di controllo del modulo Check&SODA.

La comunicazione tra i moduli Kit&SODA, Graph&SODA e Check&SODA avviene attraverso eventi ed eccezioni (quindi in modo asincrono); i tre moduli esercitano le loro azioni di aggiornamento solo sulle strutture direttamente legate al framework di appartenenza. In questo modo si ottiene un disaccoppiamento dei moduli e quindi ad una loro eventuale sostituzione od estensione con altri differenti framework.

3.4 PROGETTAZIONE DELLE FUNZIONALITA' DI CONTROLLO

Le funzionalità principali di cui il modulo di controllo si deve occupare, durante l'attività di progetto, sono:

- Consistenza delle entità definibili per fase SODA, sia per le operazioni di inserimento/aggiornamento che di cancellazione.
- Consistenza delle entità di relazione, che devono legare le giuste entità all'interno e tra le fasi SODA.
- Corretta sequenza di attivazione delle fasi SODA, (non è possibile inserire entità nella seconda fase, quando la prima risulta completamente vuota).
- Gestione delle funzionalità di import e aggiornamento del documento XML.

Per quanto riguarda la funzionalità di export, il controllo non interviene, in quanto esiste già il documento XML, pronto per essere elaborato da qualsiasi altro sistema.

3.5 PROGETTAZIONE DELLA LOGICA DI CONTROLLO

Il modulo di Check&SODA, attraverso l'estensione di alcuni extension-point, si connette al modulo grafico ed a quello per la gestione delle tabelle. L'attivazione delle verifiche di controllo avviene tramite la generazione di eventi (action-event); quindi abbiamo una gestione asincrona delle chiamate, innescata dalle attività di progetto, che avvengono sul modulo grafico e/o su quello di gestione delle tabelle descrittive, ad opera del progettista.

Successivamente all'attivazione del modulo di controllo (gestione asincrona), si sviluppano in sequenza (gestione sincrona) le azioni di verifica sui dati.
Per dare una visione più chiara della logica che regola l'attività di controllo, si è fatto uso dei diagrammi di flow chart, di cui il primo (Figura 3.15), descrive in generale le azioni principali.

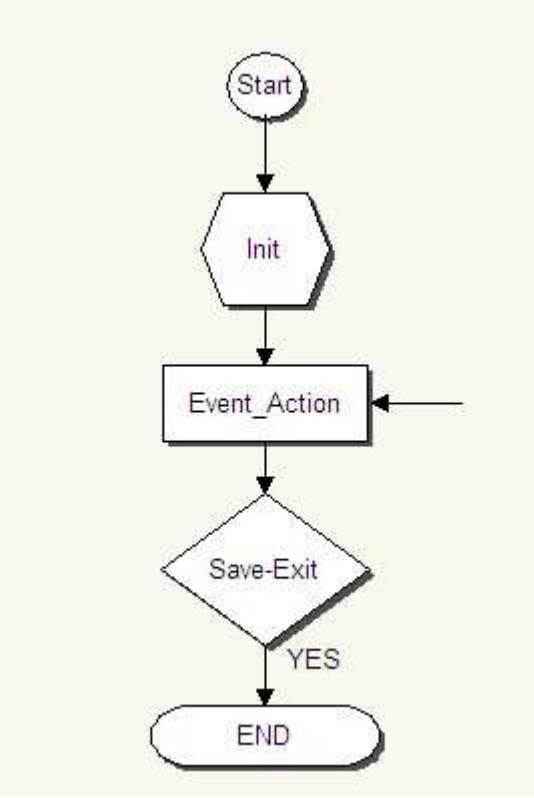


Figura 3.15. Elementi generali del processo di controllo.

L'attività di "INIT" viene richiamata una sola volta (attraverso l'istanzazione della classe "CheckSODA", la main-class del progetto Check&SODA), nel momento in cui si crea un nuovo progetto oppure nel momento in cui viene caricato il progetto esistente in Eclipse. La notifica della chiamata avviene tramite evento, ad esempio generato dall'extension-point "*editpartProviders*" di GMF, estendibile dalla classe

"*EditPartProviderExtension*", la quale a sua volta estendendo la classe "*AbstractEditPartProvider*" ed implementando l'interfaccia "*IElementSelectionProvider*", tramite il metodo "*addProviderChangeListener*" è possibile intercettare le azioni indicate in precedenza, quelle cioè di creazione di un nuovo progetto oppure quelle di apertura di un progetto già esistente.
Isolate le azioni che interessano l'attivazione della chiamata all'INIT, quest'ultima ha il compito di istanziare in memoria la struttura degli xbean ed impostare alcuni parametri utili per le verifiche sulle regole grammaticali, tipo l'HashMap per la verifica dell'univocità dei nomi delle entità e relazioni SODA, inserite.

Successiva all'INIT, l'attività "Event-Action" racchiude tutta la logica di controllo vera e propria, che viene invocata ad ogni evento di inserimento/cancellazione di un'entità o relazione.
Tutto il contesto di controllo ha termine al momento del salvataggio e uscita dal progetto.

3.5.1 DETTAGLIO DELL'ATTIVITA' DI "INIT"

Ora è possibile vedere con maggiore dettaglio le macro-attività precedentemente individuate (Figura 3.16).

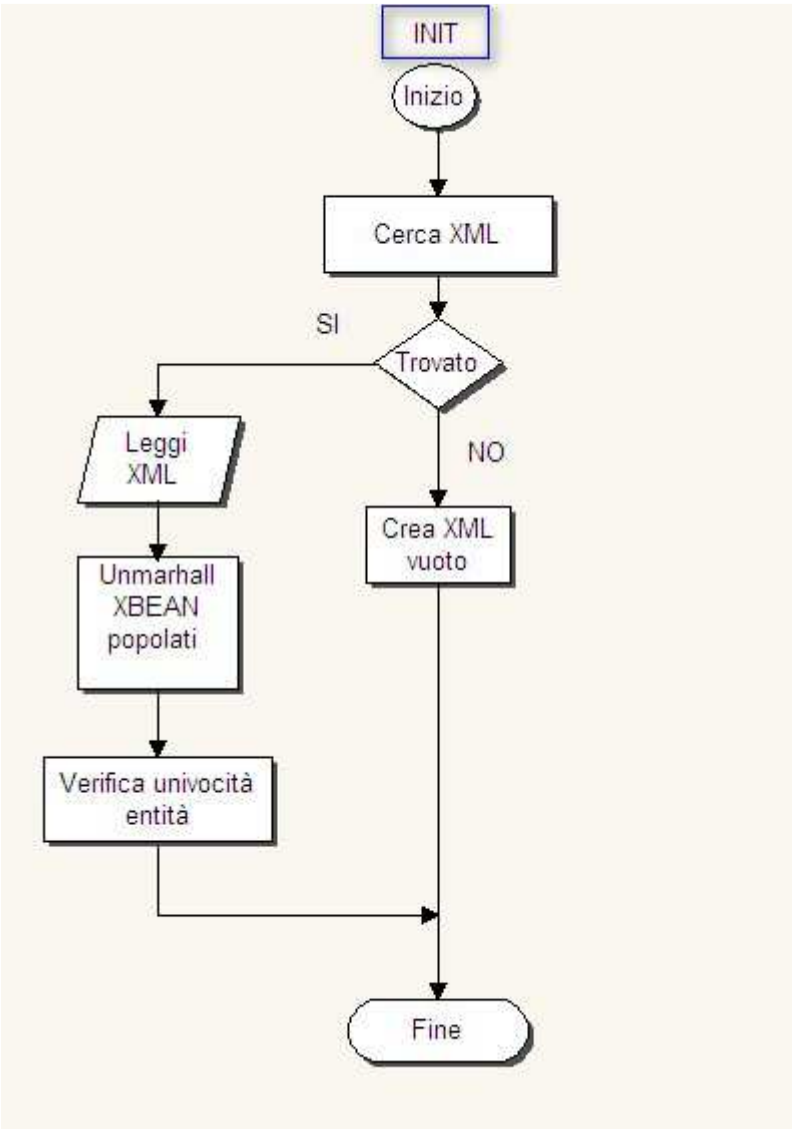


Figura 3.16. Dettaglio dell'attività di INIT.

Come si vede, l'attività di "INIT", si preoccupa di verificare l'esistenza del documento XML, che nel caso di un progetto già definito, ne implica l'esistenza, mentre se il progetto è stato appena creato, il documento XML non esiste ancora e ne viene creato il file vuoto.

Nel caso l'XML esista, questo viene letto e con i dati viene aggiornata la struttura degli xbean tramite unmarshal. Successivamente, con l'ausilio di XQuery, vengono lette le entità contenute nel file XML, allo scopo di riempire un HashMap, per le verifiche di univocità sui nomi delle entità che verranno inserite successivamente.

3.5.2 DETTAGLIO DELL'ATTIVITA' DI "EVENT-ACTION"

La seconda macro-attività la possiamo vedere scomposta di seguito (Figura 3.17).

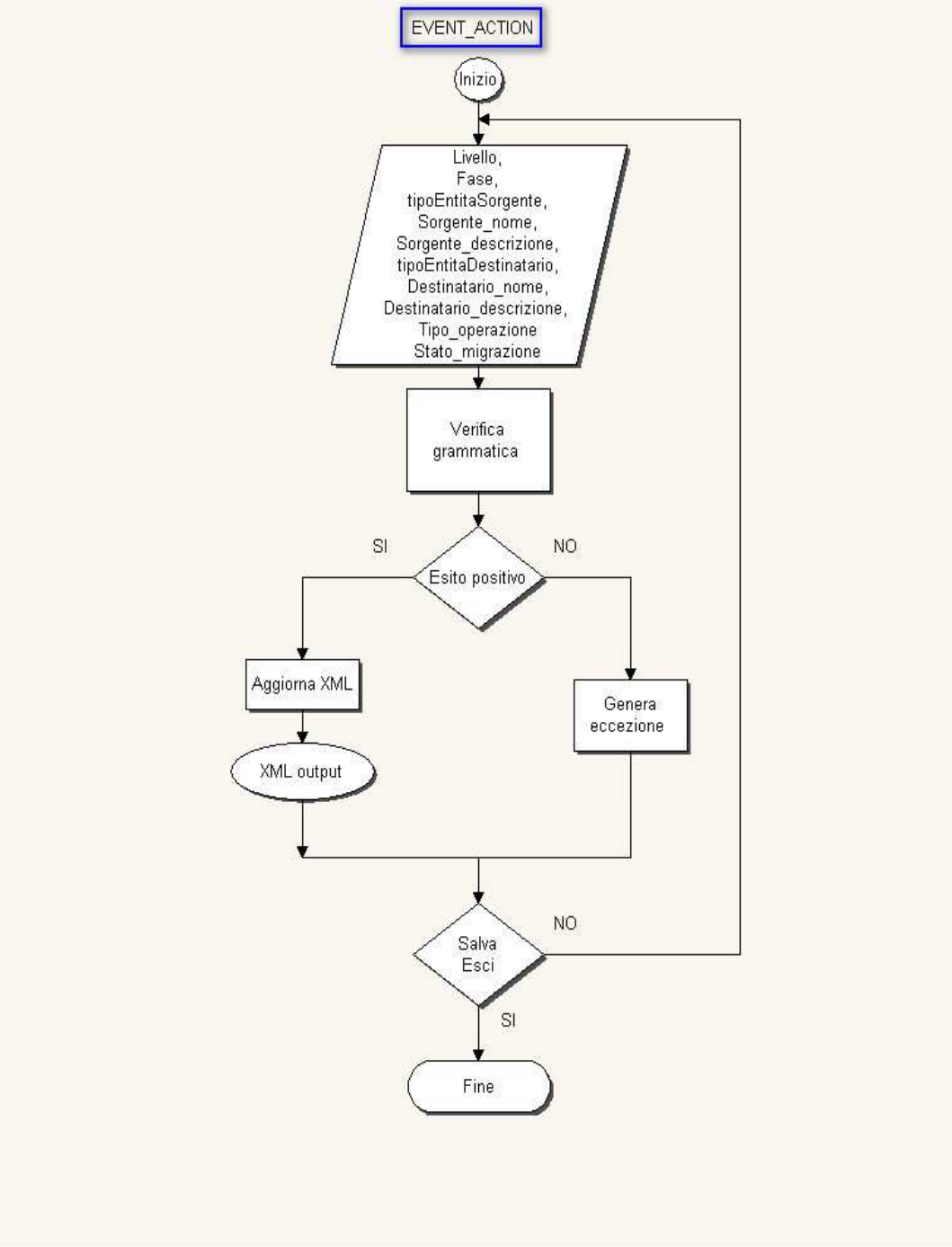


Figura 3.17. Dettaglio dell'attività di EVENT-ACTION.

A seguito di un evento che può essere generato dal modulo di Kit&SODA oppure dal modulo Graph&SODA, ogni volta che il progettista inserisce/cancella una nuova entità grafica oppure inserisce/cancella una nuova tabella e/o l'aggiorna, viene attivato il modulo Check&SODA. La chiamata del modulo di controllo può avvenire ad esempio grazie all'estensione dell'extension-point "*ParseProvider*", che dal

metodo “provides” effettua la chiamata alla classe “*CheckSODA*”, ogni volta che viene inserita o cancellata un’entità.

La classe “*CheckSODA*” si aspetta in ingresso i parametri:

- livello (Zooming table).
- fase (SODA).
- tipo entità sorgente.
- campo entità nome dell’entità sorgente.
- campo entità descrizione dell’entità sorgente.
- tipo entità destinataria.
- campo entità nome dell’entità destinataria.
- campo entità descrizione dell’entità destinataria.
- tipo operazione (inserimento/cancellazione).
- stato di migrazione (stato frapposto tra le fasi SODA).

Questi parametri servono per consentire alle classi di controllo, di navigare la struttura grammaticale, al fine di verificare la consistenza delle operazioni che il progettista compie e di mantenere aggiornato il documento XML finale.
Da notare che i parametri in ingresso rappresentano dati il più atomici possibile, per mantenere il maggior disaccoppiamento tra le parti.
Successivamente alla “verifica grammatica”, che corrisponde al core della procedura di controllo, se l’esito risulta positivo, avviene l’aggiornamento del file XML finale, che come si vede corrisponde all’output prodotto, differentemente con esito negativo, si genera un’eccezione.

3.5.3 DETTAGLIO DELL'ATTIVITA' DI "VERIFICA GRAMMATICA"

La macro-attività fondamentale del modulo di controllo è questa in figura 3.18, per ottenere la convalida dei dati di progetto.

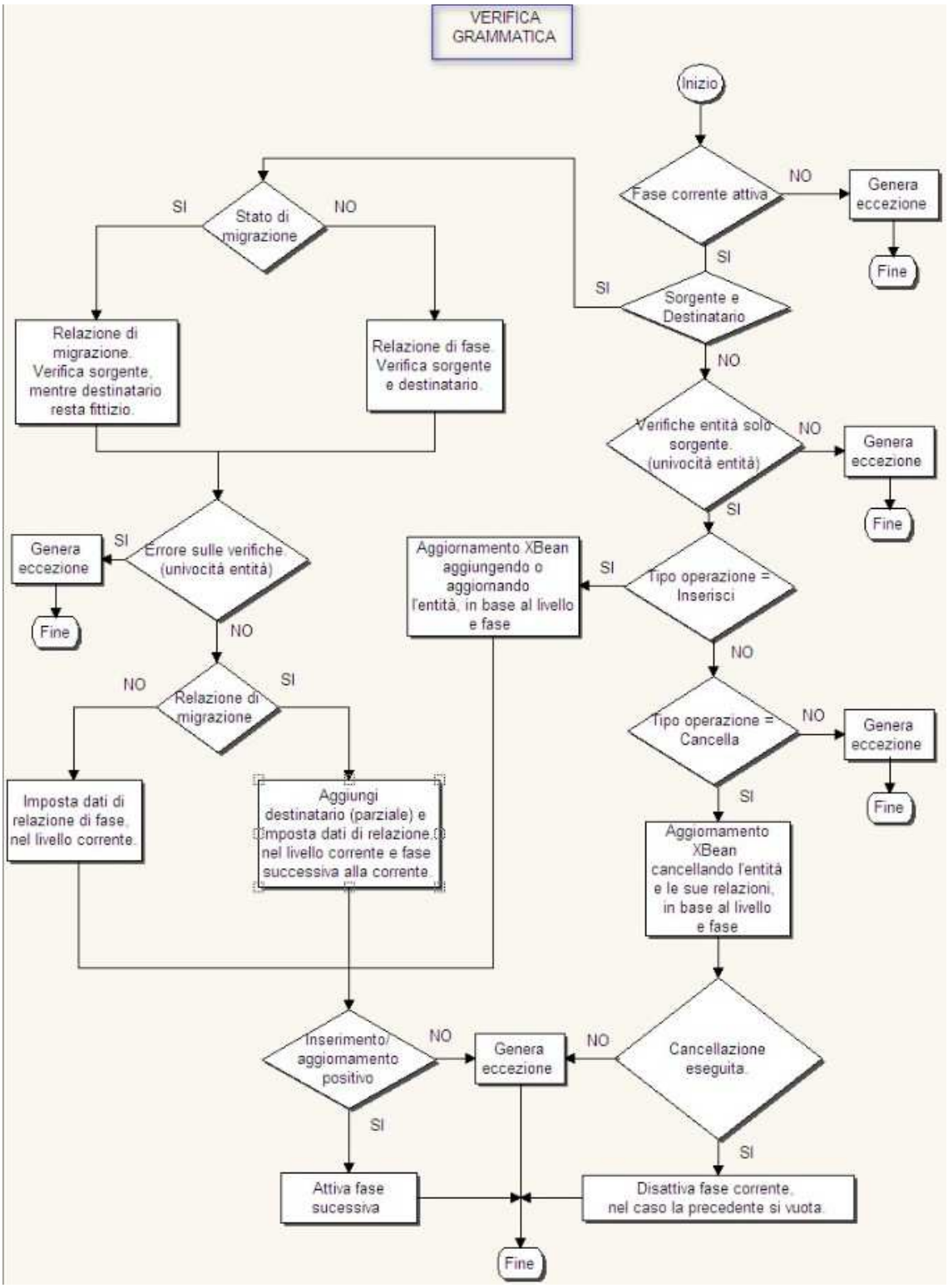


Figura 3.18. Dettaglio dell'attività di VERIFICA GRAMMATICA.

Durante la verifica della grammatica, la prima cosa da fare è controllare che la fase SODA, che arriva come parametro di input, sia attiva. Nel caso della prima fase, il problema non si pone mai, in quanto è sempre considerata attiva, mentre tutte le fasi SODA successive, si abilitano nel momento in cui viene aggiunta la prima entità nella fase corrente. Nel caso in cui la fase corrente non sia attiva, viene generata un'eccezione, con conseguente messaggio a video tramite frame java.
Lo step successivo di verifica, si basa sulla discriminazione della tipologia di entità che si ha in input, più precisamente si deve distinguere il caso di un'entità semplice oppure di un'entità di relazione. L'individuazione della tipologia di entità è resa possibile dalla presenza delle informazioni legate al sorgente ed al destinatario; se questi sono presenti entrambe, allora il controllore sa che deve gestire una relazione, altrimenti in presenza della sola sorgente deve gestire un'entità semplice.

Analizzando prima il caso di un'entità semplice, se non si verificano errori ed il controllo di univocità sul nome entità va a buon fine, viene considerata l'operazione da eseguire sull'entità, che può essere di nuovo inserimento, aggiornamento oppure di cancellazione, in caso contrario viene generata un'eccezione. Per tutte le operazioni, vale la medesima logica di ricerca dell'entità, nella struttura degli xbean. Nel caso dell'inserimento, se l'entità non esiste, viene creata; se esiste viene aggiornata. Nel caso della cancellazione, se l'entità esiste viene cancellata assieme a tutte le sue eventuali relazioni, se non esiste non succede nulla (caso teoricamente mai verificabile).
Al termine di un inserimento/aggiornamento andato a buon fine, abbiamo l'attivazione della fase successiva, mentre nel caso di una cancellazione andata a buon fine, viene disattivata la fase SODA successiva, nel caso quella corrente sia vuota.

Se l'entità inserita risulta essere una relazione, allora siamo in presenza di un sorgente e di un destinatario, che possono appartenere alla stessa fase SODA (relazione di fase), oppure a due fasi SODA diverse e consecutive (relazione di migrazione).
Nel caso di relazione di fase, si verifica la presenza delle due entità da collegare, all'interno della fase, mentre nel caso di una relazione di migrazione, si verifica la presenza della sorgente, mentre il destinatario viene parzialmente definito nello "*stato di migrazione*".

Infatti per facilità di progettazione, si è deciso di creare un'area di passaggio tra una fase SODA e l'altra, che permetta la definizione delle nuove entità destinatarie coinvolte nel legame tra le fasi, e che popoleranno la nuova successiva fase SODA. Le entità definite nello stato di migrazione, sono entità parzialmente definibili, in quanto è possibile solo sceglierne la tipologia ed il nome. Sarà compito del progettista completare le nuove entità, una volta passato definitivamente alla nuova fase SODA.
Con questo tipo di gestione, è possibile verificare la consistenza delle relazioni di migrazione essendo complete di sorgente e destinatario.

Anche in questo caso, prima di elaborare la relazione, viene verificata l'univocità del nome che la identifica, attraverso l'impiego dell'HashMap, configurata nella fase di "INIT".

3.6 PROGETTAZIONE ARCHITETTURALE

Identificando il ruolo che il modulo di controllo deve avere, possiamo dire che esso assume la regia della maggior parte delle azioni, da cui prendono origine le funzionalità dell'intero strumento di design.

Di seguito identifichiamo la struttura dell'intero applicativo, per dare una visione d'insieme (Figura 3.19)

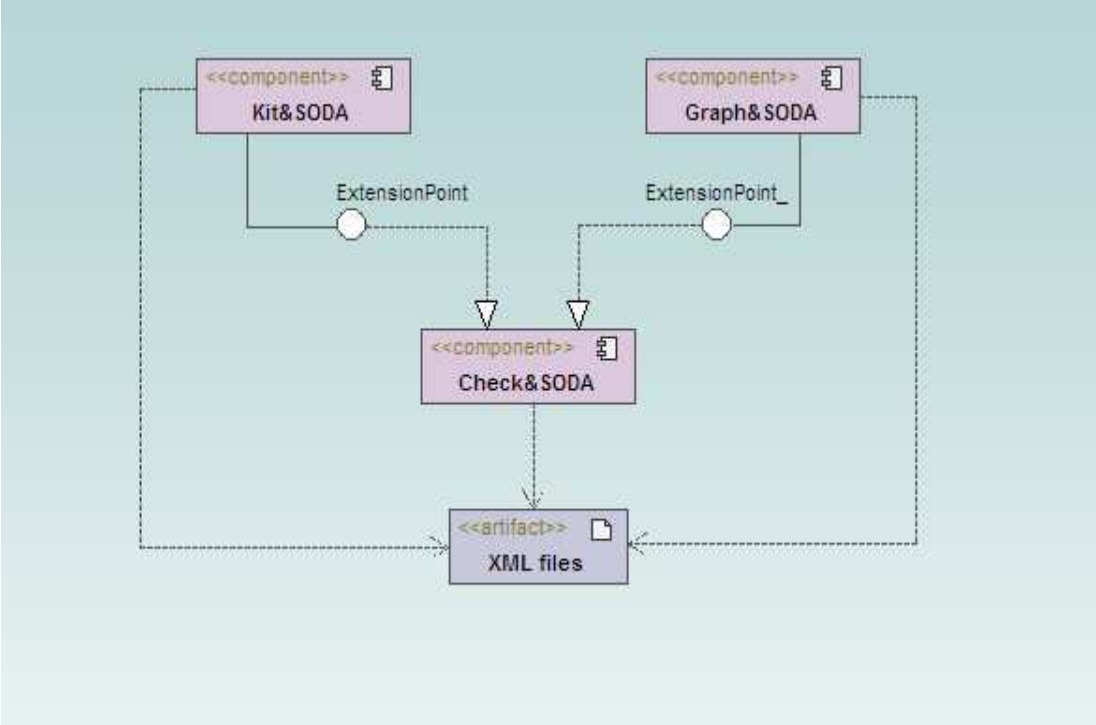


Figura 3.19 Struttura dell'applicativo di design per SODA.

Come prima cosa il modulo Check&SODA deve essere svincolato dagli altri moduli riportati nell'architettura, questo per consentire alla struttura di essere modulare e scalare, dando garanzia di poter scegliere un qualsiasi framework per la gestione della parte grafica e di quella manuale sulle tabelle. A tal proposito di seguito, è stata pensata l'interazione con il modulo grafico, per descrivere la ripartizione delle funzionalità.

Un framework potenzialmente adatto per lo sviluppo della parte grafica è GMF (Grafical Modeling Framework), adatto per la progettazione di editor grafici, utili al design di progetti di ogni tipo, attraverso la definizione di template contestualizzabili a diverse tipologie di problematiche e settori di applicazione.

Come detto precedentemente il modulo di controllo deve essere svincolato da qualsiasi altro modulo, e per questo esso verifica gli eventi scatenati dalle azioni (actions), tramite l'estensione degli extension-point.

Ad ogni evento rilevato si ha la verifica attraverso la grammatica, che può dare luogo a due macro azioni (Figura 3.20):

- aggiornamento dei file xml, in caso di nessuna anomalia;
- attivazione di un'eccezione contestuale alla problematica verificata.

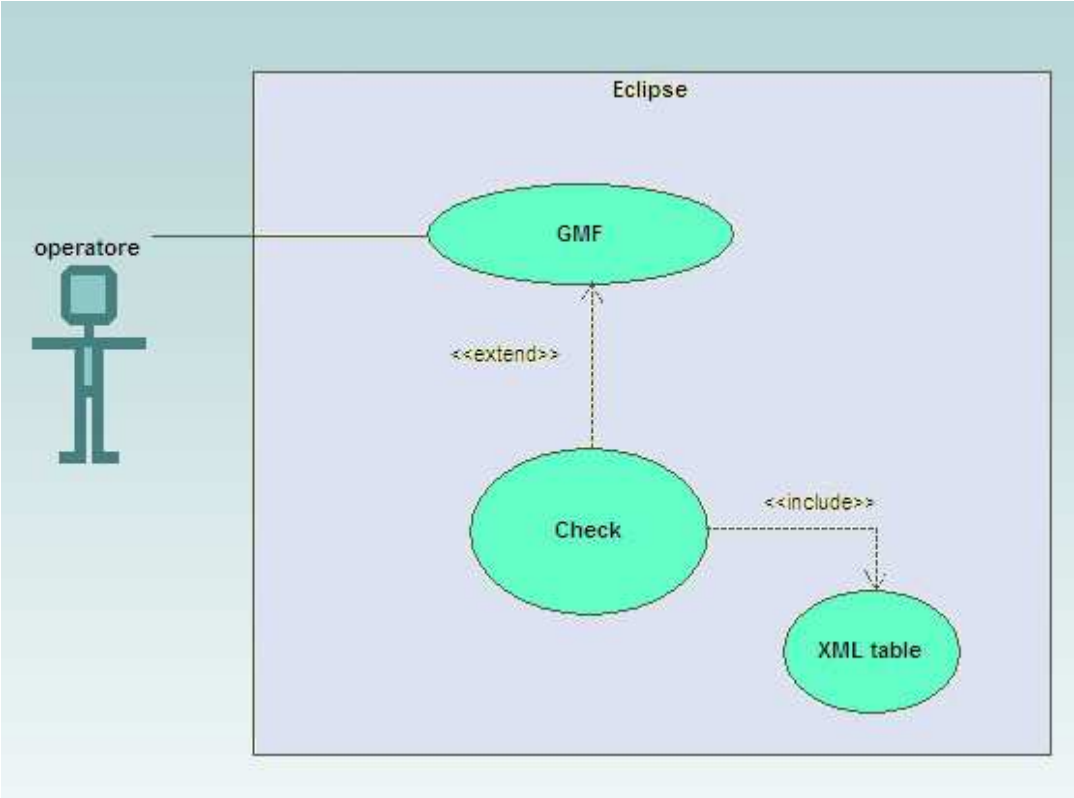


Figura 3.20 Macro azioni del sistema GMF e Check.

Il risultato finale prodotto dal controllo è la generazione di un documento XML che descrive le tabelle e il contenuto progettuale per il sistema multi-agente progettato dall'operatore (Figura 3.21).

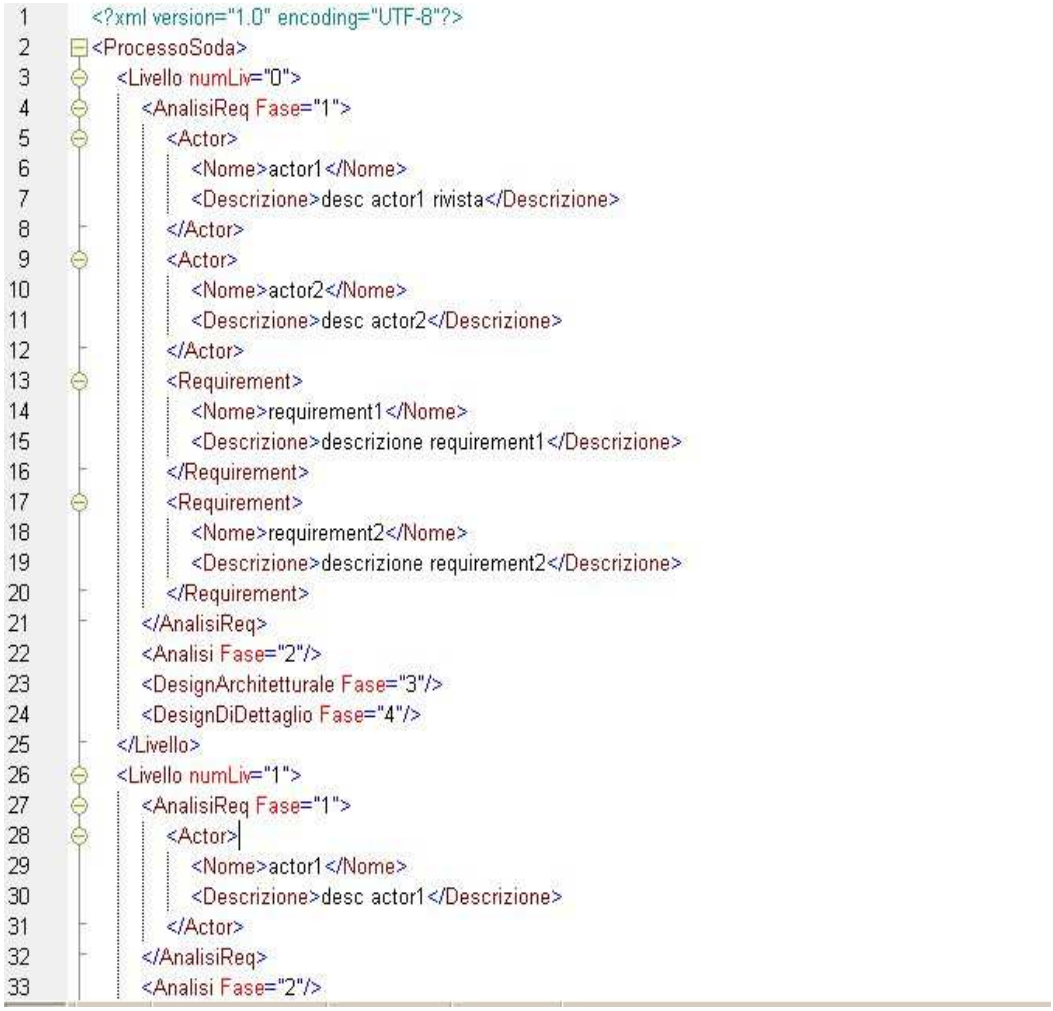


Figura 3.21 Esempio di output XML, prodotto.

Si è fatta la scelta di produrre file XML, per facilitarne l'esportazione verso altri sistemi, per consentire al modulo di "Kit&SODA" di effettuare xquery per accedere agevolmente ai dati (questo giustifica la relazione tra "Kit&SODA" e XML in figura 3.19), ed infine per consentire al modulo "Graph&SODA" di rappresentare le entità del progetto, attraverso lo specifico linguaggio grafico sviluppato ad hoc (questo giustifica la relazione tra "Graph&SODA" e XML in figura 3.19).

Il documento XML che descrive il contenuto informativo dei progetti di design, risulta essere allineato con gli XML che gestiscono il framevork GMF, e che descrivono le entità grafiche, i layout a video e la stato dei menù.

Come risultato finale abbiamo che il modulo di check, pilota le azioni dell'interfaccia grafica tramite avvisi su frame relativi all'interfaccia di Eclipse, mantenendo aggiornati i documenti finali; in questo modo abbiamo un disaccoppiamento dei moduli dove il controllo si limita a fare da cabina di regia, agendo sulle azioni che riceve in ingresso.

3.6.1 Il MODULO DI CONTROLLO

In figura 3.22, viene riportata una versione dell'architettura delle classi, relativamente al modulo di controllo (Check&SODA).

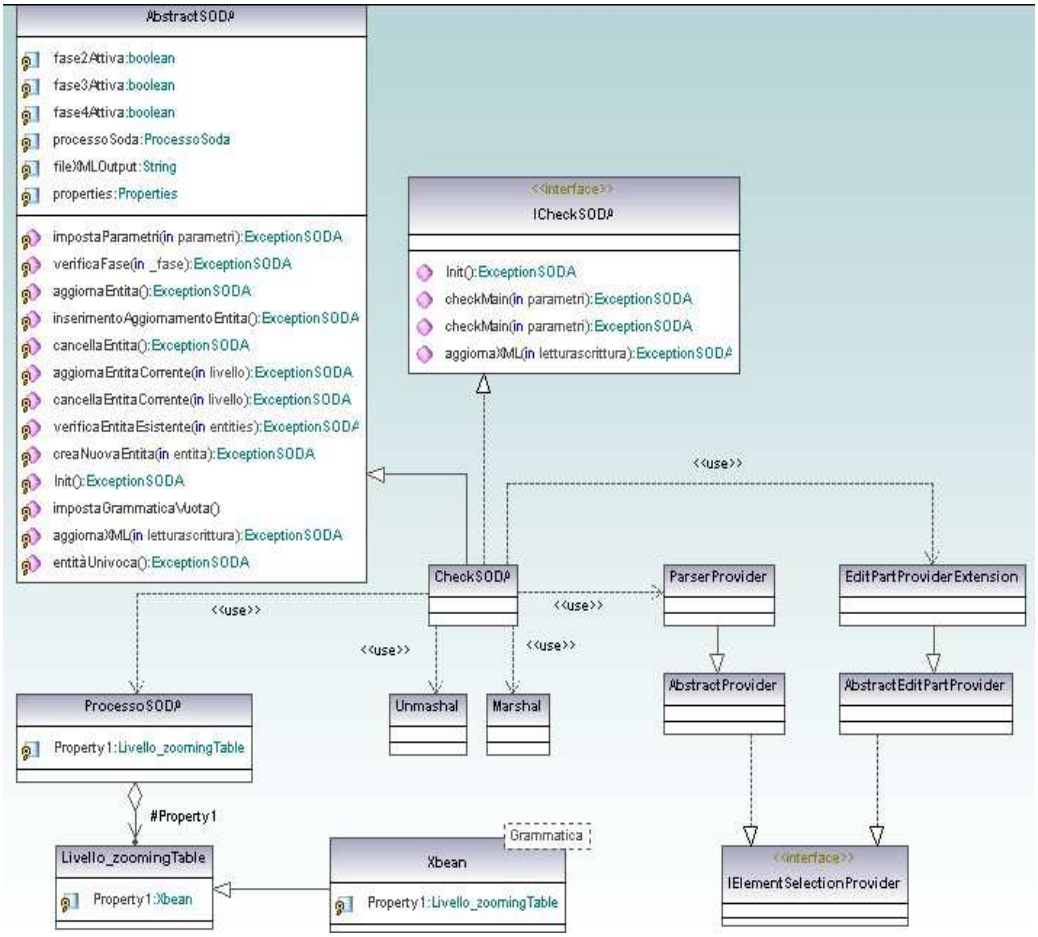


Figura 3.22 Architettura delle classi Check&SODA

"L'*AbstractProvider*" e "*AbstractEditPartProvider*" altri non sono che gli extension-point del GMF framework, tali abstract sono estesi rispettivamente dalle classi "*ParserProvider*" e "*EditPartProviderExtension*", nelle quali si innescano le chiamate al modulo di controllo Check&SODA.

Le informazioni raccolte sono gestite dalla classe "*CheckSODA*" la quale, estendendo l'" *AbstractSODA*" implementa la logica di controllo, attraverso la grammatica rappresentata nella struttura degli xbean, tramite il "*ProcessoSODA*" che contiene la struttura a livelli della Zoomin Table, vale a dire la classe "*Livello*". La "*CheckSODA*" fa uso delle classi "*Marshal*" e "*Unmashal*" per l'aggiornamento del documento finale e l'impostazione degli oggetti xbean in memoria.

3.6.2 INTERAZIONE TRA FUNZIONALITA' E CONTROLLO

Le funzionalità sono identificate prendendo come riferimento la metodologia SODA descritta nei paragrafi precedenti.

Principalmente abbiamo lo sviluppo delle quattro fasi, di cui nella prima si entra in automatico, all'atto della definizione di un nuovo progetto. Per passare ad una successiva fase, si utilizzano i pulsanti menù, messi a disposizione a livello del GMF, e sempre attivi. In questo modo all'operatore non vengono imposti limiti sull'utilizzo dell'interfaccia grafica, ma saranno successivamente le procedure di controllo a verificare, se le operazioni eseguite sono correte o meno, permettendone l'esecuzione oppure avvisando del tipo di errore che si verifica. Per tutto quello che riguarda la gestione delle tabelle a livello grafico, il contenuto a video ed il layout, resta a carico del GMF framework. La gestione dei menù e pulsanti per l'attivazione delle fasi è definita da action-set aggiunti come plug-in al GMF framework. Come prima versione dell'applicazione SODA, si è pensato di mantenere il file XML prodotto, dal modulo di controllo, all'interno della struttura a package del progetto "Check&SODA", in modo da consentirne il salvataggio ed eventuale export, direttamente dalle funzionalità del workbench di Eclipse.

Nel modulo di controllo è stato comunque previsto un file di configurazione, per dare la possibilità di poter modificare il path del file di output ed eventualmente implementare altre parametrizzazioni.

Ogni fase del processo SODA, ha la propria shape template disponibile per la progettazione, contenente le figure delle entità relative alla fase corrente che si sta progettando; la gestione dei template resta a carico del GMF framework.

Il passaggio tra le fasi, implica un processo di transizione, gestito in una zona apposita, la quale consente le mappature delle nuove entità della fase successiva, partendo dalle entità definite nella fase SODA precedente e permetterne la definizione delle relazioni di migrazione.

Non è possibile cominciare la progettazione di una nuova fase, senza che quella precedente contenga almeno un elemento (entità), e non è possibile aggiungere nuove entità nella fase corrente (esclusa la prima), senza avere definito le relazioni di migrazione tra le entità di due fasi SODA consecutive.

Nelle prime tre fasi del processo SODA, si ha la funzionalità di layering, anche questa attivabile attraverso un pulsante di menù, sempre con l'aggiunta di un action-set plugin nel modulo GMF. Il layering permette l'inserimento di un nuovo livello di dettaglio (o astrazione) delle entità, a sua volta verificato e memorizzato dall'unità di controllo nell'oggetto "*Livello*", che assume il ruolo di Zooming table.
A livello grafico sono ovviamente previsti i pulsanti per la navigazione attraverso i livelli (layer) di volta in volta definiti.
Nella quarta fase SODA, la funzionalità di layering non più presente, viene invece attivata la funzionalità di carving che permette la selezione delle entità che andranno a definire il sistema multi-agente. Tale selezione è resa possibile negli oggetti xbean tramite il flag "flCarving", che appunto indica le entità e relazioni, individuate dal progettista a qualsiasi livello di zooming.

La gestione delle funzionalità di import, da parte del modulo Kit&SODA e Graph&SODA, implica che siano loro stessi ad occuparsi dell'aggiornamento del file XML di output. A seguito dell'aggiornamento dell'output i moduli sopracitati, possono attivare un flag di avviso aggiornamento avvenuto, per gli altri moduli del sistema di progettazione, affinché anch'essi possano attivarsi per le verifiche e gli aggiornamenti dei loro stati.

4 SCELTE IMPLEMENTATIVE E COLLAUDO.

In questo capitolo, vengono date le indicazioni di carattere tecnico e le considerazioni fatte a livello implementativo, a seguito dell'analisi e del progetto che sono state sviluppate nel capitolo precedente.
Partendo dal requisito di realizzare un plug-in SODA per la piattaforma Eclipse, emerge l'esigenza di come integrare le funzionalità implicite del plug-in, con quelle eventualmente già presenti nel workbench di Eclipse (Sezione 4.1). Risulta importante progettare una corretta struttura grammaticale, per gestire al meglio l'efficienza dei controlli sui dati (Sezione 4.2), e successivamente come effettuare i test di collaudo (Sezione 4.3).

4.1 INTEGRAZIONE DELLE NUOVE FUNZIONALITA' IN ECLIPSE

Come precedentemente indicato, uno dei primi requisiti stabiliti per il sistema, è di tipo tecnologico e riguarda la piattaforma di implementazione; infatti è richiesto che lo strumento di supporto per il modulo di controllo, assieme agli altri due moduli di progettazione dei sistemi complessi multi-agente, sia un plug-in da integrare nella piattaforma Eclipse.

Tra le funzionalità richieste, quelle di import/export, relativamente al file XML di output, sono gestibili in modo autonomo da tutti i moduli che fanno parte del progetto SODA; nel caso specifico della funzionalità di import, significa che ciascuno dei moduli, può importare dati dall'esterno, preoccupandosi di aggiornare il file di output ed il proprio stato. Al termine di queste operazioni si avvisano gli altri moduli di un avvenuto aggiornamento, in modo che anch'essi possano avere la possibilità di validare il documento XML e di aggiornare il proprio stato. Nel contesto di import, il modulo Chech&SODA, ha il compito di validare il documento finale, sia che l'import venga eseguito da lui, sia che venga gestito da uno degli altri moduli. Non è comunque obbligatorio, che avvenga la validazione, proprio perché ciascuno dei moduli è indipendente ed autonomo rispetto agli altri, ma sicuramente la verifica di consistenza di tutti i moduli, serve a mantenere il sistema coerentemente aggiornato.

Per la funzionalità di export, le cose risultano più semplici, in quanto non ci sono aggiornamenti sul file di output, che risultando un XML puro, può essere trattato come un normale file di testo e possono essere sfruttate le già presenti funzionalità di Eclipse. E' bene specificare che le funzionalità di Eclipse, sono sfruttabili, nel momento in cui il documento di output prodotto, risieda all'interno della struttura di progetto del sistema multi-agente, su cui il progettista lavora. Questo aspetto non è scontato, dal momento che si è prevista una diversa collocazione del file di output, anche al di fuori dei progetti, grazie all'impiego di un file di configurazione presente nel modulo Check&SODA, in cui si può indicare una diversa impostazione del path.

Tutte le altre funzionalità di verifica, sono invece proprie delle classi di controllo.

4.2 SCELTA DELLA STRUTTURA GRAMMATICALE

La definizione della struttura grammaticale, è apparsa fin da subito d'importanza centrale, in quanto una buona struttura snella e completa, condiziona la buona organizzazione delle classi che devono effettuare i controlli.
I controlli avvengono tutti a run-time, e scattano nel momento stesso che, vengono definite le entità e le relazione nell'editor, per dare immediatamente segnalazione di errore, nel caso di incongruenze sulle regole della grammatica stessa.

Una delle maggiori difficoltà incontrate durante la progettazione del modulo Check&SODA, è stata quella di come collocare gli oggetti, rappresentativi delle entità di relazione (sia di fase, che di migrazione), rispetto alle entità delle singole fasi. Questa particolare problematica, condiziona fortemente la gestione dell'integrità referenziale, che le classi di controllo devono garantire, navigando la struttura grammaticale. Se la grammatica risulta non ben organizzata, si rischia di appesantire l'ispezionabilità delle informazioni, generando overload dei processi e conseguente decadimento delle prestazioni, in termini di velocità di elaborazione, ma soprattutto di maggiore utilizzo delle risorse legate alla memoria.

Per quanto appena evidenziato, durante la fase di progettazione della struttura grammaticale, si sono provate diverse soluzioni logistiche delle entità di relazione, prima all'esterno delle entità, poi al loro interno, ed in fine il migliore compromesso è stato quella di una struttura ibrida, tra le due menzionate.

In figura 4.1 e 4.2, si evidenziata la rappresentazione delle relazioni e delle loro tipologie.

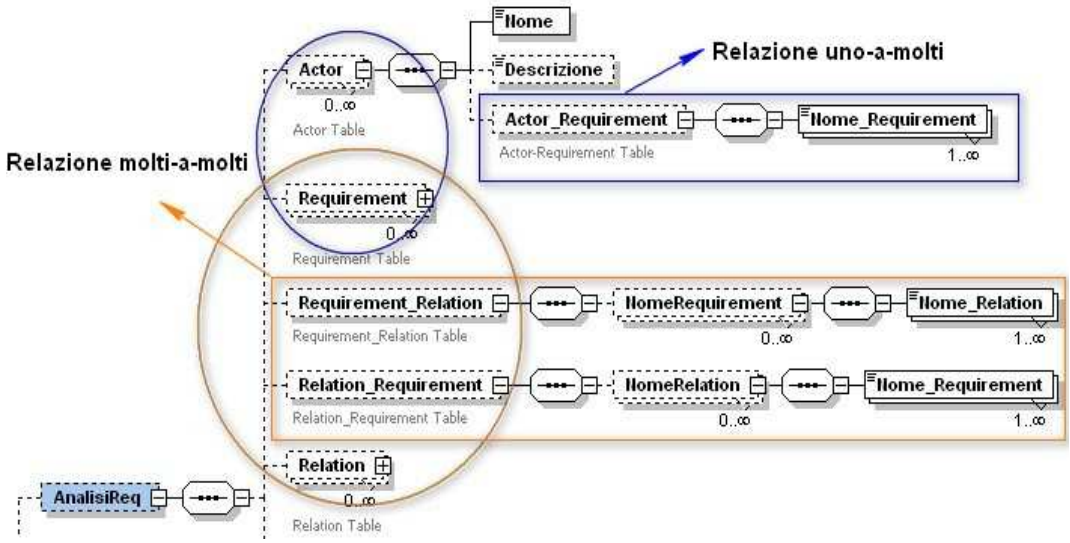


Figura 4.1 Caso del legame uno-a-molti e molti-a-molti, per le relazioni di fase.

Le relazioni di fase semplici, uno-a-molti, sono collocate direttamente all'interno delle entità, in quanto per il caso della cancellazione dell'entità, viene cancellata anche la relazione di fase, senza ulteriori verifiche sull'entità destinataria.
Per le relazioni molti-a-molti, gestite con due relazioni uno-a-molti, queste sono localizzate fuori dalle entità, per agevolare una migliore navigazione e verifica del sorgente e destinatario della relazione.

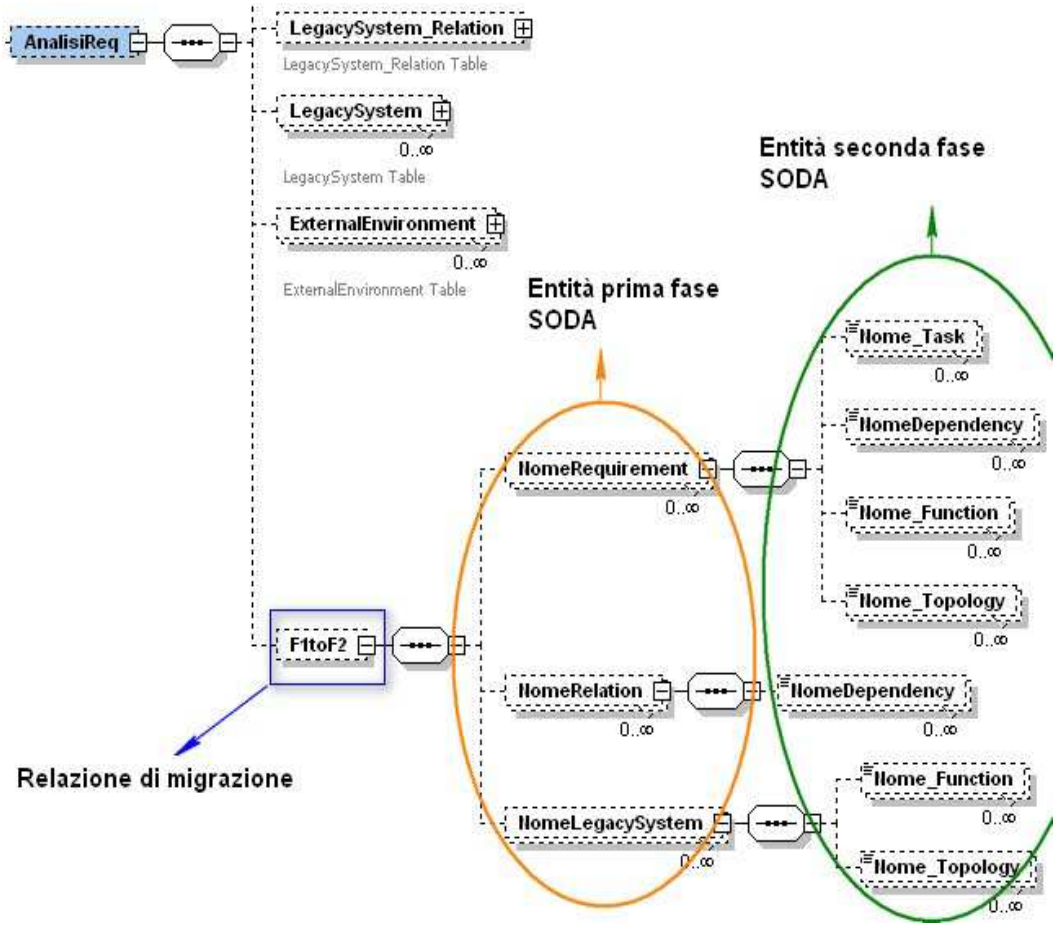


Figura 4.2 Caso del legame uno-a-molti, per le relazioni di migrazione.

Le relazioni di migrazione, contengono le entità della fase N e le entità della fase N+1 di SODA. Le relazioni sono di tipo uno-a-molti, nel caso specifico della prima e seconda fase SODA.

Tutte le entità e tutte le relazioni, sono riproducibili all'interno di infiniti livelli, che permettono la gestione della funzionalità di zooming; tutti i controlli legati ad un singolo livello, sono i medesimi degli altri livelli, quindi il modulo di controllo si muove all'interno di un singolo livello per volta.

4.3 TEST E COLLAUDO

Lo sviluppo è avvenuto all'interno della struttura standard, che di default viene definita dal modulo GMF, quando questo viene creato; è in questa struttura che sono stati ricavati i package del modulo di controllo Check&SODA.

Tale scelta è stata fatta per una ragione di comodità, infatti inizialmente, per individuare e testare gli extension-point del modulo GMF, che andassero bene per l'attivazione dei controlli, è risultato comodo avere tutto in un unico progetto Eclipse i package necessari.

I package del modulo di Check&SODA, sono facilmente individuabili dal loro nome (Figura 4.3).

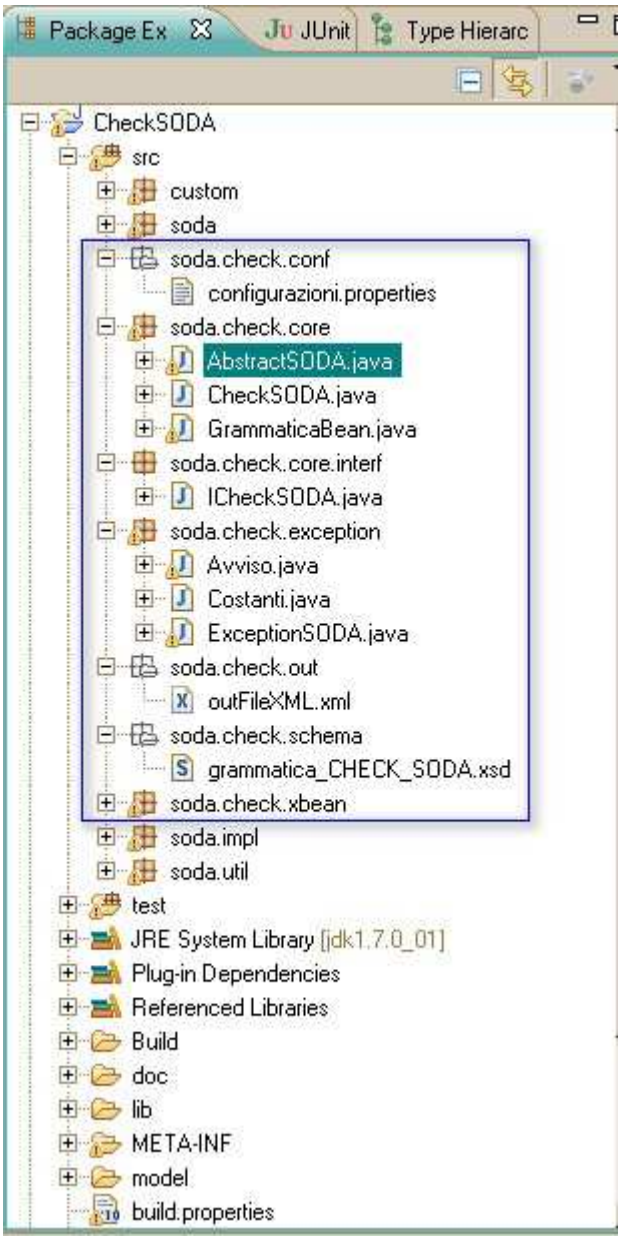


Figura 4.3 Package soda.check.

Tutte le classi del modulo di controllo si trovano contenute nel package “soda.check”.

Il passo successivo che si deve fare per distribuire il modulo Check&SODA, che risulta indipendente dagli altri framework, , è quello di racchiudere i package in un jar a parte.

Per quanto riguarda lo sviluppo dei test è stato previsto un package a parte, utile a localizzare la classe di test JUnit.

L’impiego del framework JUnit, non è stato il solo strumento di test usato, ma si è fatto anche largo utilizzo di debug su parti di codice, per analizzare meglio l’evoluzione della generazione degli oggetti xbean e loro conseguente compilazione.

Possibili migliorie per versioni successive, sono possibili, ed in particolare si potrebbe ad esempio far uso di una HashMap, per memorizzare le relazioni che sono consentite, dalla metodologia SODA, permettendo alle classi di controllo, di sapere immediatamente se le relazioni sono esistenti a meno, senza dover navigare parte della struttura ad oggetti xbean. Il popolamento della HashMap potrebbe prevedersi in fase di init, leggendo le chiavi di relazione direttamente del file di configurazione (.properties).

Allo stato attuale in questa prima versione del modulo di controllo, sono state realizzate le funzionalità di inizializzazione delle strutture che servono per le attività di verifica sui dati. Il controllo sui dati di input, prima del loro impiego nell’attività di verifica grammaticale, la quale si articola nelle funzioni già in essere di accertamento di univocità delle entità e relazioni inserite, non che del loro inserimento, aggiornamento e cancellazione, con conseguente mantenimento dell’integrità referenziale. Altre funzionalità presenti sono quelle di verifica dell’attivazione e disattivazione delle fasi, in base alla presenza o meno di almeno un’entità per fase, su di un livello.

Conclusioni

A conclusione del lavoro svolto, mettiamo in evidenza i punti che sono stati sviluppati, partendo dal contesto relativo al modulo di controllo, per il sistema software che deve permettere la progettazione di MAS, attraverso l'uso della metodologia SODA. Il modulo di controllo identificato dal nome Check&SODA, fa parte di un progetto che racchiude altri due moduli, Kit&SODA e Graph&SODA, che assieme definiscono il nuovo plug-in di sviluppo per la piattaforma Eclipse.

Per prima cosa sono state stabilite le basi tecnologiche per la realizzazione del sistema, esplorando la piattaforma Eclipse ed effettuando un'analisi della metodologia SODA, per poi arrivare alla progettazione del controllo, definendo in prima istanza la struttura grammaticale e quindi le regole che devono essere rispettate, per ottenere la validazione dei dati e la consistenza delle azioni da parte degli altri moduli di progetto. Una volta definita la grammatica, in base ad essa si sono definite quali classi di controllo fossero necessarie, individuandone per ciascuno il ruolo e la logica in base alle funzionalità di validazione richieste.
Durante la progettazione di Check&SODA, si è cercato di mantenere chiarezza nelle varie fasi di sviluppo e in quella che veniva prodotto di volta in volta, dalla grammatica, alle classi, ai ruoli in base alle funzionalità, con l'obiettivo di ottenere un codice altrettanto chiaro, snello e pulito. L'obiettivo è stato in parte ottenuto, ma può essere migliorato, soprattutto curando la gestione dell'integrità referenziale.

Non solo l'analisi e lo sviluppo descritto in questa tesi, costituiscono la parte documentale del progetto Check&SODA, ma direttamente dalle classi si hanno delle informazioni utili, attraverso il javaDoc implementato ed ai commenti a livello di codice.
E' quindi possibile ottenere una versione documentale anche in formato html, del progetto java, aspetto fondamentale per la fruizione delle classi, che possono essere tranquillamente riprese e sviluppate in evoluzioni future.

Sfruttando le indicazioni fornite dalla parte grafica [7], sono stati individuati i punti di contatto, tra i moduli Check&SODA e Graph&SODA facendo uso degli extension-point.

Come risultato finale della tesi, si è ottenuto il progetto, funzionante con tutte le caratteristiche richieste. Restano da sviluppare solamente casi particolari, per far funzionare l'integrità referenziale in modo completo ed affidabile.

Appendice A:

A.1 DETTAGLIO DELLA GRAMMATICA

Di seguito il dettaglio delle fasi SODA 2,3,4 con le relative entità di migrazione

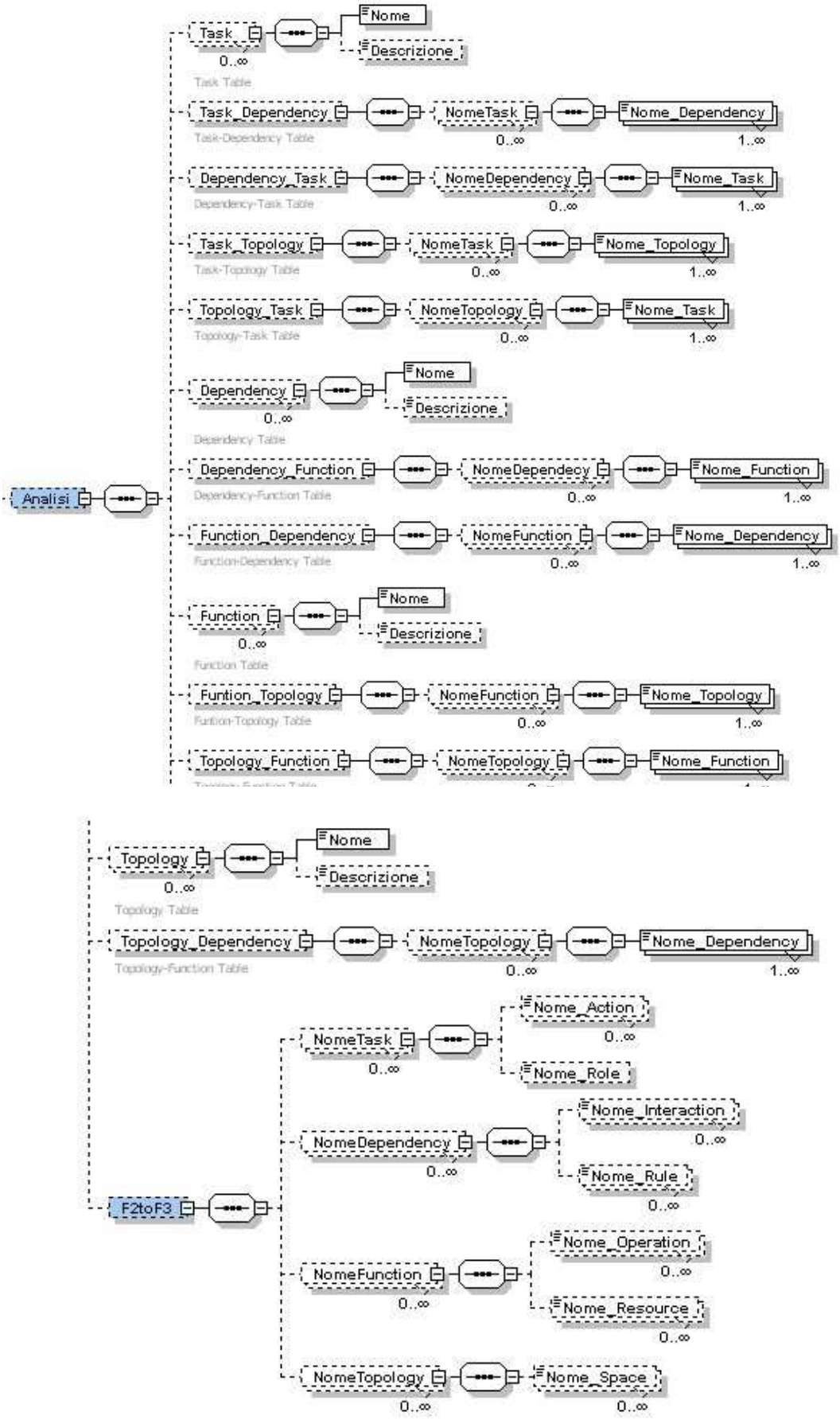
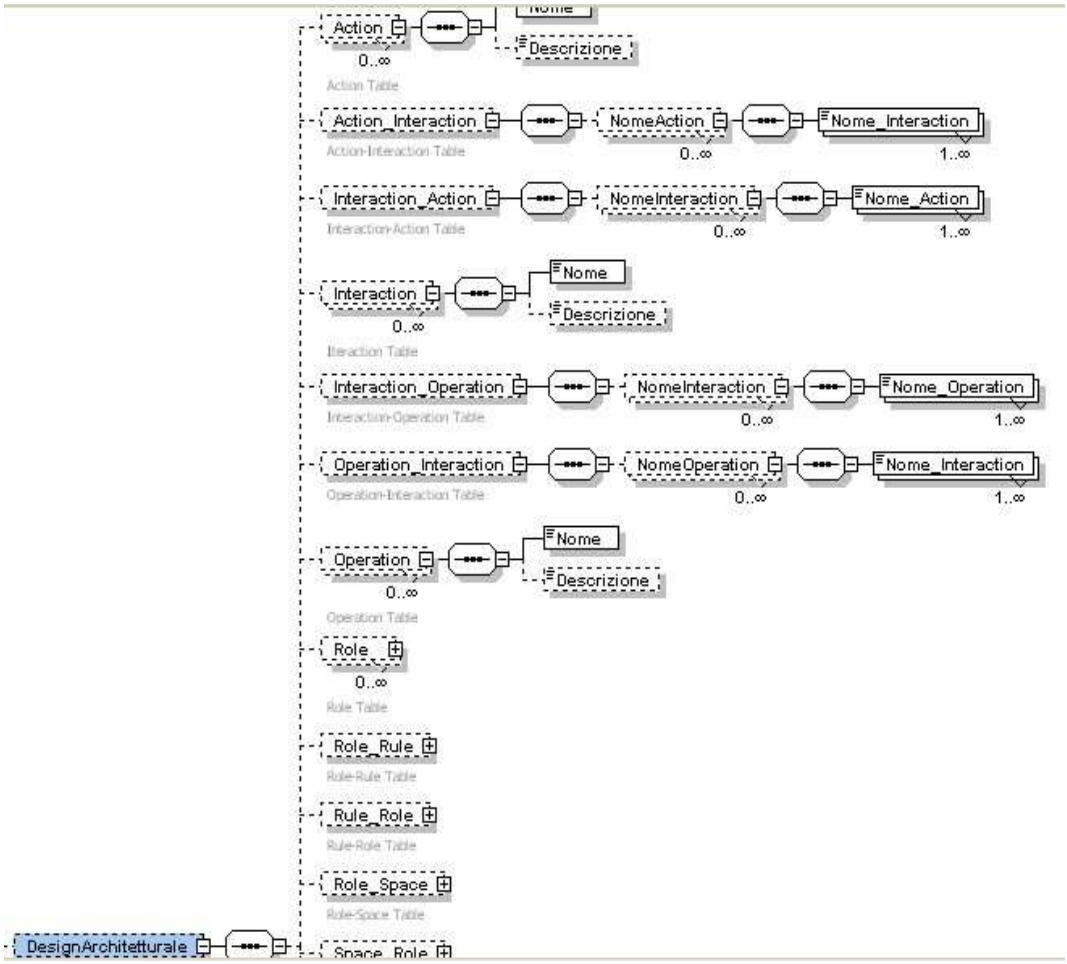
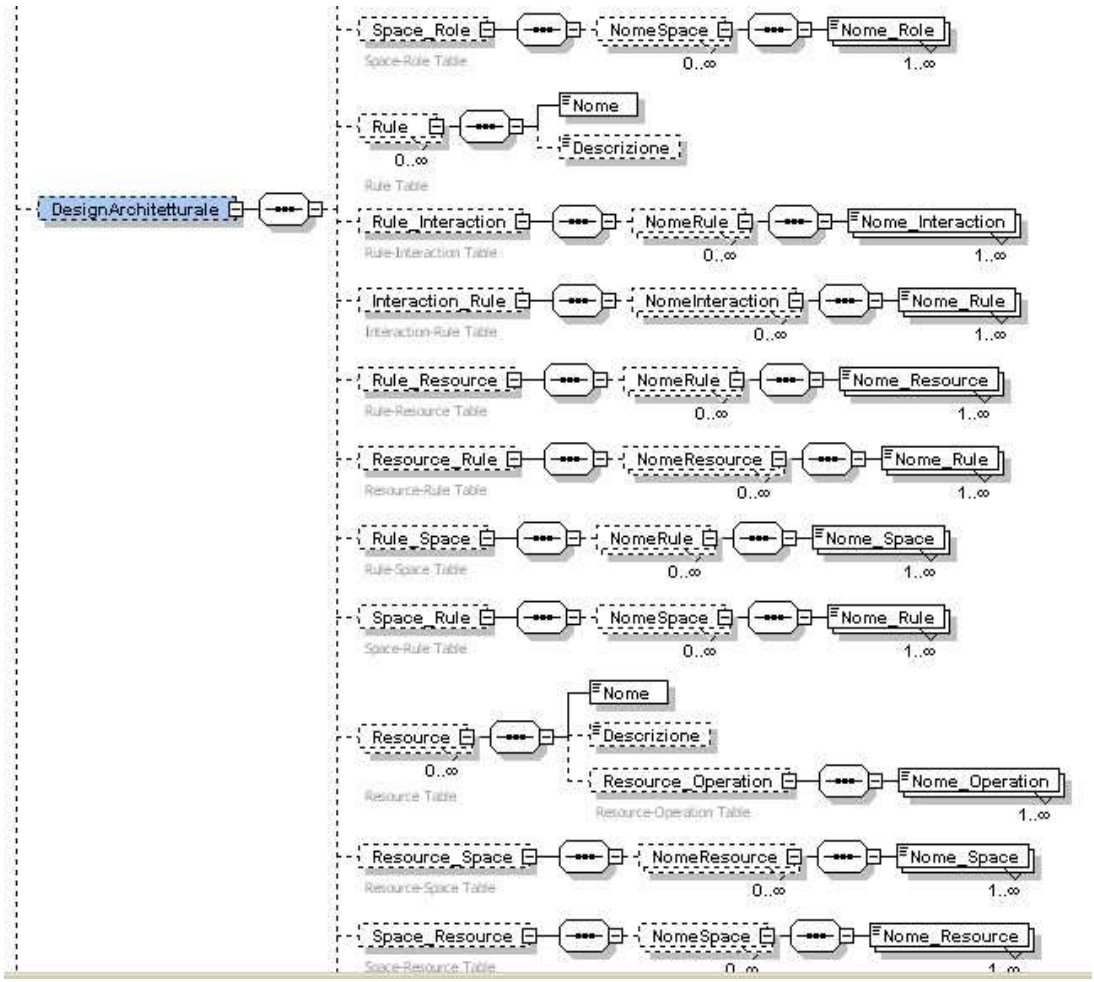


Figura A.1. Dettaglio della struttura grammaticale, relativa alla seconda fase SODA.


Action
0..∞
Action Table
Nome
Descrizione
Action_Interaction
Action-Interaction Table
NomeAction
0..∞
Nome_Interaction
1..∞
Interaction_Action
Interaction-Action Table
NomeInteraction
0..∞
Nome_Action
1..∞
Interaction
0..∞
Interaction Table
Nome
Descrizione
Interaction_Operation
Interaction-Operation Table
NomeInteraction
0..∞
Nome_Operation
1..∞
Operation_Interaction
Operation-Interaction Table
NomeOperation
0..∞
Nome_Interaction
1..∞
Operation
0..∞
Operation Table
Nome
Descrizione
Role
0..∞
Role Table
Role_Rule
Role-Rule Table
Rule_Role
Rule-Role Table
Role_Space
Role-Space Table
DesignArchitetturale
Space_Role


DesignArchitetturale
Space_Role
NomeSpace
Nome_Role
Space-Role Table
0..∞
1..∞
Rule
Nome
Descrizione
0..∞
Rule Table
Rule_Interaction
NomeRule
Nome_Interaction
Rule-Interaction Table
Interaction_Rule
NomeInteraction
Nome_Rule
Interaction-Rule Table
Rule_Resource
Nome_Resource
Rule-Resource Table
Resource_Rule
NomeResource
Resource-Rule Table
Rule_Space
Nome_Space
Rule-Space Table
Resource
Resource_Operation
Nome_Operation
Resource Table
Resource-Operation Table
Resource_Space
Resource-Space Table
Space_Resource
Space-Resource Table

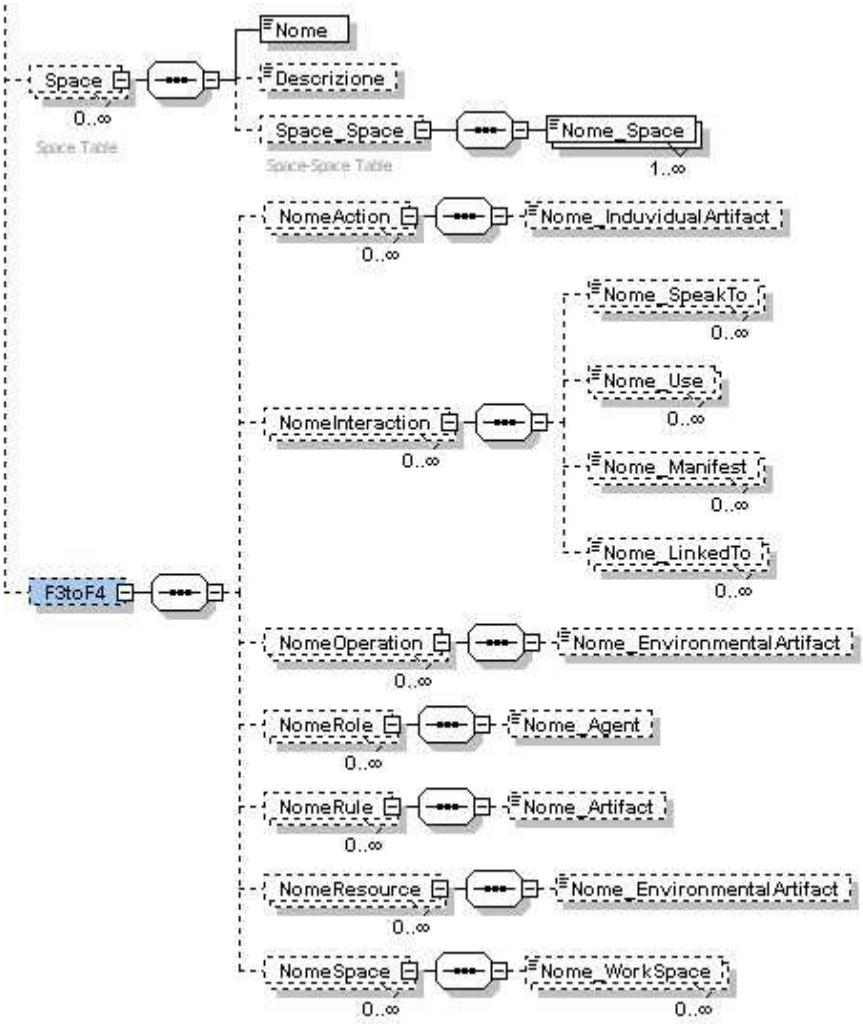
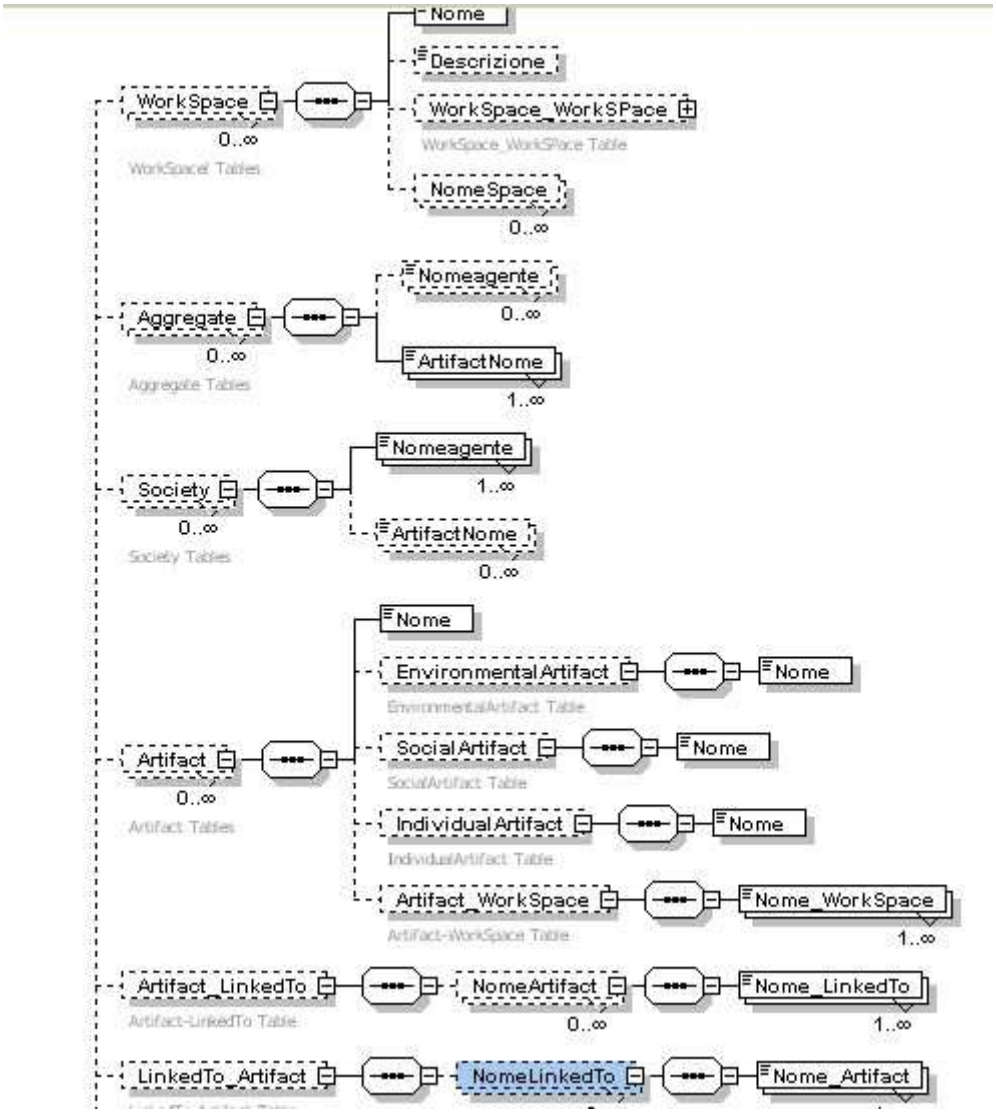


Figura A.2. Dettaglio della struttura grammaticale, relativa alla terza fase SODA.


Nome
Descrizione
WorkSpace
0..∞
WorkSpace Tables
WorkSpace_WorkSPace
WorkSpace_WorkSPace Table
NomeSpace
0..∞
Nomeagente
0..∞
Aggregate
0..∞
Aggregate Tables
ArtifactNome
1..∞
Nomeagente
1..∞
Society
0..∞
Society Tables
ArtifactNome
0..∞
Nome
EnvironmentalArtifact
Nome
EnvironmentalArtifact Table
Artifact
0..∞
Artifact Tables
SocialArtifact
Nome
SocialArtifact Table
IndividualArtifact
Nome
IndividualArtifact Table
Artifact_WorkSpace
Nome_WorkSpace
1..∞
Artifact-WorkSpace Table
Artifact_LinkedTo
Artifact-LinkedTo Table
NomeArtifact
0..∞
Nome_LinkedTo
1..∞
LinkedTo_Artifact
NomeLinkedTo
Nome_Artifact

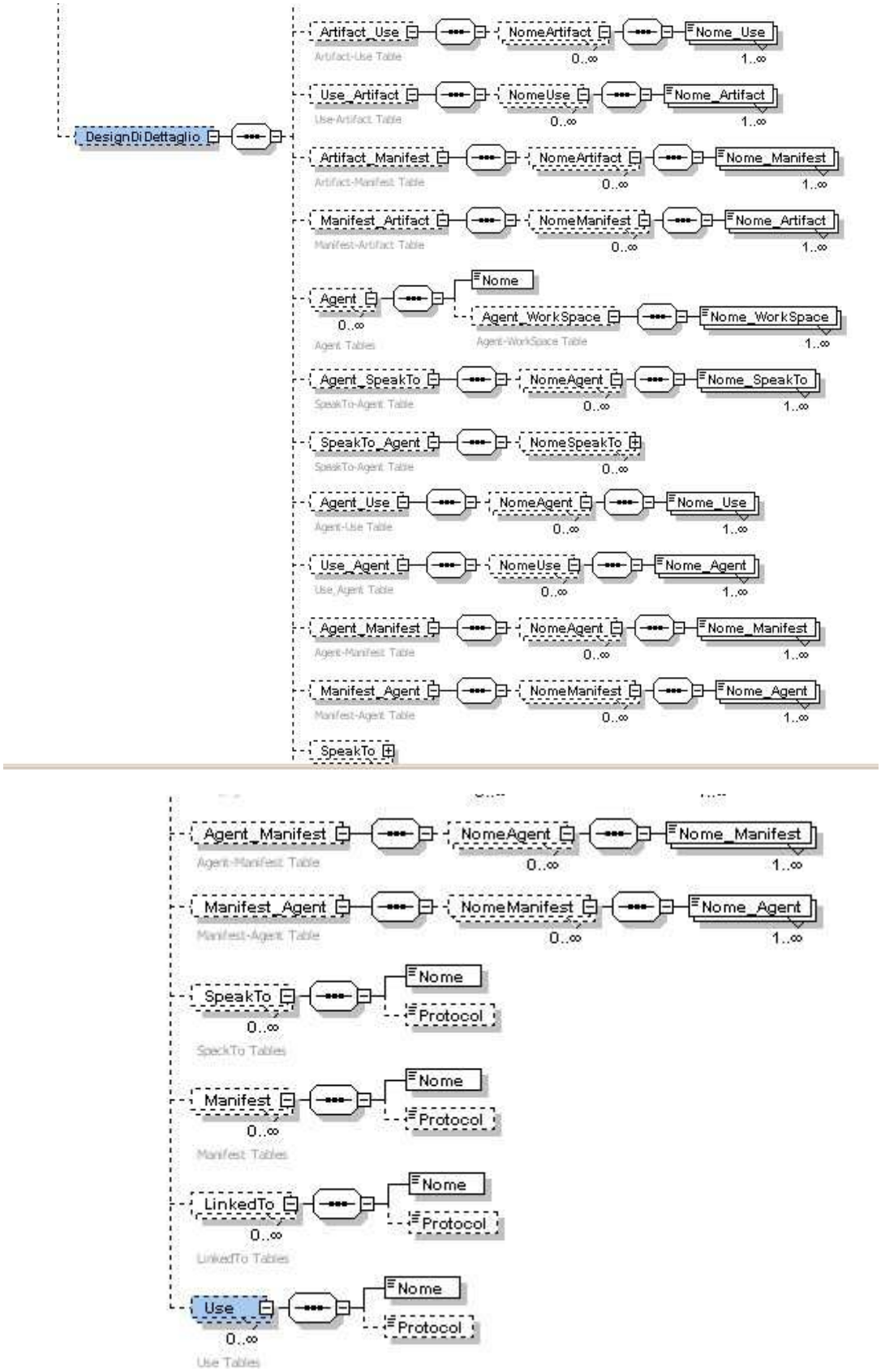


Figura A.3. Dettaglio della struttura grammaticale, relativa alla quarta fase SODA.

Bibliografia

[1] Documentazione “OSGi with Eclipse Equinox”.
[Online]: http://www.vogella.de/articles/OSGi/article.html

[2] Documentazione “The OSGi Architecture”.
[Online]: http://www.osgi.org/About/WhatIsOSGi

[3] Documentazione “Eclipse platform architecture”.
[Online]:http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/arch.htm

[4] Documentazione “Eclipse 3.X Plugin development”
[Online]: http://www.ing.unisannio.it/dilucca/GSSW/materiale09/eclipse-plugin-dev.pdf

[5] Documentazione “Notes on the Eclipse Plug-in Architecture”.
[Online]: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

[6] Documentazione della metodologia SODA.
[Online]: http://apice.unibo.it/xwiki/bin/download/SODA/Documents/SODADoc.pdf

[7] Aldrovandi Davide, Enrico Denti, Ambra Molesini, “Analisi e sviluppo di un linguaggio grafico e del relativo strumento software per il supporto alla metodologia SODA”