



AMPLE
Aspect-Oriented, Model-Driven, Product Line
Engineering
Specific Target Research Project: IST-33710

**Software support for the
traceability framework,
including extension of
current configuration
management and product
line evolution model**

ABSTRACT

This document describes the first version (prototype) of a traceability framework for aspect-oriented, model-driven software product line development. This framework consists of repository where traceability information is recorded, a wrapping graphical interface offering basic queries, views, and "populators" to manage the information in the repository and additional plugin mechanism to add new functionalities to the framework. We describe the functionality of the base repository and the available mechanisms (plugins) to extend it. We present the graphical framework wrapping the repository in its current version, as well as plans for a new architecture. All individual, implemented plugins are then described. We conclude with a description of future endeavours.

Document ID:	AMPLE D4.2
Deliverable/Milestone No:	D4.2
Workpackage No:	WP4
Type:	Deliverable
Dissemination:	PU
Status:	COMPLETE
Version:	V1.1
Date:	October 15, 2008
Author(s):	Jean-Claude Royer (EMN), Joost Noppen (EMN), Nicolas Anquetil (EMN), Andreas Rummeler (SAP), Ralf Mitschke(TUD), André Sousa (UNL), Uira Kulesza (UNL), Raffi Khatchadourian (Lancs), Phil Greenwood (Lancs), Awais Rashid (Lancs), Ismênia Galvão (UT)

Project Start Date: 01 October 2006

Duration: 3 years

History of Changes

Version	Date	Changes
0.1	2008-07-18	Initial Version
0.2	2008-08-22	First Draft Version
0.3	2008-09-02	Version with GAMBLE
0.4	2008-09-08	Version with overview
1.0	2008-09-12	Final version with conclusion
1.1	2008-09-14	Last minute additions and corrections

Contents

Overview	8
1 ATF: The Base Repository	10
1.1 Concepts	10
1.1.1 Infrastructure	10
1.1.2 Eclipse Integration	11
1.2 Common Tasks	13
1.2.1 Repository Creation	13
1.2.2 Type Creation	14
1.2.3 Artefact and Link Creation	16
1.2.4 Artefact and Link Removal	17
1.2.5 Collecting Trace Information	18
1.2.6 Querying for Artefacts and Links	21
2 Graphical Front-End	23
2.1 Current Structure	23
2.2 Common Tasks	24
2.2.1 Creating a Traceability Project	24
2.2.2 Definition of new Trace Links	27
2.2.3 Submitting Queries and Viewing Results	27
2.3 Extension Mechanism	31
2.4 Evolution of the Traceability Framework	32
3 Implemented plugins	36
3.1 Graph representation	36
3.1.1 Hypergraphs	36
3.1.2 Overall Design	37
3.1.3 The Graph Visualization Plugins	37
3.2 Metrics	40
3.3 Configuration and evolution management	42
3.3.1 Feature Driven Versioning	43
3.3.2 Illustrative Example	43
3.3.3 Future Work	52
3.4 Exports	53
3.5 Extractors	54
3.6 Future Work	55
4 Planned Extension	56
4.1 Traceability of Design Decision Rationale	56

4.1.1	Introduction	56
4.1.2	Tool Support for Rationale Management	57
4.1.3	Tool Support for Traceability of DDRs	57
4.2	GAMBLE: A Generalized Framework for Traceability Link Rejuvenation	58
4.2.1	Introduction	58
4.2.2	General Proposal Overview	59
4.2.3	Concern Graph Topology	62
4.2.4	Proposed Extensible Framework Architecture	62
4.2.5	Specific Approach and Implementation: Pointcut Rejuvenation	64
4.2.6	Conclusion and Future Work	80
	Conclusion	81
	Bibliography	82

List of Figures

1.1	Opening the ATF perspective	14
1.2	Selecting the ATF perspective	15
1.3	Opening the ATF repository wizard	15
1.4	Using the extension wizard	19
1.5	Setting the extractor class	19
1.6	Registering an extractor	21
2.1	Traceability Framework architecture overview	24
2.2	Trace link definition workflow	25
2.3	Trace query and trace view workflow	25
2.4	New Traceability Project creation window	26
2.5	Traceability Project context menu	28
2.6	Execution of a Trace Register instance	29
2.7	Trace Query and Trace View instances	30
2.8	Execution of a Trace Query instance (left) and query results shown in a Trace View instance (right)	31
2.9	Traceability Framework components diagram	33
2.10	Traceability Framework planned workflow	34
2.11	“Black Box” framework instantiation scenario	35
3.1	How to represent trace links? Three graphical representations of hyperlinks: links that relate more than two vertices.	38
3.2	Screenshot of the basic graph view. The pointer is over a link (red), its source artefact is in green and target in blue. Top part gives information on the current node (the link).	39
3.3	Screenshot of the radial graph view. The pointer is over an arte- fact (brighter green), top part gives information on the current node.	40
3.4	Screenshot of the hypergraph visualization. Small circles are artefact (links do not appear as node here), the coloured “sets” are links, source artefacts are green coloured and targets are blue. The pointer is over the artefact numbered 165, top part gives information on this artefact and bottom part is a description of the hyper link numbered 11.	41
3.5	Feature Model Wizard Selection	44
3.6	Empty Feature Model	45
3.7	Starting a new Product Line	45
3.8	Vocabulary Trainer Feature Model	46
3.9	Properties of the Vocabulary Trainer root feature	46
3.10	Feature Model Versioning Wizard	48

3.11 Feature Model Evolution	48
3.12 Providing Feature Implementations	49
3.13 Feature Dependency Mapping	50
3.14 Empty Product Model	51
3.15 Vocabulary Trainer Product Model	52
3.16 Example of the output of native Excel export.	53
3.17 Overview of traceability information provided by configuration management	54
4.1 GAMBLE Phase I: Correlation analysis flowchart.	60
4.2 GAMBLE Phase II: Link rejuvenation.	61
4.3 GAMBLE rejuvenation meta-model.	62
4.4 GAMBLE Phase I planned extensible architecture.	63
4.5 GAMBLE Phase II planned extensible architecture.	64
4.6 Hybrid automobile example.	66
4.7 Speeding prevention aspect.	66
4.8 A new fuel cell class.	67
4.9 Algorithm formalism notation.	68
4.10 Top-level rejuvenation algorithm.	69
4.11 A subset of $CG_{\mathcal{P}}^+$ computed from the motivating example.	70
4.12 Intention graph construction formalism notation.	71
4.13 Intention pattern creation algorithm.	73
4.14 Pattern attribute equations.	77
4.15 The suggestions view of the rejuvenate pointcut tool.	79

Overview

WP4 has to provide a framework for traceability, that means a set of various tools to create, visualize, query or analyse trace links. As such, the purpose of this deliverable is to describe the first version of a traceability framework allowing backward and forward traceability throughout the software product line development process.

In a previous deliverable (D4.1), the principles guiding the creation of this traceability framework were identified as well as a traceability meta model sufficiently general to attend the needs of the other work packages of the project. The implementation of this meta model into a prototype should follow the same principles and be easily adaptable to a wide range of context, from variability traceability in requirement to the source code.

To promote this generality and flexibility we chose to develop our traceability framework as an extension of the Eclipse platform. This offers us several advantages:

- Widely spread platform, known and used by many developers at all stages of a development process.
- A stable and rich platform offering numerous features (editors, project management, graphical interface, etc.)
- Easily updated through the Eclipse update mechanism and distributed update sites
- Flexible extension mechanism through the definition of extension points and plugins.

In this document we describe the current state of our prototype traceability framework and possible future extensions. This prototype is organized as (i) a base repository, (ii) a graphical front-end, and (iii) various plugins that attach to this front-end and provide new functionalities. The base repository handles registration and retrieval of trace links with very basic querying functionalities. On top of this repository was build the graphical front-end that offers several extension points for new trace registers, new queries, and new visualizations. These extensions allow, respectively, to populate the repository with new trace information, retrieve some information of interest from the repository, and present these information to the user. We have developed several of these extensions and will present them here.

Finally, this prototype is only a first version of the traceability framework and served mainly as proof of concept and to help us understand better the needs. We are already working on a new version of this framework based on the

lessons learned from the first version. Planned future extensions will be presented.

The prototype is available as a series of eclipse plugins on the website: <http://ample.holos.pt/updatesite/>. Installation requires the creation of the subclipse¹ update site: http://subclipse.tigris.org/update_1.4.x

¹Subversion plugin for Eclipse

1 ATF: The Base Repository

1.1 Concepts

This section gives a brief overview about common tasks to be performed with the ATF. The underlying metamodel follows the metamodel described in [GKN⁺07] and is not described in this document.

1.1.1 Infrastructure

In addition to the elements described in [GKN⁺07], which form the basic data structure(s) of the ATF, an access infrastructure for manipulation of those basic elements exists. The following elements form this infrastructure:

- **RepositoryManager:** There are a couple of auxiliary components that enable access to the trace repository and manipulation of its content. The central point of entry is the RepositoryManager which is responsible for initialization, connection handling and handling of other manager classes.
- **TypeManager:** The TypeManager is an auxiliary component that exposes services like adding, deleting and managing artefact and link types to the user.
- **ItemManager:** The ItemManager is an auxiliary component that exposes services like adding, deleting and managing artefacts and links and creating and removing relations among them.
- **QueryManager:** The QueryManager is the point of entry for querying the repository content. It offers access to several common, preconfigured queries, enables the creation and execution of queries based on conditions and provides the results of these queries.
- **ExtractionManager:** The ExtractionManager manages extractors - modules that extract actual trace information. Extractors may be registered by users, configured in a proper way and executed to collect trace information. Usually each extractor is specialized on a special aspect of the development stages when designing a product line, *i.e.* finding relations between features and requirements.

1.1.2 Eclipse Integration

The ATF requires an Eclipse installation of version 3.4. It is recommended to use the Eclipse Modeling Tools package [ecl08] as a basis.

Trace Repositories in Eclipse A trace repository has to reside somewhere on a harddisk. The Eclipse Platform provides the concept of projects, which act as container structures for semantically related files. This concept has been adopted to organise trace repositories. Each repository resides in such an Eclipse project. Conversely an Eclipse project must be created in advance to create a trace repository.

In Eclipse, each project has a number of associated Project Natures, indicating the type of the project, how to process certain elements in the project and what to expect as the structure of directories inside the project directory. ATF defines its own nature to indicate that some Eclipse project is actually a project containing a trace repository. The id of the nature is `net.ample.tracing.core.TraceNature`.

The directory structure of a trace project/repository is as follows:

- root directory
 - `.project`
 - `.persistence`
 - ...

The file `.project` contains project information and is created and used by the Eclipse Platform. The file `.persistence` contains the identifier for the persistence manager that is used for the repository.

Configuring Trace Repositories The term configuring repositories refers to the initial setup of a repository with certain artefact and link types. In other words, a configuration defines which artefact types are available inside a repository and which kinds of relations may exist between them. This configuration is done by defining so-called repository profiles.

A profile is described in an XML document. The source for such an XML document is described by an extension point and must be provided upon repository creation. The content of the document is defined by an appropriate XML Schema. An example for such a profile document looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<trp:profile xmlns:trp="http://www.ample-project.net/trp"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ample-project.net/trp profile.xsd ">

  <trp:artefactTypes>
    <trp:artefactType name="requirement"
      uuid="ec96ac35-3559-408a-b3c5-07381da49677">
```

```

    <trp:properties>
      <trp:property name="displayname" value="Requirement"/>
    </trp:properties>
    <trp:inheritance/>
  </trp:artefactType>
  <trp:artefactType name="component"
    uuid="5d09d7f9-50fe-4f2d-bd47-cf5aabb519e1">

    <trp:properties>
      <trp:property name="displayname" value="Component"/>
    </trp:properties>
    <trp:inheritance/>
  </trp:artefactType>
  ...
<trp:linkTypes>
  <trp:linkType name="dependency"
    uuid="a8bfe2ea-4842-4b9a-8a8d-168a8e9b6bcc">

    <trp:properties>
      <trp:property name="displayname.forward"
        value="depends on"/>
      <trp:property name="displayname.backward"
        value="is dependent of"/>
    </trp:properties>
    <trp:inheritance/>
    <trp:scope/>
  </trp:linkType>
  <trp:linkType name="temporalDependency"
    uuid="48be8291-cea5-49d2-9f77-c58a58fdbfc0">

    <trp:properties>
      <trp:property name="displayname.forward"
        value="temporal depends on"/>
      <trp:property name="displayname.backward"
        value="is temporal dependent of"/>
    </trp:properties>
    <trp:inheritance>
      <trp:typeRef uuid="a8bfe2ea-4842-4b9a-8a8d-168a8e9b6bcc"/>
    </trp:inheritance>
    <trp:scope/>
  </trp:linkType>
  ...
</trp:linkTypes>
</trp:profile>

```

1.2 Common Tasks

In the following sections common tasks that need to be carried out when working with a trace repository are explained.

1.2.1 Repository Creation

Creating Repositories Programmatically Each repository is managed by a `RepositoryManager`, which can be used to establish connection to already existing repositories. The following code shows how a repository is created and how a connection is established. The central point of entry into the tracing system is the class `TraceWorkspace`, which provides access to repository managers.

```
// create a new repository manager for repository with the name repo1
TraceWorkspace.getInstance().newRepositoryManager( "repo1" );
```

The code above will create a new Eclipse project and assign it the project nature `TraceProjectNature`. The next step to take is to establish a connection to the repository and initialize it.

```
// get the repository manager for an existing repository with
// the name repo1
RepositoryManager mgr = TraceWorkspace.getInstance()
    .getRepositoryManager( "repo1" );
// the repository is not yet created/initialized
mgr.createRepository( "net.ample.tracing.core.XMIPersistenceManager" );
mgr.initializeRepository(
    "net.ample.tracing.core.DefaultRepositoryProfile"
);
// the persistence of the repository is configured,
// now connect to it
mgr.connectToRepository();
// after connection the repository is ready to be used
```

Note that the repository must be created with the ID of a `PersistenceManager`. At the moment there are two available managers, which are `net.ample.tracing.core.XMIPersistenceManager` and `net.ample.tracing.h2.H2PersistenceManager`. The former persists models in XMI files, while the latter uses a relational database (H2 to be exact). It is recommended to use the H2 provider, the XMI provider will be removed in the future when the H2 provider proves to be stable. In addition, persisting models in XMI does not scale at all, when huge amounts of data must be saved.

The actual initialization is done via a repository profile. This profile contains the default types and their relations. At the moment there is one default profile.

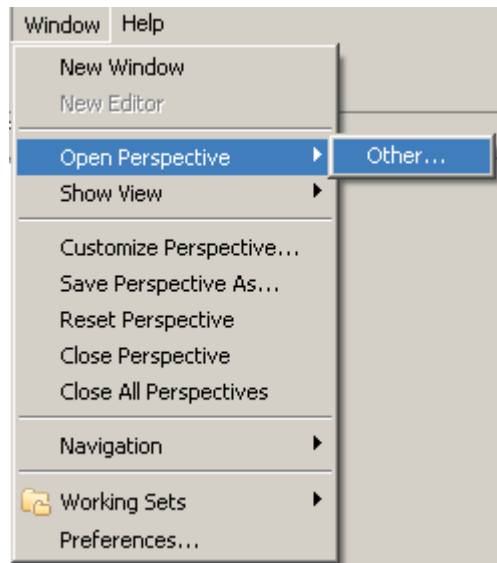


Figure 1.1: Opening the ATF perspective

Creating Repositories via User Interface Repositories can also be created via the user interface. The first step is to open the Trace Perspective, which is provided by the ATF user interface. Select **Window > Open Perspective > Other ...**:

Select Tracing and click OK:

After having opened the perspective it is possible to create a new repository by selecting **File > New > Other ...**

A wizard will open, guiding the user through the process of repository creation. The newly created repository will be displayed afterwards in the Repository Browser-View.

1.2.2 Type Creation

The repository is initialized with a set of predefined artefact and link types. Of course a user may define his own types, which is shown in this chapter. Along with that, this section shows how to commit changes to a repository.

Types may be created directly via the EMF Factory. This is discouraged, because there are some things that need to be initialized. Such initialization tasks are done by a dedicated manager. Therefore it is highly recommended to use the TypeManager for creating types. The type manager can be accessed via its RepositoryManager.

```
// connection to a repository exists ...
RepositoryManager mgr = ...;
// access to the type manager
TypeManager typeMgr = mgr.getTypeManager();
// create an new artefact type
TraceableArtefactType type = typeMgr.createArtefactType(
    "newType", baseType
);
```

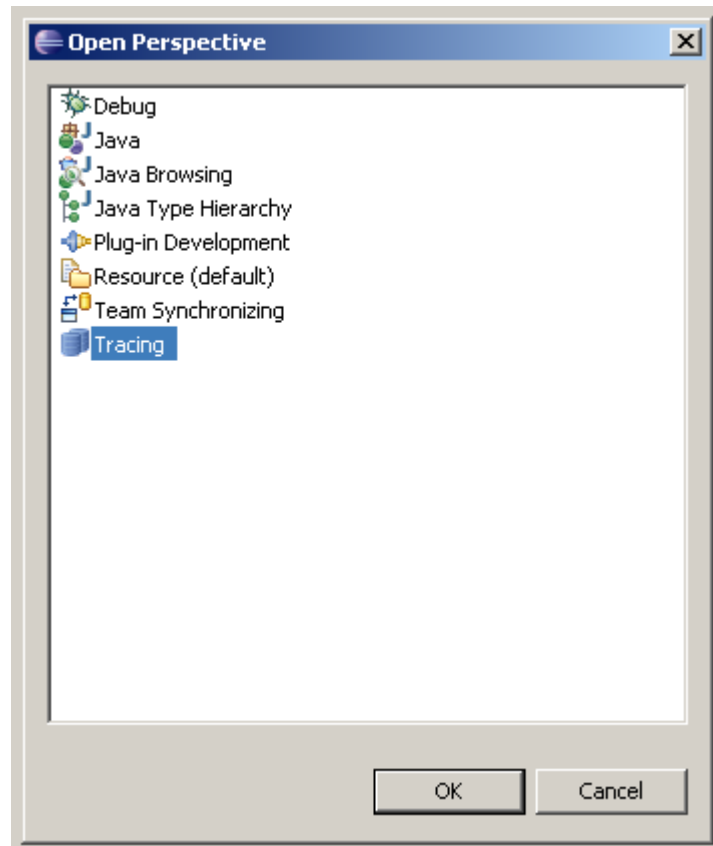


Figure 1.2: Selecting the ATF perspective

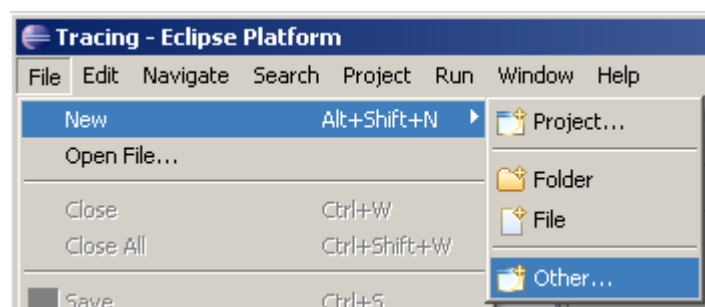


Figure 1.3: Opening the ATF repository wizard

When looking at the create method, one question is obvious: what is the base type and where does a user get one from? Types are structured into a hierarchy. A user is completely free to structure this hierarchy, all types can be user defined. As of version 0.1.5 the common root type has been removed and multiple inheritance has been introduced. For this reason any existing type can be used as a base type. Types may also exist without a base type. At the moment there is no check for cyclic dependencies among types. A user is responsible for not building cycles when creating types and their base types. The creation of link types works the same way.

Persisting Changes to the Repository Creation of types is one thing but of course these types need to be saved. That's why a user needs to persist the changes that have been made to the repository. Responsible for persistence is a so-called PersistenceManager. The interface of a persistence manager mimics the tasks of persistence managers known from database abstraction layers like JDO or JDBC. With that knowledge it is possible to complete the example from above:

```
RepositoryManager mgr = ...;
// access the persistence manager and start a transaction
mgr.getPersistenceManager().begin();
// create a new artefact type without a base type
TraceableArtefactType type = mgr.getTypeManager()
    .createArtefactType( "newType", null );
// mark the object as persistable and schedule it for saving
mgr.getPersistenceManager().add( type );
// commit the changes (causing the repository to persist the changes)
try {
    mgr.getPersistenceManager().commit();
} catch ( IOException e ) {
    e.printStackTrace();
}
```

1.2.3 Artefact and Link Creation

After having read the section about creating artefact and link types, creating artefacts and links should not be a problem for a user anymore, because this task works in exactly the same manner. The only difference is the use of another manager for this. Just as the TypeManager is responsible for creating link and artefact types, there is an ItemManager responsible for creating artefacts and links between them. In the following example two artefacts are created and connected with a link.

```
// a connection to a repository already exists
RepositoryManager mgr = ...;
// access to the type manager
TypeManager typeMgr = mgr.getTypeManager();
```

```
// retrieve the types for the artefacts/links from type manager
// each type is identified with a unique identifier,
// that can be used for retrieving the type
TraceableArtefactType artType = mgr.getTypeManager()
    .findArtefactTypeByUuid( ... );
TraceLinkType linkType = mgr.getTypeManager()
    .findLinkTypeByUuid( ... );
// create two artefacts
TraceableArtefact art1 = mgr.getItemManager()
    .createTraceableArtefact( artType, "artefact1" );
TraceableArtefact art2 = mgr.getItemManager()
    .createTraceableArtefact( artType, "artefact2" );
// connect both artefacts with a link
TraceLink link = mgr.getItemManager()
    .createTraceLink( art1, art2, linkType );
```

The transaction mechanism shown here is also valid when creating artefacts and links. Otherwise the changes to the repository will not be persisted.

1.2.4 Artefact and Link Removal

In the previous section about artefact and link creation, it has been explained in what way artefacts and links are created and stored in the persistence layer. Of course, one may also remove them at a later point in time. As a prerequisite there are references to the links and artefacts that are to be removed. This example shows how to do the removal programmatically:

```
RepositoryManager mgr = ...
//
TraceableArtefact artefact = ...;
TraceLink link = ...;
// remove the link
mgr.getPersistenceManager().begin();
mgr.getPersistenceManager().remove( link );
mgr.getPersistenceManager().commit();
// remove the artefact
mgr.getPersistenceManager().begin();
mgr.getPersistenceManager().remove( artefact );
mgr.getPersistenceManager().commit();
```

Removing the link in the way shown above will disconnect the link from its adjacent artefacts and delete it from the underlying database. Removing the artefact does more than that: because the adjacent links of the artefact to be removed cannot exist any longer, they are removed too. Therefore, the removal operation of a link will touch the link itself only, while the same operation on an artefact will affect the artefact and all of its neighbours.

1.2.5 Collecting Trace Information

In this section it is explained how to collect actual trace data. As a prerequisite the reader should be familiar with Eclipse plugin development and the extension point mechanism Eclipse provides.

One way to fill the trace repository is to use the access methods as described in the previous sections about types and links and artefacts directly. Another way is to create a dedicated trace extractor and plug into the ATF via an extension point, which in the end allows to run the extractor from the user interface.

The first step for plugging into the ATF is to create an Eclipse plugin. After creation the plugin manifest needs to be opened and the dependency to the ATF core plugin needs to be added. Then the user needs to go to the tab Extensions and add a new extension with the id `net.ample.tracing.core.TraceExtractor`.

The next step is the configuration of this extension. This is mainly to enter the class which implements the extractor as shown below.

Of course the hardest part of the work comes afterwards: the implementation of the extractor. As an example it is assumed that one needs to trace dependencies among Eclipse plugins. Plugins are artefacts and the dependencies are links. The source code below shows how to implement such an extractor:

```
// derive class from AbstractTraceExtractor
public class MyExtractor extends AbstractTraceExtractor {

// define the UUIDs of artefacts and links as static fields,
// the UUIDS have set when defining the appropriate types
private static final UUID UUID_PLUGIN = UUID.fromString(
    "a670361d-72c7-4f04-8d01-d5d6eb55131e"
);

private static final UUID UUID_FRAGMENT = UUID.fromString(
    "c63e91f6-c1b6-4164-9979-d19050c00999"
);

private static final UUID UUID_DEPENDENCY = UUID.fromString(
    "41399a05-37ad-44f0-8422-66bb32c11a7c"
);

// the only method that needs to be implemented is the run()-method
public void run( RepositoryManager manager, IProgressMonitor monitor ) {
// retrieve the types used for artefact and link creation
TraceableArtefactType pluginType = manager.getTypeManager()
    .findArtefactTypeByUuid( UUID_PLUGIN );
TraceableArtefactType fragmentType = manager.getTypeManager()
    .findArtefactTypeByUuid( UUID_FRAGMENT );
TraceLinkType dependType = manager.getTypeManager()
    .findLinkTypeByUuid( UUID_DEPENDENCY );
// access the plugin base of the Eclipse platform
IPluginModelBase[] base = PluginRegistry.getAllModels();
```

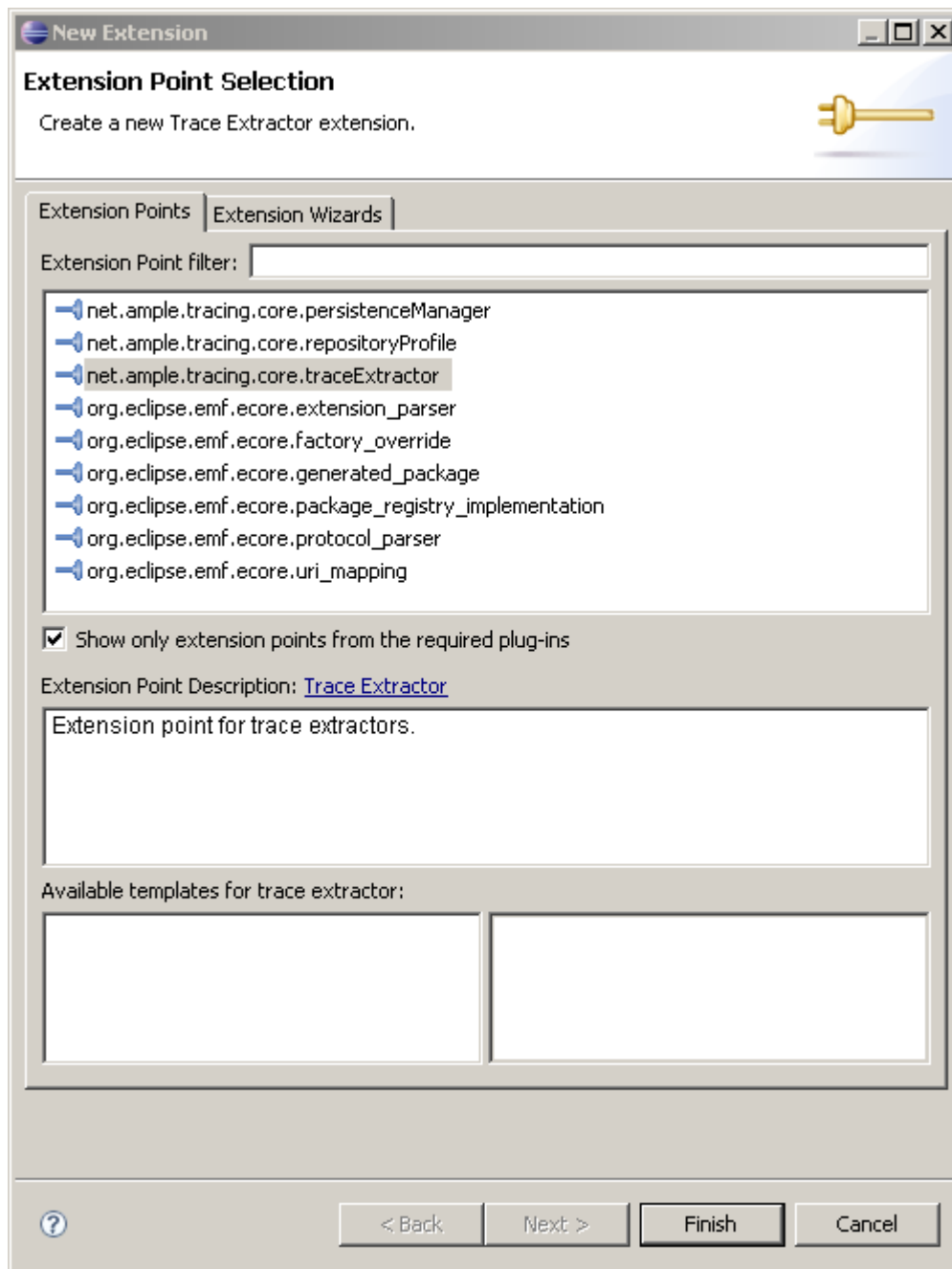


Figure 1.4: Using the extension wizard

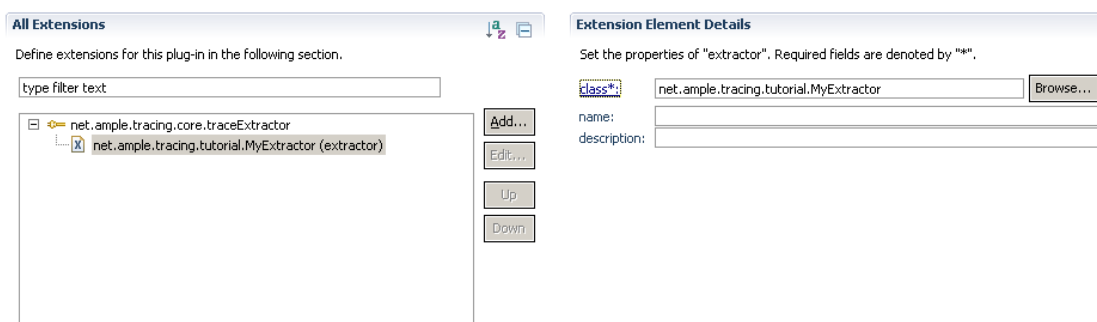


Figure 1.5: Setting the extractor class

```
// create a hashmap for new artefacts
HashMap<String , TraceableArtefact> newArtefacts =
    new HashMap<String , TraceableArtefact>();
// open a transaction
manager.getPersistenceManager().begin();
// look for artefacts
for ( int i = 0; i < base.length; i++ ) {
// create a new artefact for each plugin from the plugin base
TraceableArtefact artefact = null;
if ( base[i].isFragmentModel() ) {
artefact = manager.getItemManager()
    .createTraceableArtefact(
        fragmentType, base[i].getPluginBase().getId()
    );
} else {
artefact = manager.getItemManager()
    .createTraceableArtefact(
        pluginType, base[i].getPluginBase().getId()
    );
}
// mark the newly created artefact as to be saved
manager.getPersistenceManager().add( artefact );
// remember the id of the plugin and the artefact
newArtefacts.put( base[i].getPluginBase().getId(), artefact );
}
// look for links between the artefacts
for ( int i = 0; i < base.length; i++ ) {
// access the remembered artefact for a plugin
TraceableArtefact plugin = newArtefacts.get( base[i]
    .getPluginBase().getId() );
// access the dependencies
IPluginImport[] imports = base[i].getPluginBase().getImports();
// remember the dependent artefacts
HashSet<TraceableArtefact> importedPlugins =
    new HashSet<TraceableArtefact>();
for ( int j = 0; j < imports.length; j++ ) {
TraceableArtefact artefact =
    newArtefacts.get( imports[j].getId() );
if ( artefact != null ) {
importedPlugins.add( artefact );
}
}
// if there are dependencies: create a link
if ( importedPlugins.size() > 0 ) {
TraceLink link = manager.getItemManager().createTraceLink(
    plugin, importedPlugins.toArray(
        new TraceableArtefact[importedPlugins.size()]
    ),
    dependType
);
}
```

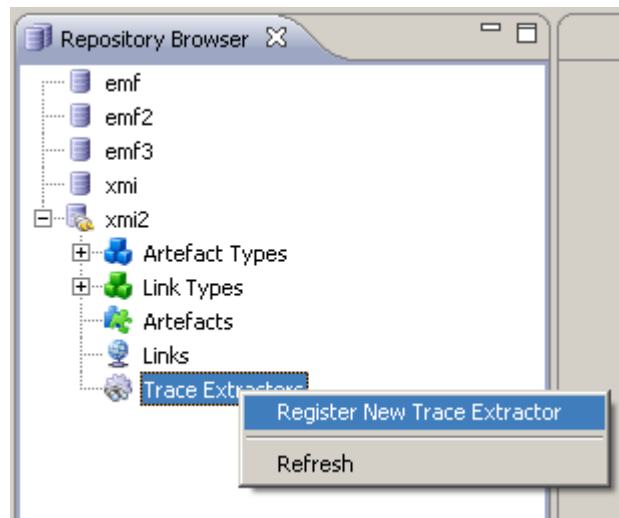


Figure 1.6: Registering an extractor

```

);
// mark the newly created link as to be saved
manager.getPersistenceManager.add( link );
}
}
// finally commit all changes to the repository
try {
manager.getPersistenceManager().commit();
} catch ( IOException e ) {
e.printStackTrace();
}
}
}
}

```

The extractor has been plugged into the extension point concept of Eclipse (and the ATF) and the implementation has been written. Now it is possible to run the extractor. A new Eclipse runtime environment needs to be created including all ATF plugins and the newly created plugin. Then, a new repository needs to be created and a connection to it needs to be established. In the repository browser there are several subcategories, one of them is Trace Extractors: right-click the category and choose Register New Extractor.

In the following dialog it is possible to select the extractor and register it to the repository. Afterwards one may right-click the extractor in the UI and run it.

1.2.6 Querying for Artefacts and Links

In the previous sections the reader has learned how to create artefacts and links, but this is of course just one side of the story. After persisting these things there must be a way to gain access to the data. This section takes a closer look at the query mechanism of the ATF.

As of version 0.1.5 of the ATF an instance of TraceRepository reflects the runtime view while the data in the underlying database contain the whole trace graph. For being able to work with artefacts and links, they must be loaded from the database into the repository. This is done by querying for data.

Querying the underlying database is naturally closely related to the persistence layer. For this reason a low level query mechanism is embedded into the persistence layer. The source code below shows how to create and run queries:

```
// access to a repository via its manager
RepositoryManager mgr = ...
// access the persistence manager and create new query
Query<TraceableArtefact> q = mgr.getPersistenceManager()
    .query( TraceableArtefact.class );
// narrow the scope of the query with constraints
q.add( Constraints.name( "myArtefact" ) );
// concat constraints
q.add( Constraints.name( "name1" )
    .add( Constraints.artefactType( someType ) );
// finally execute the query to get results
List<TraceableArtefact> result = q.execute();
```

2 Graphical Front-End

On top of the traceability metamodel described in Chapter 1, we have developed a traceability framework that aims to provide an open and flexible platform to design and implement tools and methods that allow software developers to define and store trace links between the different artifacts used during SPL development. Our framework uses the services provided by ATF and can be seen as a high-level API for defining trace links and querying the trace information that is stored in ATF.

2.1 Current Structure

An architectural overview of the framework is shown in Figure 2.1 as an UML component diagram, where the four main modules and their relationships are depicted. The framework has three hotspots that need to be instantiated to provide trace mechanisms for each desired scenario or stage of development. The ATF module is based on the traceability metamodel described in the previous deliverable D4.1 [GKN⁺07].

Inside the “Framework Core” package resides three hotspots of the framework: *TraceRegister*, *TraceQuery* and *TraceView*. *TraceRegister* instances are used for performing CRUD (create, read, update and delete) operations on the artifacts and links stored in ATF. This can be done using fully automatic techniques, or by providing a GUI for manual definition and maintenance of the trace information, or a combination of both. *TraceQuery* instances provide means to perform specific queries on a set of trace links. It uses the basic query capabilities (trace links and trace artifact retrieval) of ATF to execute more complex and powerful queries, like feature interaction detection and change impact analysis. Finally, *TraceView* instances are responsible for supplying some sort of view (graphical, textual, etc.) for the results returned by a trace query execution.

On top of the “Framework Core” package lies the “Framework Manager” package. This package contains all the classes that are necessary for loading the instances that are provided for each hotspot. It is composed of a *PluginsHandler* which browses all the available instances and filters the ones of interest. Finally, the *RegisterLoader* and *QueryViewLoader* simply load the chosen instance and execute the desired methods.

The workflow for defining new trace links using a register is shown in Figure 2.2. The user first begins by populating the ATF repository with artifacts extracted from the source models (e.g., feature model, use case model, source code). Once that step is concluded, the selected Trace Register instance is executed which will be responsible for creating the trace links between the artifacts

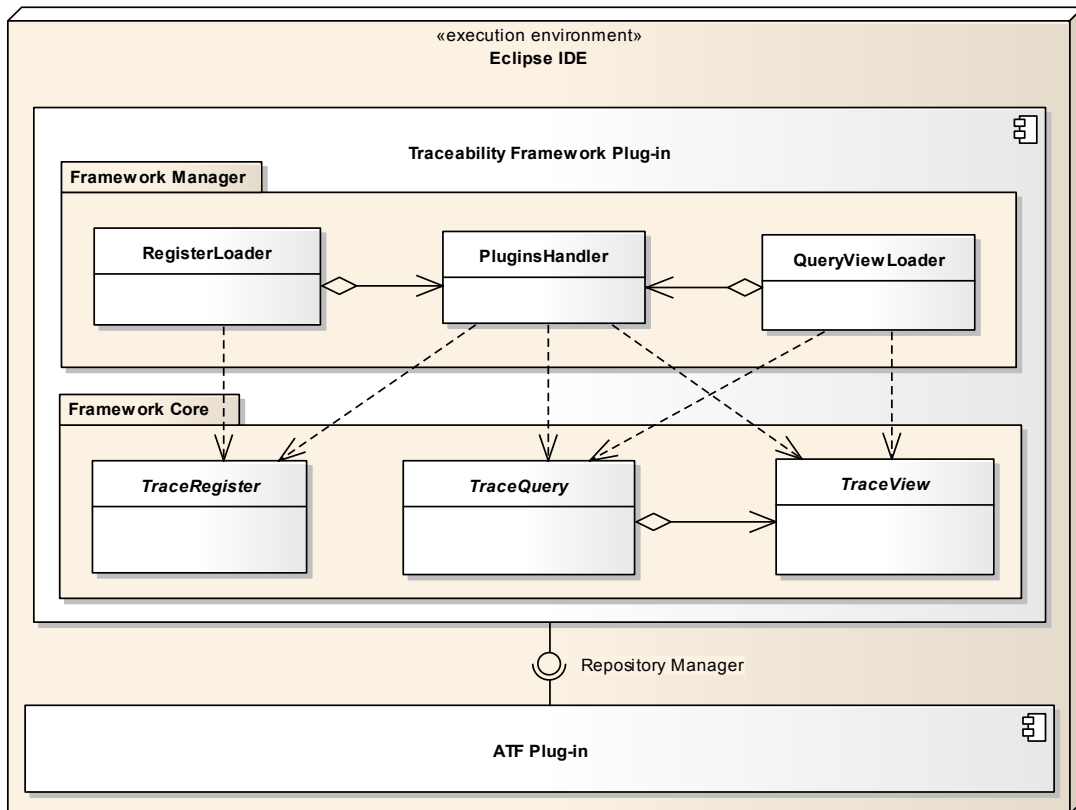


Figure 2.1: Traceability Framework architecture overview

residing in ATF. As mentioned previously, this step can be automatic, manual or a combination of both.

Figure 2.3 depicts the workflow for executing trace queries. The user begins by selecting a Trace Query instance to execute. The next step is choosing the query parameters, if any exist (e.g., selecting the type of artifacts to be queried). The chosen Trace Query will then retrieve the relevant links and artifacts from ATF and pass them to the chosen Trace View for visualization.

2.2 Common Tasks

2.2.1 Creating a Traceability Project

New “Traceability Projects” can be created via the user interface. A traceability project will contain a file with all the necessary configurations to run instances of the Traceability Framework. This functionality can be executed by selecting the menu option *File > New > Other...* and then choosing the new **Traceability Project** option (Figure 2.4). A wizard will guide the user through the process of a new traceability project creation. One must choose the traceability project file, the name of the ATF repository to use, and the desired extractors.

After pressing the finish button, a new ATF repository is created (with the chosen name), and initialized with several new types of trace links and trace-

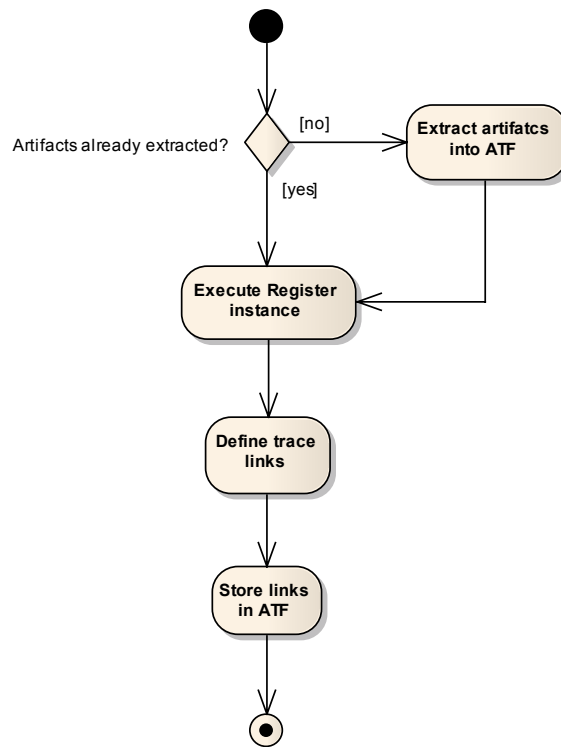


Figure 2.2: Trace link definition workflow

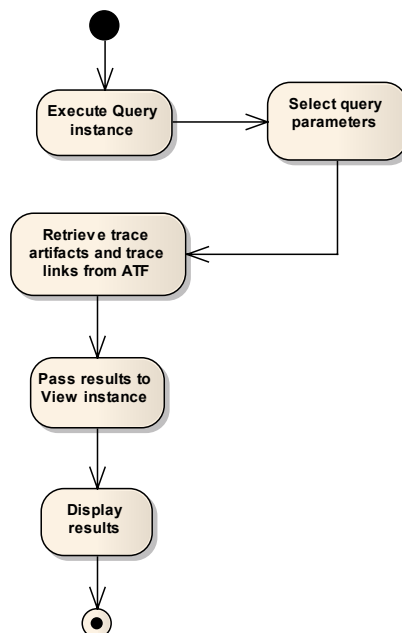


Figure 2.3: Trace query and trace view workflow

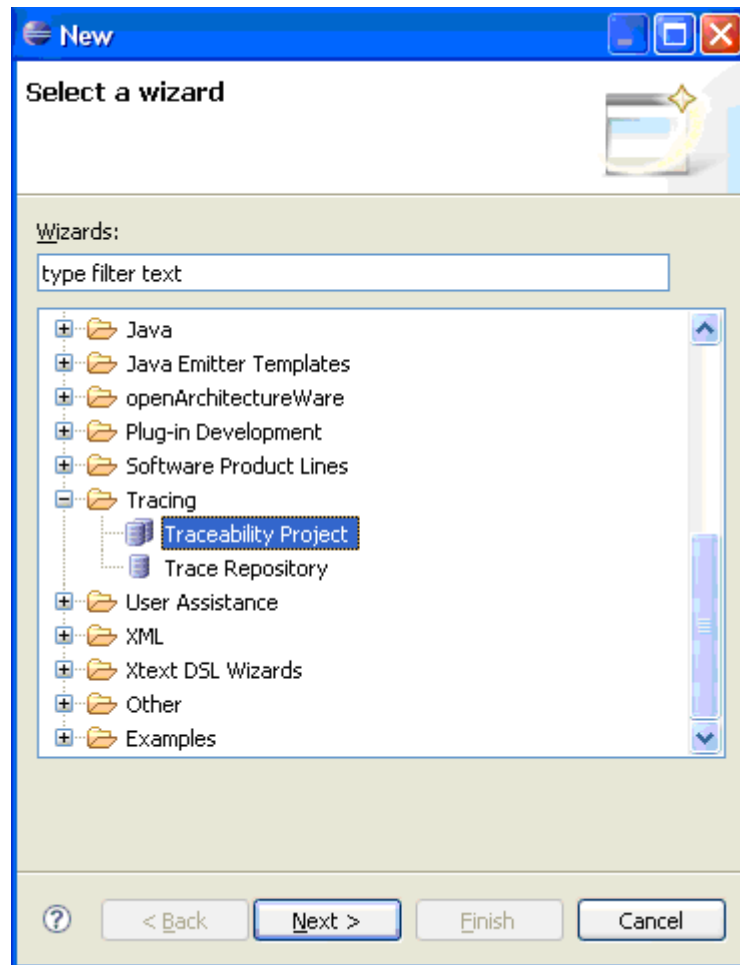


Figure 2.4: New Traceability Project creation window

able artifacts. Once the project is created a popup menu will be available for the traceability project file. To access it, one right clicks on the project file and choose the desired action (Figure 2.5). Depending on the purpose of using the traceability framework, different kinds of trace links and traceable artifacts can be selected to be manipulated. Examples of existing types of trace links supported by our framework are: Relationship and Hierarchy. Relationship is used to create a link between two abstractions from different models that specify the same SPL. Hierarchy links define hierarchical information between two abstractions of SPL models.

One of the existing instances of the framework was developed to support the specification of the requirements and features of a SPL during the domain analysis stage. The types of traceable artifacts provided by this instance are Feature, Use Case, Step, Actor and Package. Information about these artifacts can be imported from existing Feature and UML models that specify the SPL requirements and features. Additionally, this instance also allows to specify: (i) Relationship trace links that connect a feature (from the variability model) to an Actor/Use case/Step/Package (from the requirements model); and (ii) Hierarchy trace links that is used, for example, to indicate that a step belongs to its parent use case.

2.2.2 Definition of new Trace Links

To define new trace links, we must initialize a trace register instance. Choose **Initiate Trace Register** from the popup menu of the traceability project file, and then choose the desired trace register instance from the list of available trace registers.

In a specific trace register instance that we have already implemented, it is possible to define the desired trace links between features and requirement artifacts by checking the corresponding boxes. Figure 2.6 shows the dialog implemented for this trace register. As we can see, the box labeled “Display Photo for Incoming Call” that represents a use case is checked. It means that this use case is related to the “Incoming Call feature”. When the **Save** button is pressed the changes performed in the trace register window are committed to the ATF repository. For the example mentioned, it means that a trace link between that feature and that use case step will be created in the repository.

2.2.3 Submitting Queries and Viewing Results

To submit a query to the ATF repository, choose **Submit Query** from the popup menu for the traceability project file, and then choose a trace query instance and a trace view from the list of available extensions (Figure 2.7).

Next we must choose which model elements are to be queried. Once the choice has been made, click the **Submit Query** button and the results will be displayed in the chosen view. Figure 2.8 shows an instance of a trace query and trace view implemented to support the traceability between features and

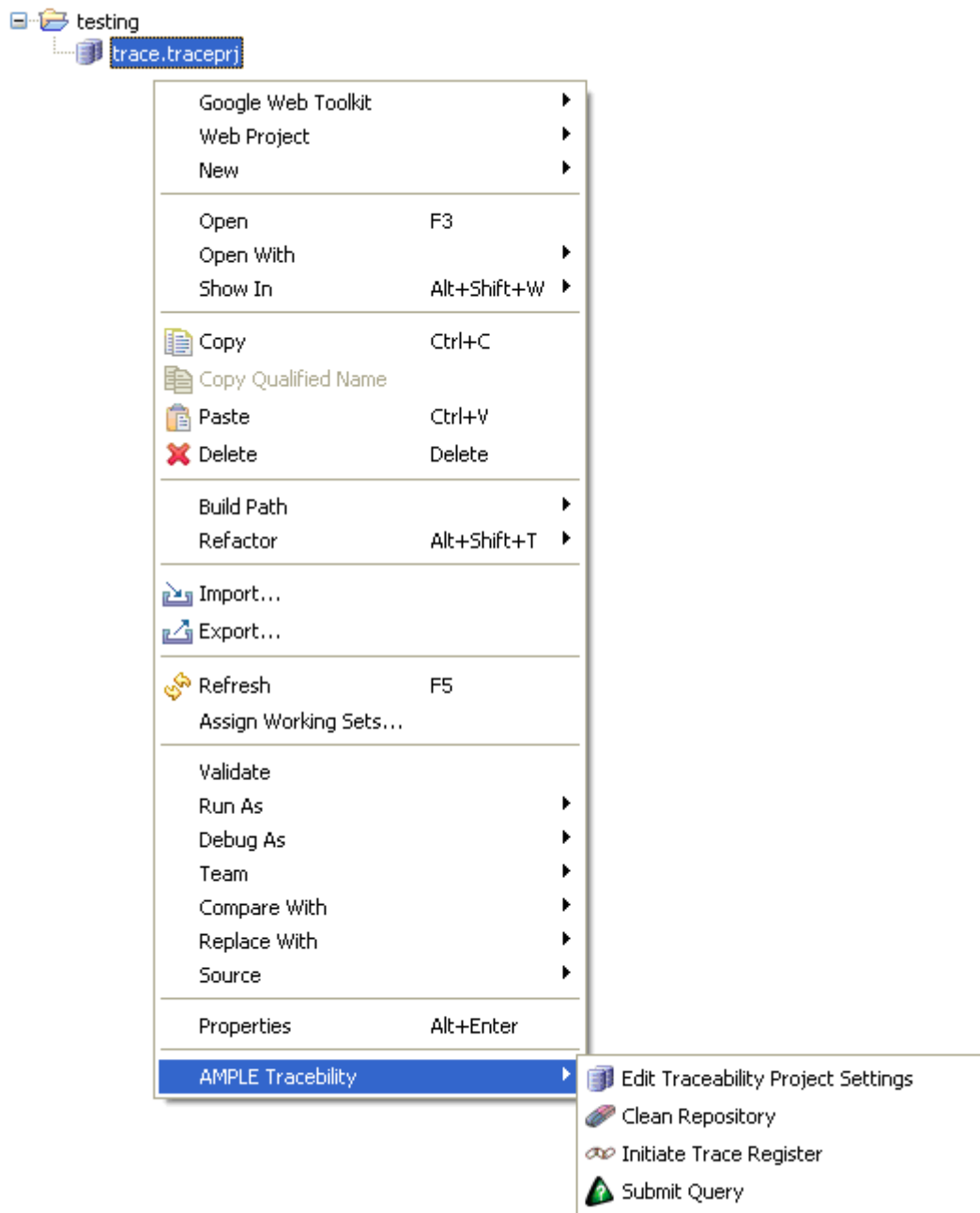


Figure 2.5: Traceability Project context menu

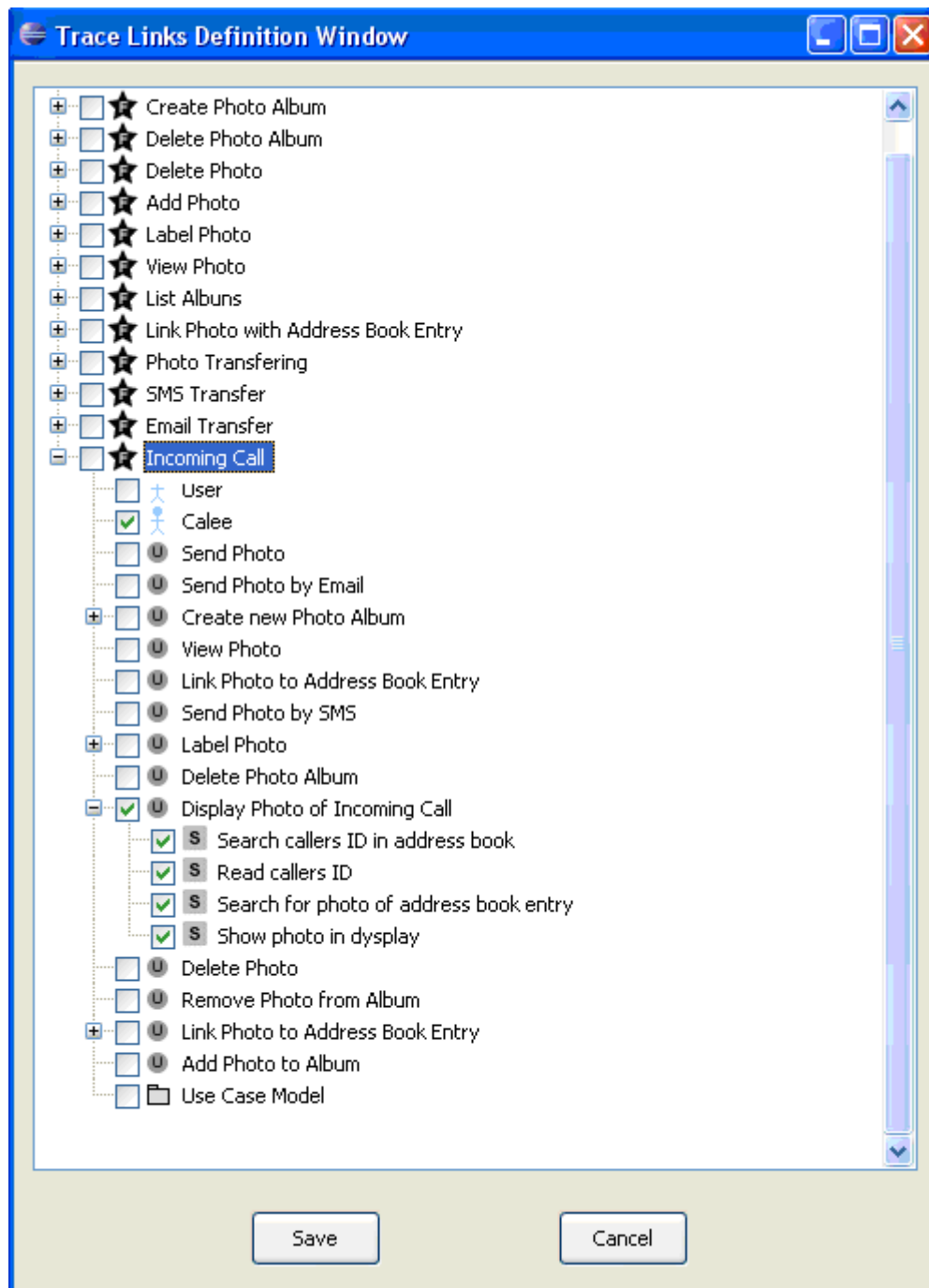


Figure 2.6: Execution of a Trace Register instance

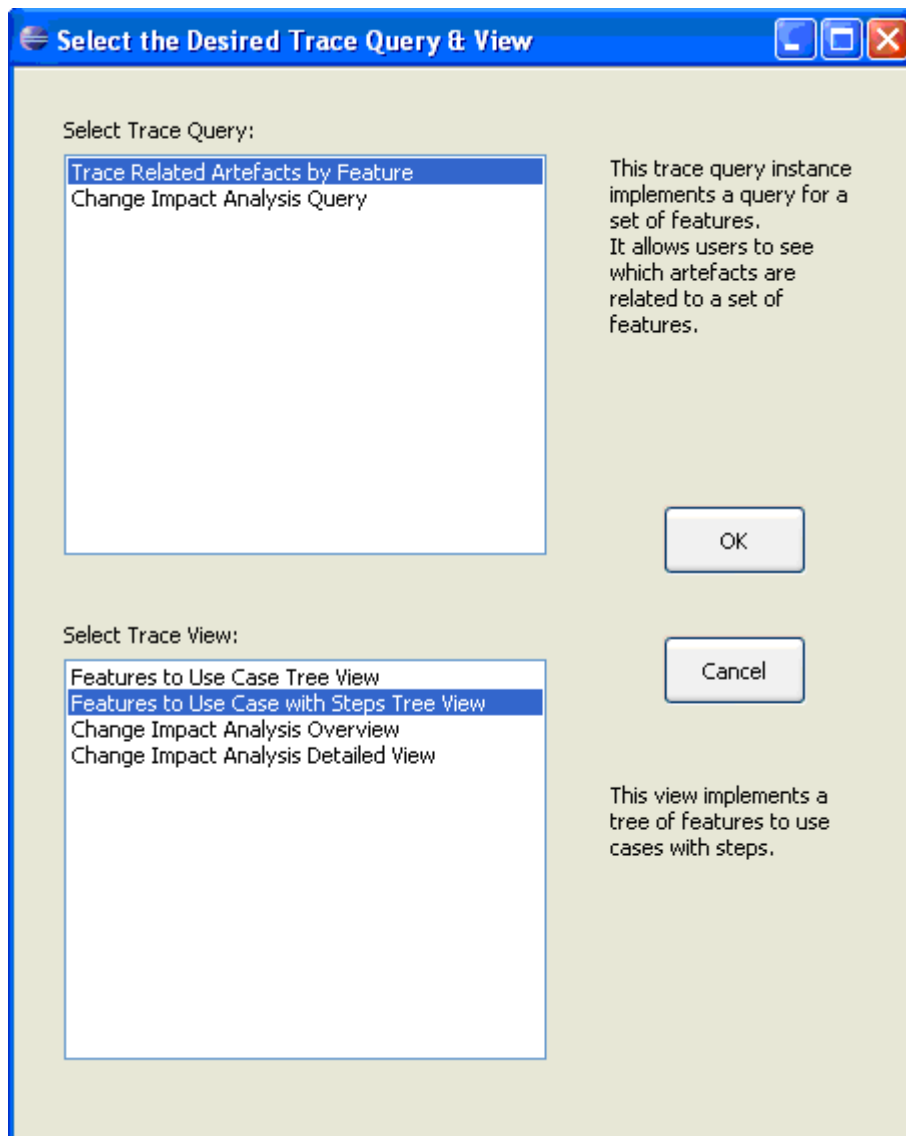


Figure 2.7: Trace Query and Trace View instances

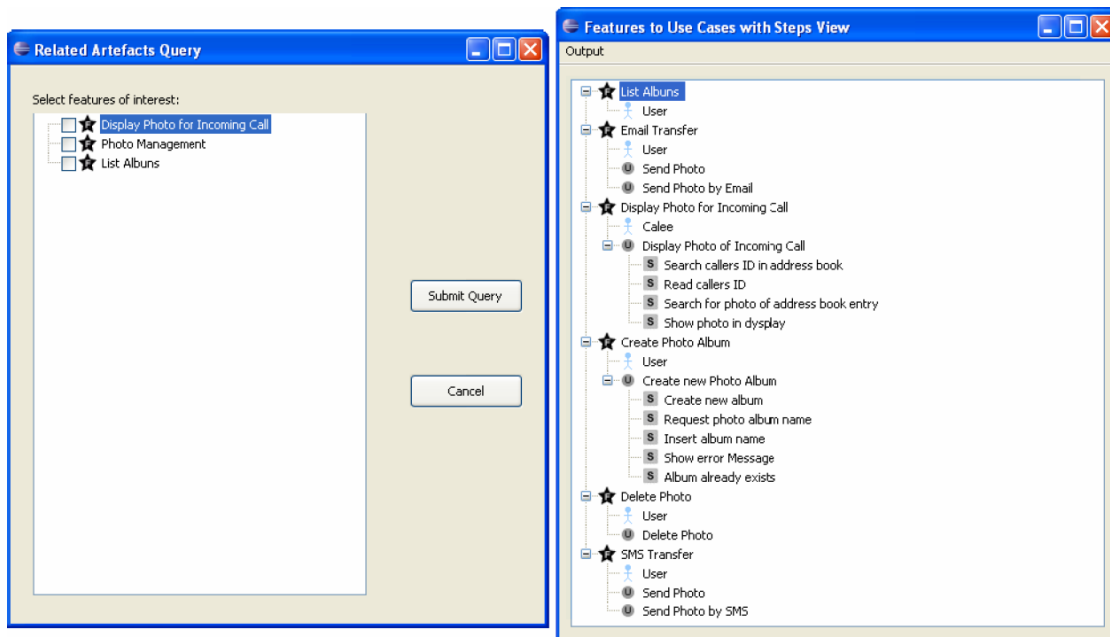


Figure 2.8: Execution of a Trace Query instance (left) and query results shown in a Trace View instance (right)

use cases. On the left side of the figure, we can see that the user can choose the specific features that he/she is interested to trace. The right side of the figure shows the trace view between features and use cases that allows illustrating the trace links between those elements.

2.3 Extension Mechanism

To achieve the goal of allowing the instantiation of the framework in different scenarios, an Eclipse plug-in with the required extension points was implemented. This mechanism allows framework developers to implement new instances of each hotspot by implementing an Eclipse plug-in that extends the extension points from the plug-in that represents the traceability framework. It allows adapting the framework to the desired scenario.

Figure 2.9 shows a UML component diagram with the extension points provided by the framework (represented using component ports) and the base implementation for each extension point (represented by an abstract class). Each extension point has an abstract class that is used to provide a base implementation of all the methods that are common to all instances. The developer of an extension is only required to implement the code that is specific to a particular instance (*e.g.*, the method that performs the necessary operations for implementing the feature interaction query). A list and description of the already implemented extensions is given in Chapter 3.

The current extension points are:

- **net.ample.tracing.framework.core.traceRegister:** This extension point is used to plug in additional trace registers for establishing trace links be-

tween SPL artifacts. The base implementation is provided by the *AbstractTraceRegister* class.

- **net.ample.tracing.framework.core.traceQuery**: This extension point is used to plug in additional trace queries for performing new types of queries demanded by specific traceability scenarios. Its based implementation is offered by the *AbstractTraceQuery* class.
- **net.ample.tracing.framework.core.traceView**: This extension point is used to plug in additional trace views. Its based implementation is codified by the *AbstractTraceView* class.

Another important point is the ability to add trace extractors to ATF to populate the repository with trace artifacts and/or links. The extension point:

net.ample.tracing.core.traceExtractor (shown as a component port in Figure 2.9) can be used to this end. By implementing an extractor that parses a source model (e.g., use case model modeled in Rational Rose or Enterprise Architect) we can populate the repository with the artifacts extracted from the input model and on a later step use a *TraceRegister* instance to define the trace links between the imported elements. Another option is to extract the trace artifacts and trace links in a single step. Whatever is the path chosen, a trace register can be used to perform maintenance of trace artifacts and links.

Figure 2.9 also shows how the “Framework Manager” is used to load the framework instances. Each framework hotspot (extension point) provides an abstract class and an interface. The *PluginsHandler* class searches the entire scope of Eclipse Plug-ins to find the ones that implement extensions to the desired extensions points (“net.ample.tracing.framework.core.traceRegister”, “net.ample.tracing.framework.core.traceQuery” and “net.ample.tracing.framework.core.traceView”), and passes that information to the *RegisterLoader* or the *QueryViewLoader* which use the provided interfaces to load the chosen hotspot instance.

2.4 Evolution of the Traceability Framework

There are plans to evolve the current version of the existing Traceability Framework. One of the improvements under development is to implement a new UI for the framework, that will make use of Eclipse views and editors to provide a flat view of the available functionalities (in contrast to the current popup menus).

The second improvement is to develop a new workflow for queries and views. The proposed workflow is depicted in Figure 2.10 where we can see the most important steps for performing queries on the trace information. The idea is to perform a query, and refine the query results (by resubmitting a new query which will return a refined set of trace links or artifacts) until the desired information is reached. The idea is to perform a round-trip between the queries and the views. The query results are passed to a view, which in turn will allow the user to select a set of trace links or artifacts and execute a new query with that selection. The

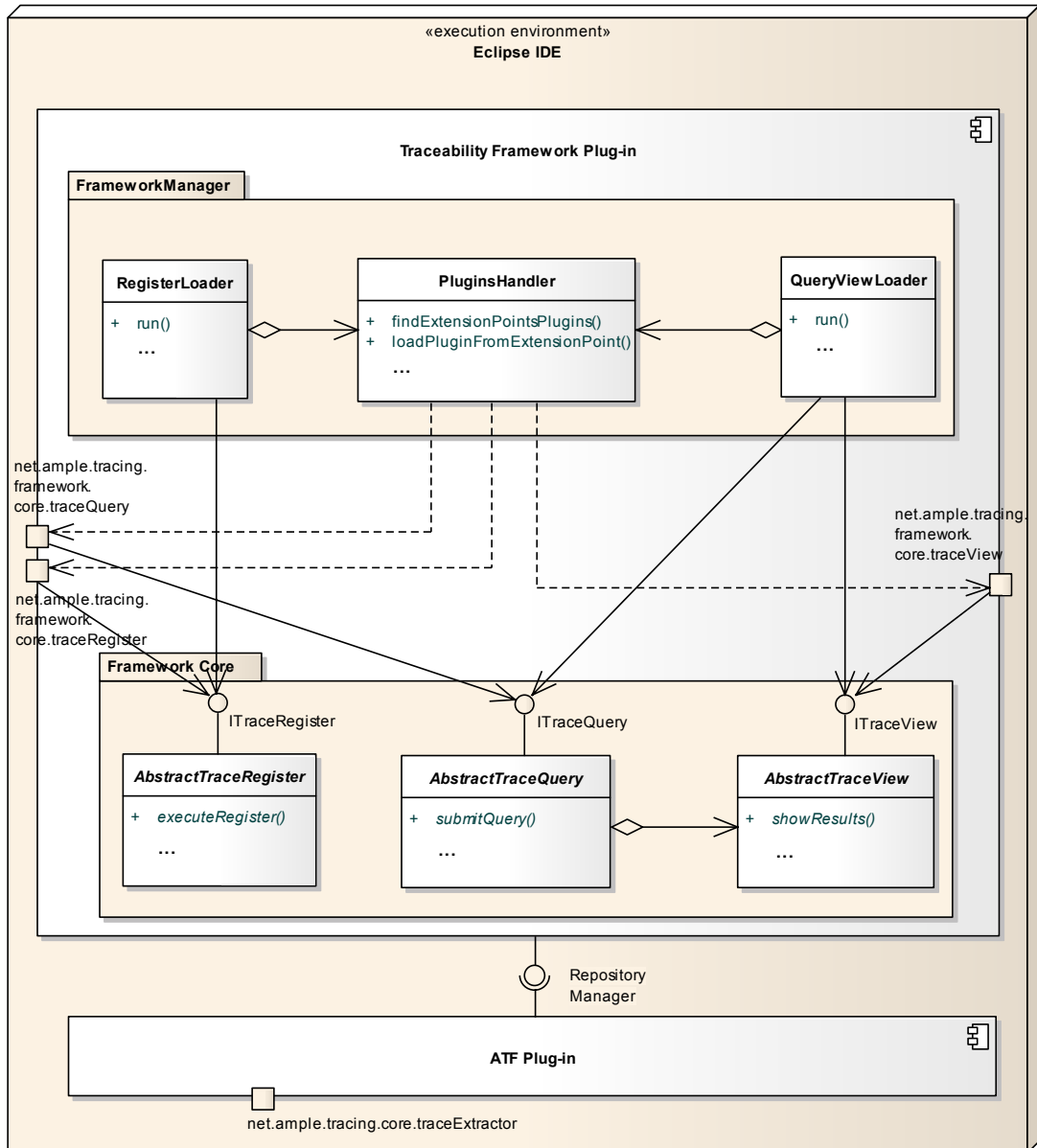


Figure 2.9: Traceability Framework components diagram

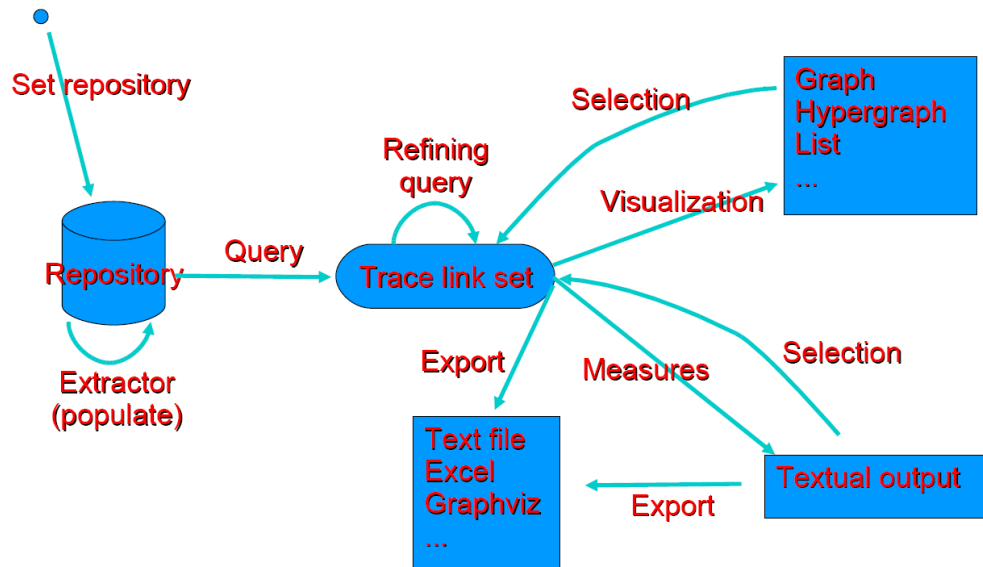


Figure 2.10: Traceability Framework planned workflow

results may also be exported to several formats, by using special views for that purpose.

The final improvement that is under consideration is to provide a new development mechanism for framework instantiation. We plan to implement a “Black box” instantiation environment. In the current implementation, a developer must be aware of all the mechanisms required to implement a hotspot instance (*e.g.* the framework architecture and APIs, the Eclipse graphical library). We are planning to implement generic instances for each hotspot that will provide a default implementation for the common aspects for several instances. For instance, we can develop a trace register using a matrix to define trace links between the elements. This generic instance could be then instantiated to different scenarios (*e.g.* features to use cases, features to source code). The “Black box” instantiation scenario is shown in Figure 2.11

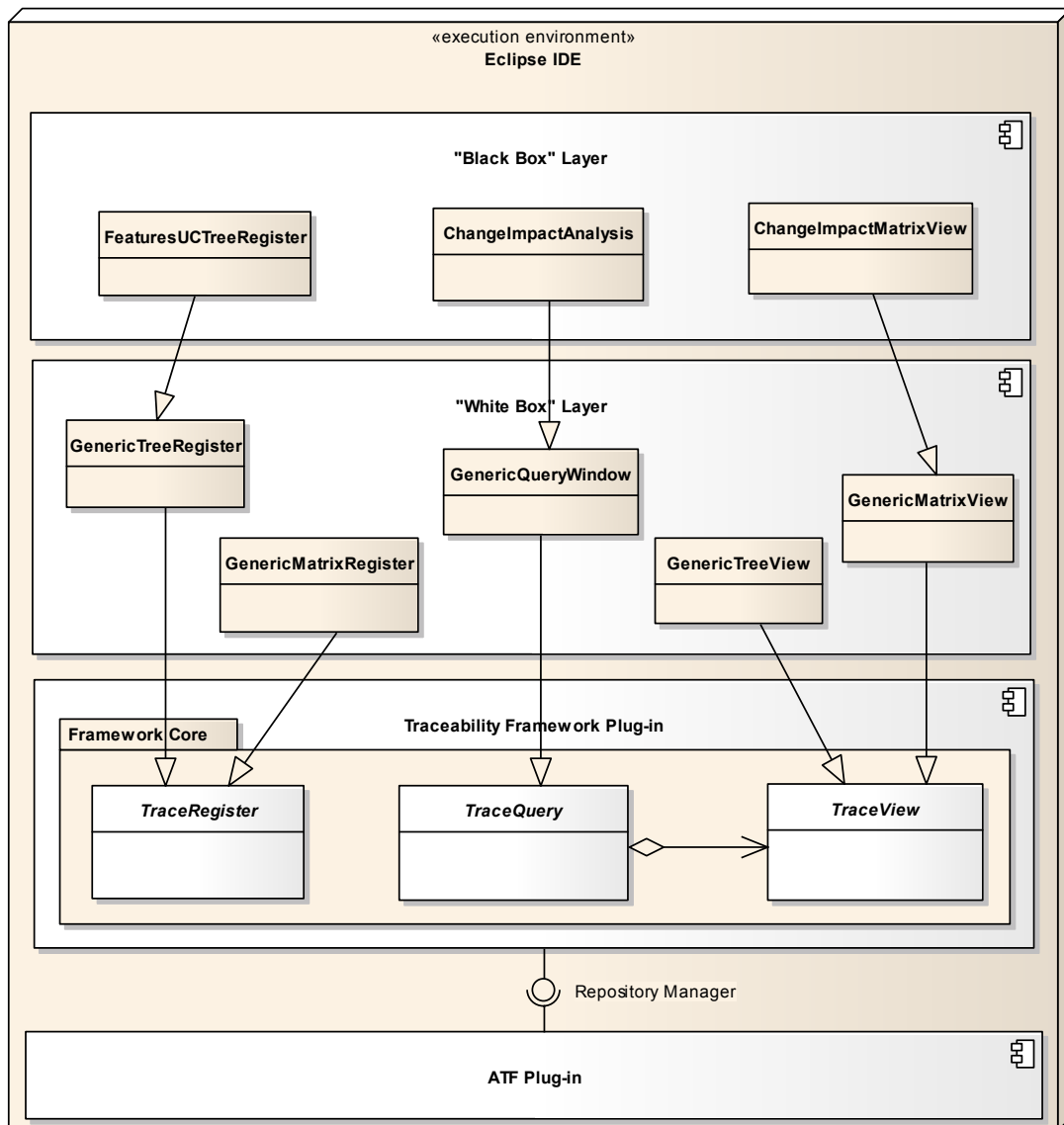


Figure 2.11: "Black Box" framework instantiation scenario

3 Implemented plugins

To help software engineers manipulate and view traceability links, we have implemented several visualization and export tools. These tools were implemented first in an independent graphical front-end on top of the trace repository. In a second phase, they were integrated with the graphical front-end of the AM-PLÉ project as plugins. Plugins are features that can be added to the traceability framework to perform predefined activities (see Chapter 2).

We present here the tools as they were integrated in the traceability framework. We discuss the topic of graphical representation of trace information and propose different answers to this problem. We will also present some metrics on traces and the export facilities.

3.1 Graph representation

We implemented visualization tools that see the traceability information as a graph. Our hypothesis is that a well-designed visualization tool uncover some structures in a graph that would be difficult or impossible to see in another form. Graph visualization is an old preoccupation of computer science and there are many tools (graphviz, prefuse, jung), conferences (e.g. <http://graphdrawing.org/index.html>) and web sites (<http://www.caida.org/tools/visualization/walrus/>, <http://jung.sourceforge.net/>, <http://www.jgraph.com/>, <http://prefuse.org/>, and <http://www.graphviz.org/About.php>).

3.1.1 Hypergraphs

The decision was made early in the project to represent traceability links as n -ary relations between artefacts. For example the realization traceability link may relate one software requirement to several design artefacts, all packaged in one or two components (see example in Figure 3.1, top). Trace information, therefore, may be modelled by an hypergraph, a graph were links may relate more than two vertices [Ber70] (see also <http://en.wikipedia.org/wiki/Hypergraph>). Unfortunately there are few tools to represent and manipulate hypergraphs. We could only find a demonstration from a package called prefuse (<http://prefuse.org/>) which can be used as a basis. Recently another example was using the Jung package.

Although graphs have a very natural and intuitive graphical representation (points linked by lines), hypergraphs may be more challenging. There are three possible representations for traceability links (pictured in Figure 3.1):

1. The first representation (top of the figure) is the most intuitive, but it is complex to render graphically, especially when there are many vertices scattered over the surface (for example imagine “class4” and “class5” on the left of the requirement).
2. The second representation (middle of the figure) is a different view of the first one. It used “sets” to represent the links. This is the same as a Venn Diagram (http://en.wikipedia.org/wiki/Venn_Diagram). It can look cluttered, and directed links are not easy to represent.
3. Finally, the third representation (bottom of the figure) is based on a different kind of graph. The links are promoted to a new kind of vertices (or nodes) turning the hypergraph into a bipartite graph (graphs with two different kinds of vertices and vertices of one kind are only related to vertices of the other kind). It is less intuitive, but would also suffer less from the problem of scattered vertices. The benefit is to be able to use the numerous tools, measures and theory that exist for this kind of graph.

3.1.2 Overall Design

The design principles are simple, the plugins work on a list of traceability links and their related artefacts. Conceptually one could say they work on the repository's content, however, for efficiency purpose, we use internal memory to cache the data. The data are provided by the graphical front-end, through queries. These queries return a set of traceability links (normally a subset of the entire repository) and artefacts. The informations we exploit are `name` and `type` for artefacts, and `source` artefacts, `target` artefacts and `type` for a link (note: the trace links are not named). Other attribute could be used, but this would be less generic and will be left to more advanced users of the traceability framework.

3.1.3 The Graph Visualization Plugins

We implemented three plugins that exemplify the second and third representation of hypergraphs (see 3.1.1). These plugins were implemented using the `prefuse` library (<http://prefuse.org/>) packaged as a `.jar` file. The `prefuse` library allows to define and visualize graph, its main characteristic compared to many other graph tools is its ability to interact with the graph view. Thus one could imagine displaying a set of trace links and artefacts, selecting some of them and use that as a new subset of trace links to work on. In this sense, the visualization could be viewed as an interactive query returning a set of trace links.

Other interesting features of `prefuse` is that it allows to drag (or pan) the entire graph, drag a specific node, search for some string (*e.g.* a node's name) or zoom in and out. It also detects the position of the pointer (mouse), so that we configured it to show details of artefact and link nodes when the pointer is

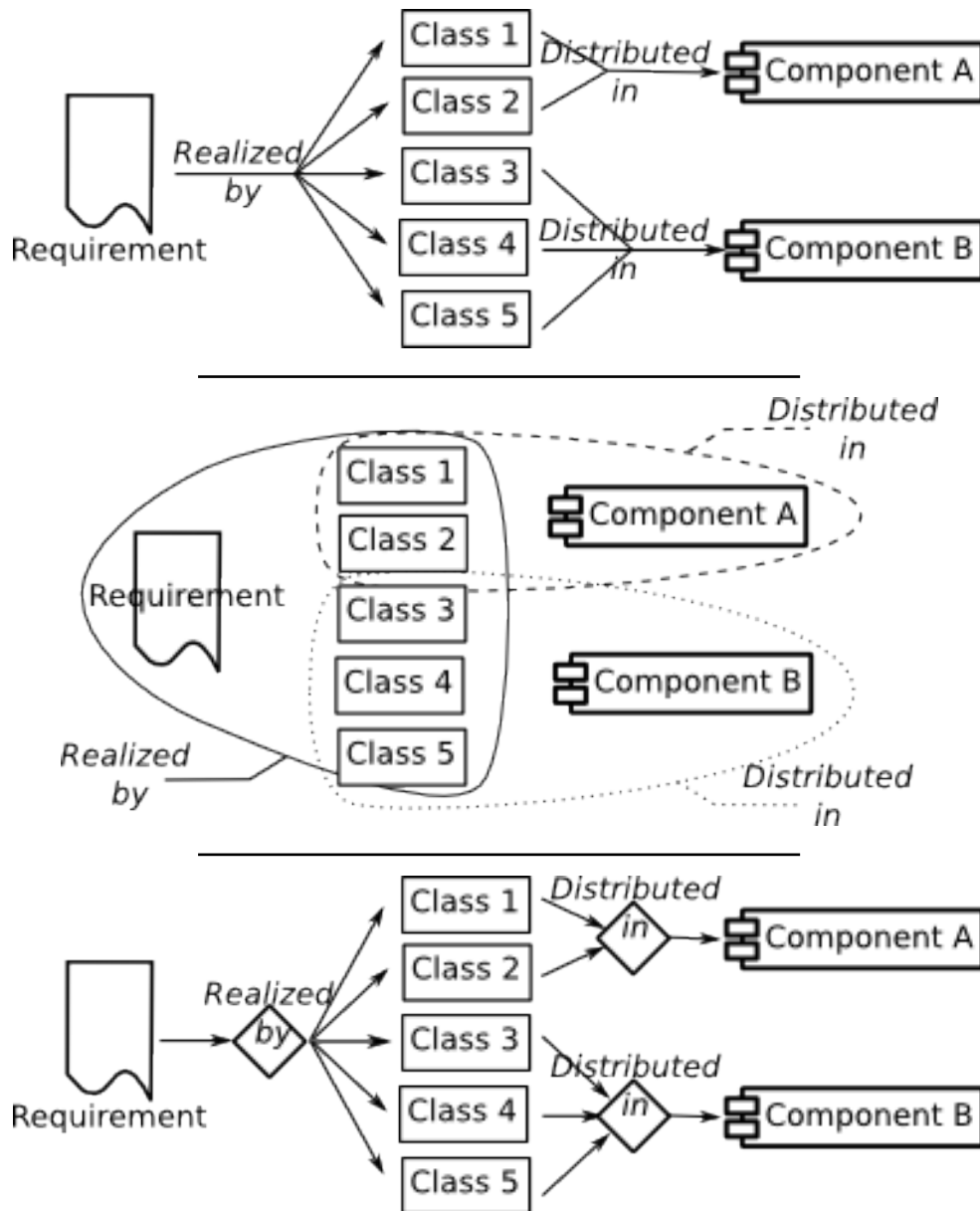


Figure 3.1: How to represent trace links? Three graphical representations of hyperlinks: links that relate more than two vertices.

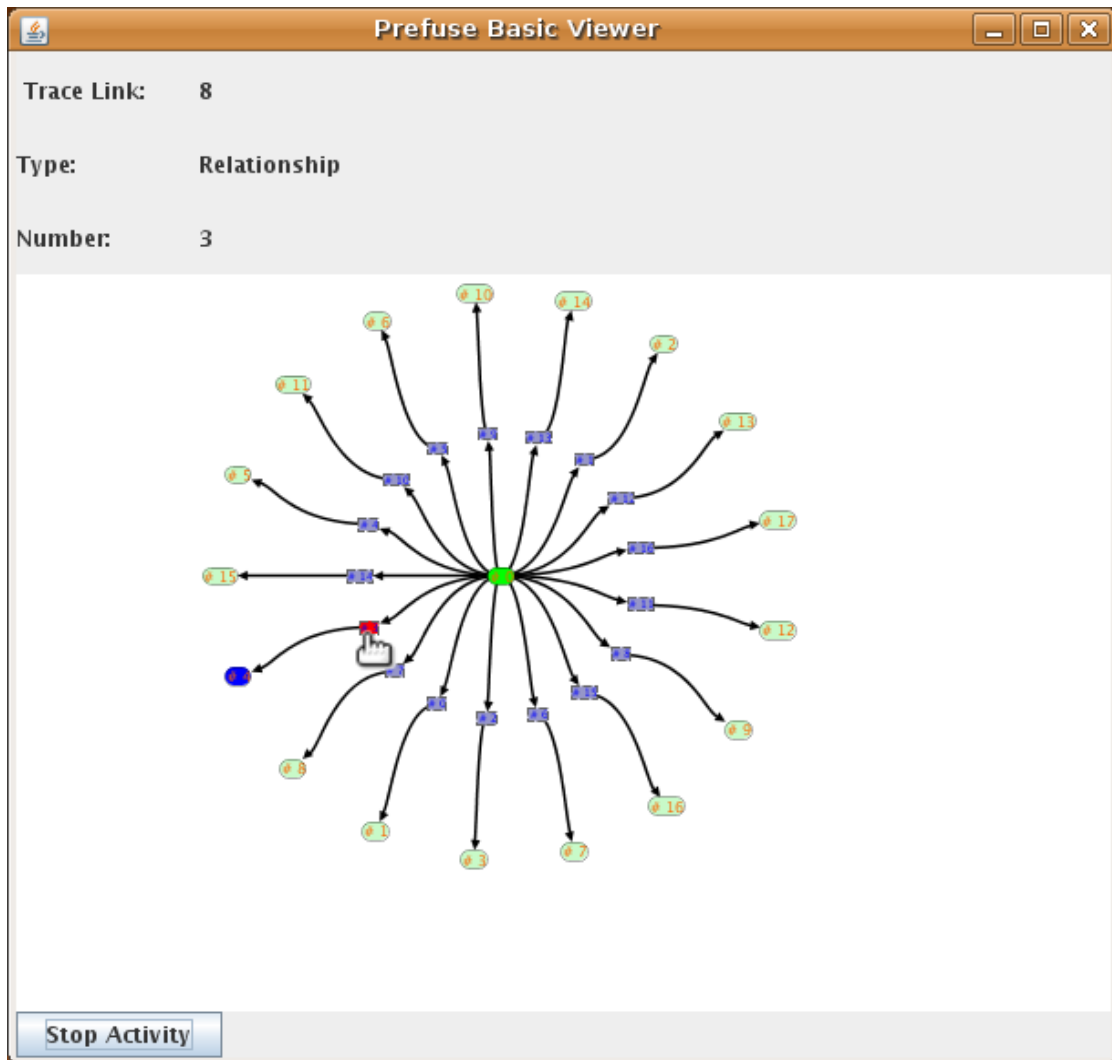


Figure 3.2: Screenshot of the basic graph view. The pointer is over a link (red), its source artefact is in green and target in blue. Top part gives information on the current node (the link).

over them. This current node is highlighted as well as its neighbours: source artefacts are green, links are red, and target artefacts are blue.

The three views are:

- **Basic:** Displays a simple bipartite graph representation (Figure 3.2) using a spring layout. The spring layout positions graph elements based on a physics simulation of interacting forces. The default is that nodes repel each other, edges act as springs, and drag forces (similar to air resistance) are applied.
- **Radial:** Also a bipartite graph representation (Figure 3.3), but the algorithm used for positioning the artefacts is “radial” and based on the spanning tree of the graph. It takes one node as the centre and places the other related nodes in semi circular layers. This visualization assumes the graph is connected (there are not several unrelated subgraphs of nodes). The figure allows to see the set of impacted nodes selecting one originating

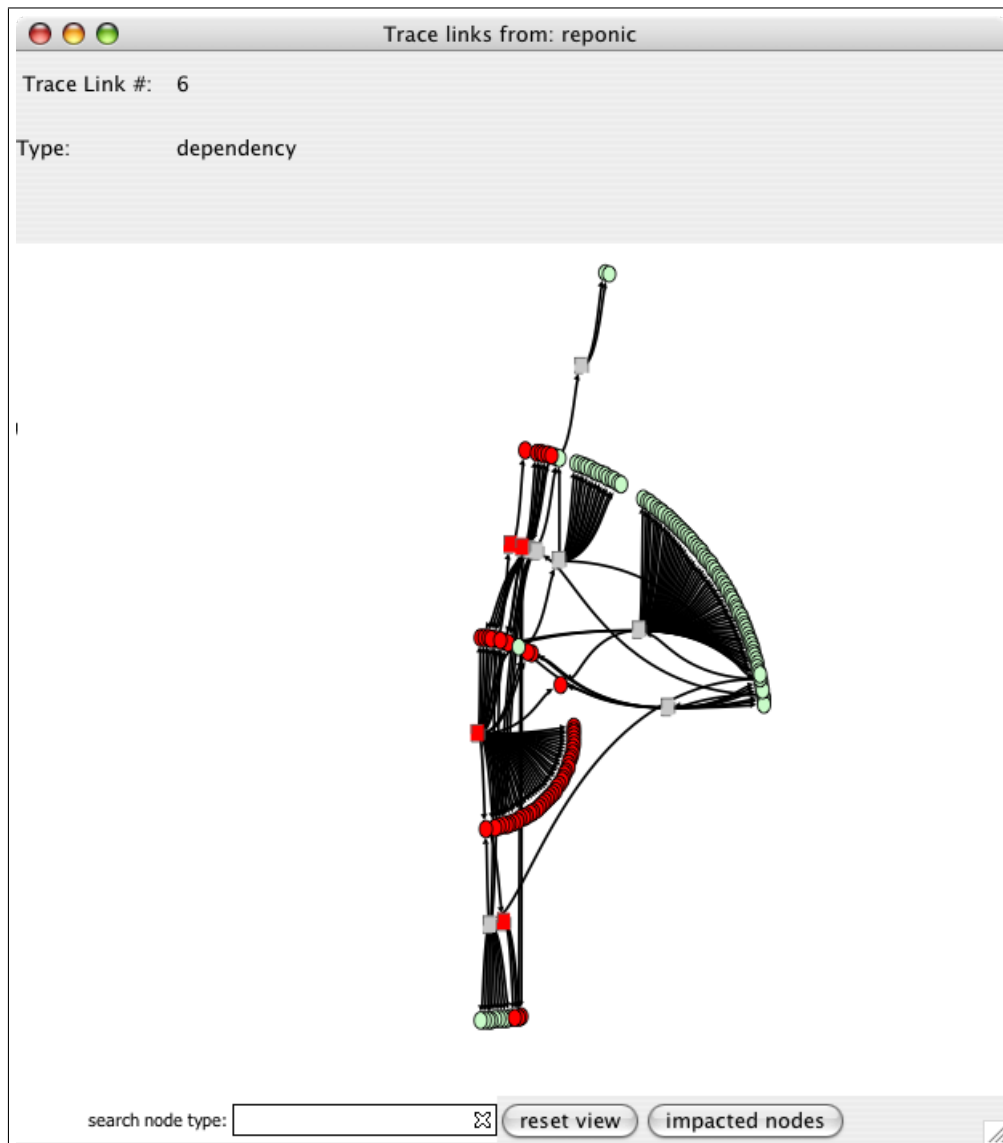


Figure 3.3: Screenshot of the radial graph view. The pointer is over an artefact (brighter green), top part gives information on the current node.

node.

- Hypergraph: Displays an hypergraph view (Figure 3.4) of the trace links. It uses the Venn Diagram principle and a spring layout which gives a kind of cluster view.

3.2 Metrics

Some basic measures have been provided for a trace graph. We have categorized them to ease understanding. Many of these metrics consider a bipartite graph, where links are vertices and are linked with edges to artefact (as represented in the meta model, see Chapter 1). For example, this means that links

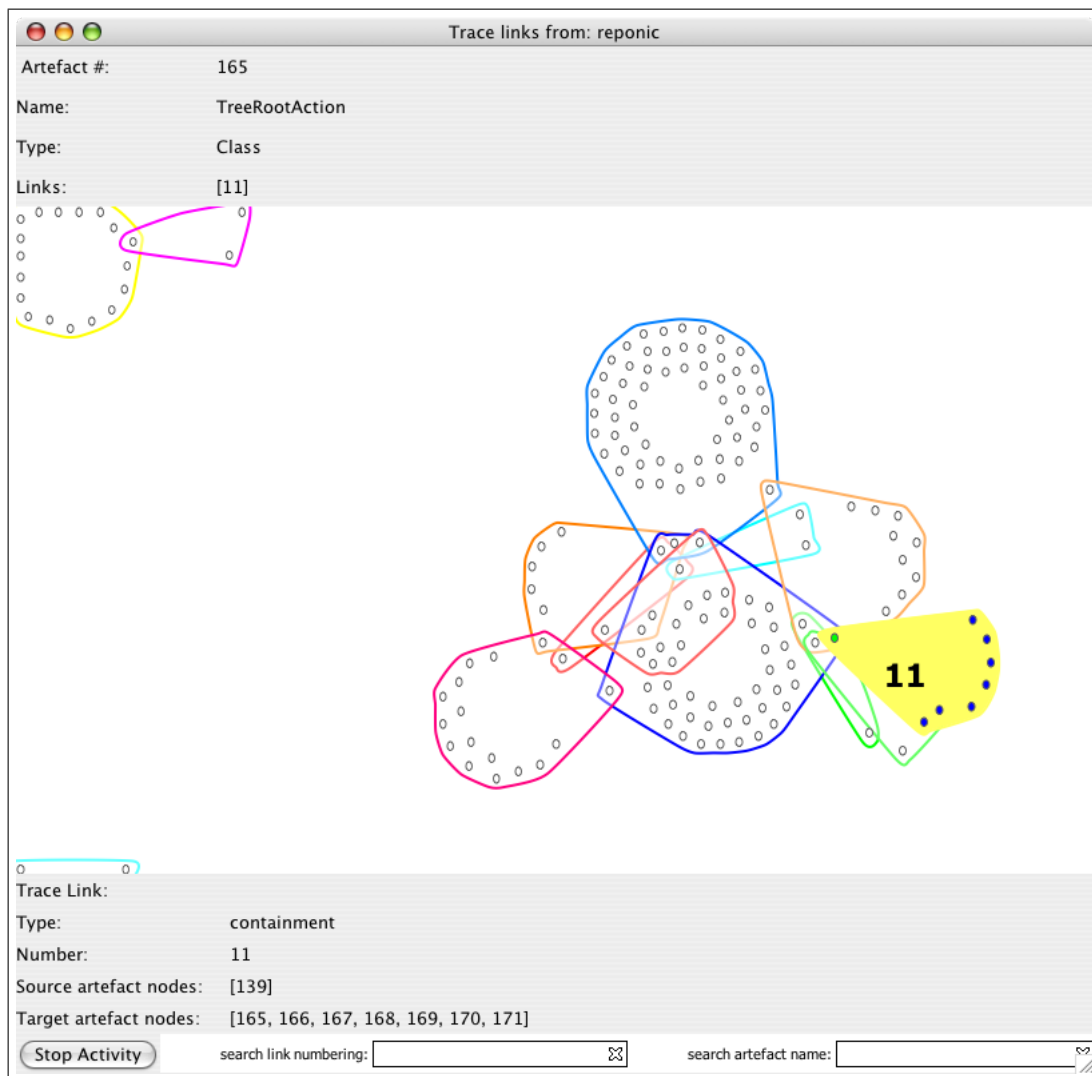


Figure 3.4: Screenshot of the hypergraph visualization. Small circles are artefact (links do not appear as node here), the coloured “sets” are links, source artefacts are green coloured and targets are blue. The pointer is over the artefact numbered 165, top part gives information on this artefact and bottom part is a description of the hyper link numbered 11.

have incoming and outgoing edges which may seem counter intuitive:

For all links: total number of link types, total number of links, total number of incoming or outgoing edges¹, average number of incoming or outgoing edges, standard deviation for incoming or outgoing edges.

For all artefacts: total number of artefact types, total number of artefacts, total number of incoming or outgoing links¹, average number of incoming or outgoing links, standard deviation for incoming or outgoing links.

For each link type: number of links, total number of incoming or outgoing edges, average number of incoming or outgoing edges, standard deviation for incoming or outgoing edges.

For each artefact type: number of artefacts, total number of incoming or outgoing links, average number of incoming or outgoing links, standard deviation for incoming or outgoing links.

For each link: number of incoming and outgoing edges.

For each artefact: number of incoming and outgoing links.

Graph metrics: total number of edges (between a link vertice and an artefact vertice), total number of nodes ($totalNumberLink + totalNumberArtefact$).

A possible use of these metrics would be to detect outliers: links or artefacts that have a number of related edges very different from the others. Such outliers may be perfectly normal (e.g. a library package could have a very large number of incoming import links) or they may also indicate some abnormal situation of interest. It would be interesting to be able to categorize artefacts or links based on these metrics.

3.3 Configuration and evolution management

This section discusses the contributions of AMPLE to configuration management for evolution of software product lines. In this section we focus on the practical issue of a tool support that is being developed as part of task 4.3. The theoretical considerations in regard to configuration management, were part of the investigations performed in task 4.2.4. and are discussed in deliverable D4.1.

¹Incoming and outgoing edges for artefacts are the same as outgoing and incoming edges for links.

3.3.1 Feature Driven Versioning

The evolution of software product lines exhibits a high level of complexity, due to the interrelated evolution of feature models, product line artifacts and final products that incorporate different features and artifacts. Our thesis is that the complexity of the evolution requires effective mechanisms for reasoning about these interrelated changes.

Feature-Driven Versioning is an approach, developed as part of the AMPLE project, that uses feature models [CHE05, KCH⁺90] to organize the artifacts of a product line with respect to the features they realize. The approach enables traceability between products and incorporated features and artifacts. Thus, Feature-Driven Versioning enables comprehensive reasoning about the relations between features, products and a software product line as a whole. A detailed discussion of Feature-Driven Versioning was published in [ME08].

Feature-Driven Versioning supports a decoupled evolution of feature models, products and artifacts, while correlating the evolution of these entities into a coherent traceability framework. For example, evolution can occur at the feature modeling level, where strategic decisions are made about the inclusion or discontinuation of product line features. The evolution of products, that incorporate a set of features and of artefacts that realize different features, is controlled separately. Thus, for example products can be maintained with older versions of features and do not need to adapt to every change in the feature model. Likewise, artefacts that realize a feature, e.g. through direct implementation in source code, can be assessed according to the changes in the feature model and the evolution of the implementation for features can arise independently of the feature evolution.

In the following section we discuss the prototype implementation of Feature-Driven Versioning by modeling an exemplary product line. Afterwards, we highlight current limitations of the prototype and discuss future work. The traceability information provided by the prototype is supplied to an ATF² repository by means of an extractor. A detailed discussion on the integration to ATF is provided in 3.5.

3.3.2 Illustrative Example

To illustrate the capabilities of the Feature-Driven Versioning prototype, we model an exemplary software product line, used to build vocabulary trainers, i.e. software tools that help in learning languages by offering training units for foreign language vocabularies.

The vocabulary trainer has a graphical front-end, which is based on GTK, a framework for graphical user interfaces (GUI) that is portable to both the Windows and the Linux operating systems. An additional GUI is designed for Mobile Devices and based on Java 2 for Mobile Edition (J2ME). The trainer is shipped with a basic set of vocabulary for beginners (basic vocabulary) and a larger

²AMPLE Traceability Framework

set for advanced learners (advanced vocabulary).

Feature Model Creation

To create a feature model, you have to create a new empty Eclipse-Project (named 'Model' in our example). This project will contain a product and a feature model at the end of this section. Create a new FeatureModel by clicking *File* → *New* → *Other...* and select *SimpleFeatureModel* in the category *EMF – Model Versioning*.

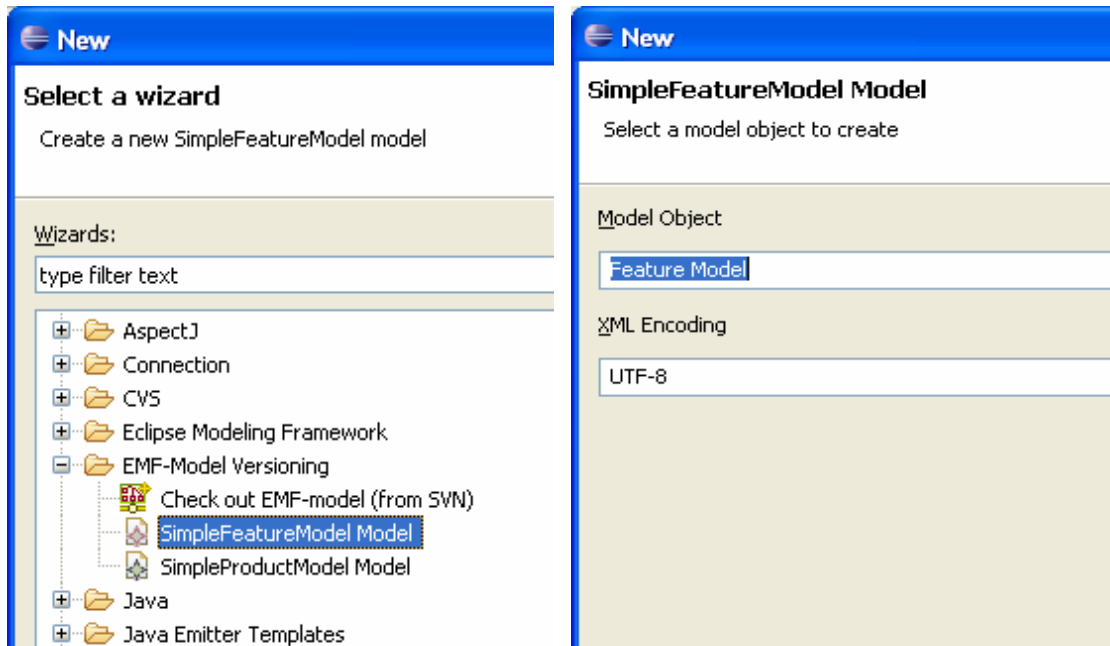


Figure 3.5: Feature Model Wizard Selection

A wizard will guide you through creating a SimpleFeatureModel file. In the example, the file will be named *VocabularyTrainer.simplefeaturemodel*. Now select the Eclipse-Project 'Model' as parent for the model and choose Feature Model to be the Model Object as depicted in Figure 3.5. Finish the wizard by clicking Finish and the file *VocabularyTrainer.simplefeaturemodel* will be created in the Project 'Model' and opened in a new editor as depicted in Figure 3.6.

Feature Modeling

The feature model is a domain oriented model based on hierarchical structure, which explicitly models the variability in the product line. In our graphical representation, features are represented as nodes in a tree. Our representation of a feature model contains an explicit 'feature model' entity that references the topmost (root) feature of the model. The root feature is the most general feature and typically contains a description of the software product line as a whole. Under the root feature, there can be any number of 'sub features', which may have any number of sub features themselves.

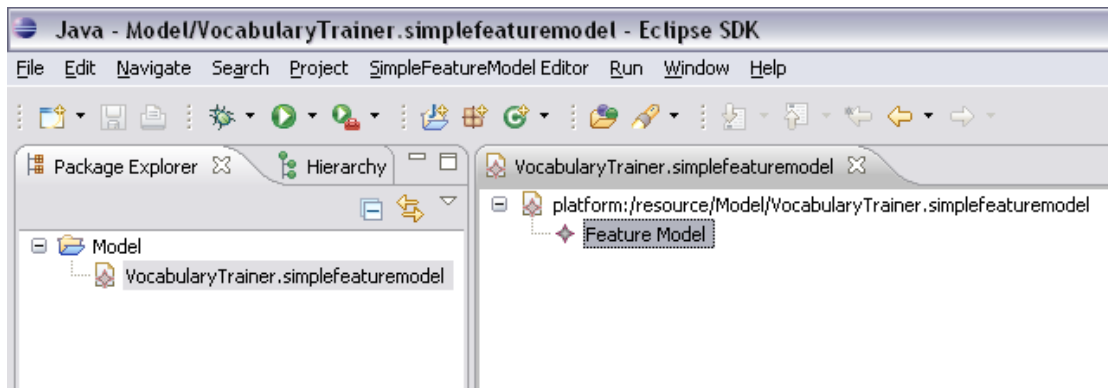


Figure 3.6: Empty Feature Model

To add new features, right click an entity in the tree and select *New Child* → ... in the sub menu. All entities will be listed, which are allowed to be created as children of the current selected entity. For example in Figure 3.7 we selected a feature model entity, so the only allowed child is a root feature.

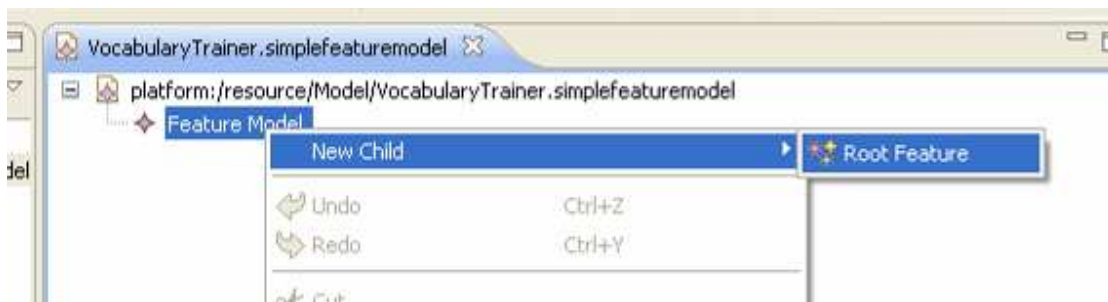


Figure 3.7: Starting a new Product Line

Now we create a feature model for the Vocabulary Trainer software product line. The Vocabulary Trainer has two main features. The first is the *GUI* feature, which is the parent feature for variable user interfaces. The second is the *Content* feature, which is used to model variability regarding the content of vocabularies contained in the software. The GUI feature has two sub features GTK and Mobile (J2ME). These features model variable user interfaces based on different user interface frameworks. The different content features has two children that express variability in terms of a basic and an advanced vocabulary. The feature model is depicted in Figure 3.8.

Each feature has a set of properties, which describe the feature in more detail. Figure 3.9 depicts the properties of the Vocabulary Trainer root feature.

- *Lower Bound and Upper Bound*, are constraints for the selection of the sub features into products. In our model, this means: "If a product incorporates the root feature Vocabulary Trainer, it also has to select a minimum and a maximum of two sub features of this root feature". I.e. all products have to select exactly two sub features of Root Feature Vocabulary Trainer. These constraints are important for the validation of the correctness of products, which are defined as selections of the features.

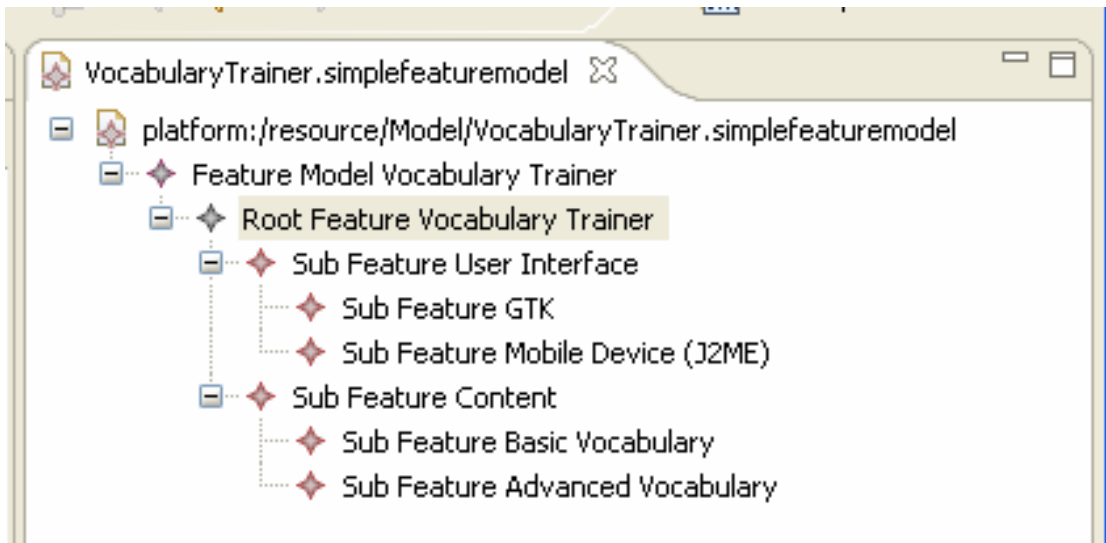


Figure 3.8: Vocabulary Trainer Feature Model

Property	Value
Feature	
Lower Bound	2
Upper Bound	2
Misc	
Description	
Id	2545c80a-53cc-4859-8283-f5358fa4b0d0
Name	Vocabulary Trainer
Version	1

Figure 3.9: Properties of the Vocabulary Trainer root feature

- *Id*, is an automatically created, unique identifier for each entity of the model.
- *Name*, is the name of the feature (as shown in the tree). A name labels a feature to convey a certain meaning but does not uniquely identify features. Thus the name of a feature may change during the evolution of the feature model.
- *Version* contains version information of the feature. The version is incremented, when the feature changes. Versions are not changed automatically during editing. To enable versioning, the model is 'checked-in' into a Subversion repository.

Feature Model Versioning

In the Feature-Driven Versioning approach, each feature is assigned its own unique version. Thus, it is possible to reason about changes in the feature model locally. Traditional configuration management systems such as Subversion are agnostic to the structure of their content and provide only versioning for whole files, which correspond to the whole feature model in our case. In order to provide comprehensive traceability for the evolution of product lines, we have adapted Subversion to provide fine-grained versioning for software product line modeling.

In Feature-Driven Versioning changes of a feature also entail changes of the parent features, up to the Feature Model. This means, if you change something in the feature GTK, GUI and Vocabulary Trainer will also change logically, because GTK is a part of feature GUI and Feature Model. This propagation of change is a necessity for local reasoning. Without propagation the structural changes can not be uniquely determined as there would exist a dependency between one version of the parent feature and two different versions of the child features. Using the version propagation mechanism a unique identification of each change to the feature model is possible.

To enable feature model versioning in the prototype, we have to share the project containing the feature model via selecting *Team* → *Share* → ... in the context menu of the model project. One chooses Product and Feature Model Provider (SVN) as depicted in Figure 3.10 and click on next. A wizard will guide you through the process of sharing, asking for the location of the subversion repository to store the model, and the default path for containers (default repository to use for artefacts that implement/realize features or products).

After clicking finish, a little question mark will indicate that the project is shared. In the context menu of the feature model, it is now possible to choose *Team* → *Check in model to SVN*.

After sharing the feature model on Subversion, all changes to the feature model can be assessed in terms of new versions. Assuming, we want to add a new feature, called 'Speech Recognition'. I.e., the Vocabulary Trainer should not only check written answers, but also detect mispronunciation of vocabularies. After adding the new feature to the mode, we check the model in again.

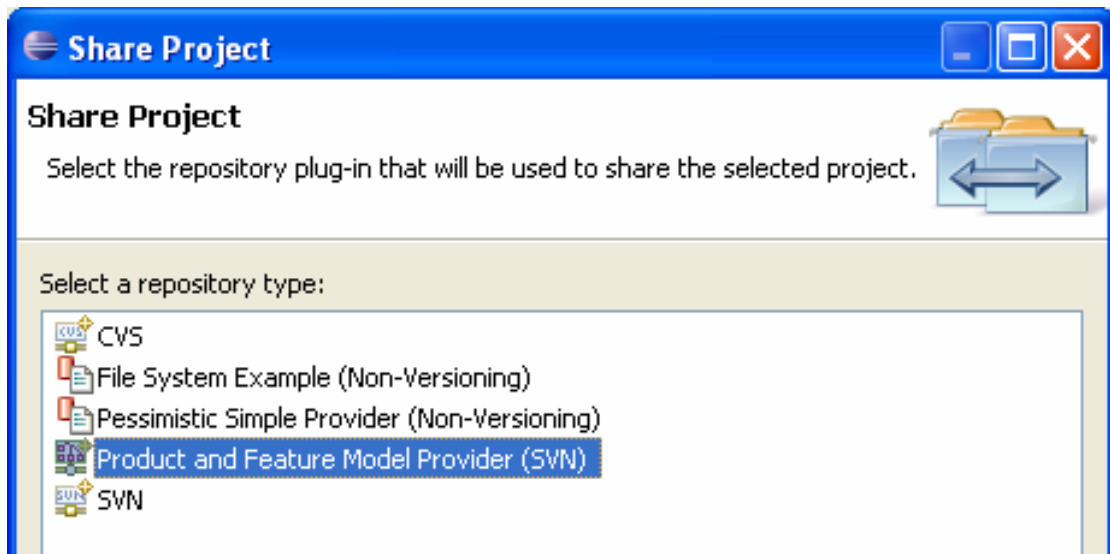


Figure 3.10: Feature Model Versioning Wizard

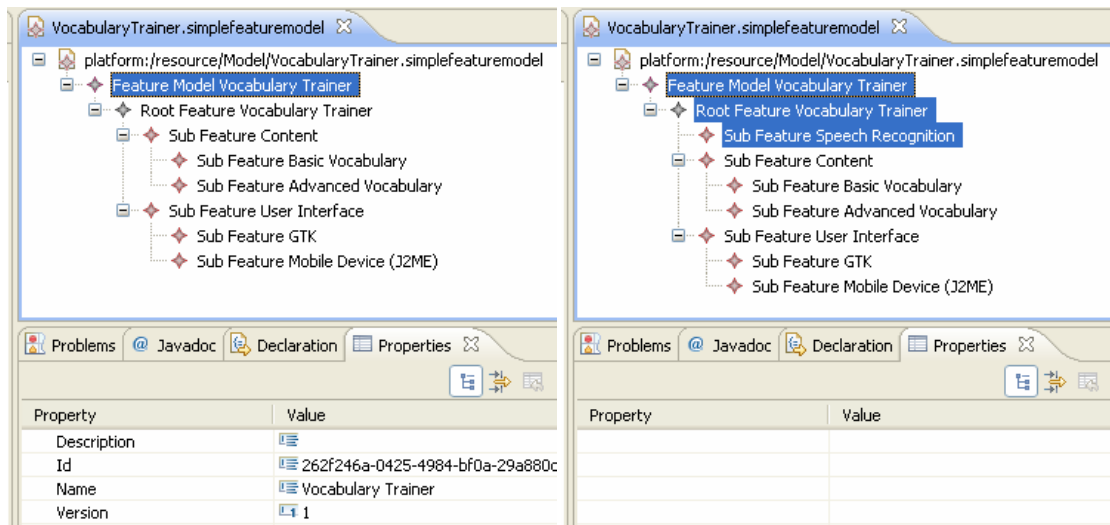


Figure 3.11: Feature Model Evolution

Figure 3.11 depicts the initial feature model on the left and the changed model on the right for comparison. The highlighted features of the changed model have a version property of 2. This version indicates, that conceptual changes took place either in the feature itself or in a child feature. The version property of all other features remains unchanged, indicating that neither the features nor its children changed.

Feature Implementation

In Feature-Driven Versioning a feature can be traced to the artefacts that are responsible for realizing the feature in the product line. This traceability to artefacts also covers the evolution of the product line, where firstly artefacts themselves may evolve, and secondly the evolution of the features entails changes to the realizing artefacts. To enable this traceability, each version of a feature has a repository container for all the artefacts connected to this particular version of the feature. The necessary information regarding the repository container is also stored in the properties of a feature.

The repository container of the feature also exists in different versions in order to enable a decoupled evolution of the feature model and the connected artefacts. For example, the addition of the 'Speech Recognition' feature results in a new version of the 'Voabulary Trainer' feature. However, this does not mean that we necessarily add new artefacts to realize the new version of the 'Voabulary Trainer' feature. In addition not all features are necessarily associated with artefacts. The property *Implementation allowed* indicates, if the feature has a container (*true*) or not (*false*). In our example we don't need artefacts to realize the feature 'Content' and only require this feature in the model for an organizational purpose.

To demonstrate the repository containers and their evolution, we add some implementation code to our feature model. We start with the implementation of the feature GTK. To create a repository container click *Implementation* → *Add implementation for GTK...* in the context menu of the feature GTK as depicted in Figure 3.12.

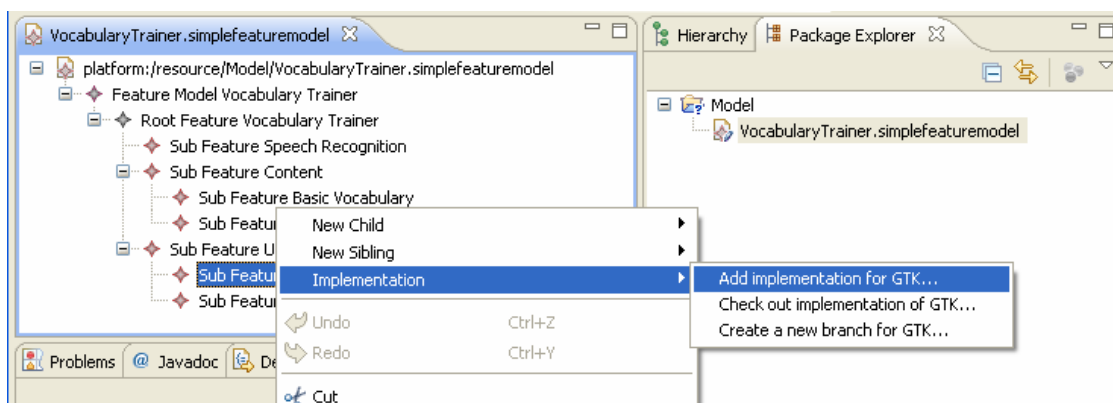


Figure 3.12: Providing Feature Implementations

As a result, an eclipse project is created for each feature, up to the root fea-

ture. The projects have the same name as the feature they belong to. Artefacts that are connected to specific features can then be placed inside these projects. In addition, the projects have references according to the feature dependencies defined in the feature model. This is done by utilizing the eclipse project dependency mechanisms as depicted in Figure 3.13.

This current form of associating features with artefacts is a simplification made for the prototype. More complex mechanism for associating features and artefacts can be explored in the third year of the AMPLE project. Likewise, the current feature model only captures dependencies from features to their parent features. More elaborate feature interactions mechanisms are also future work.

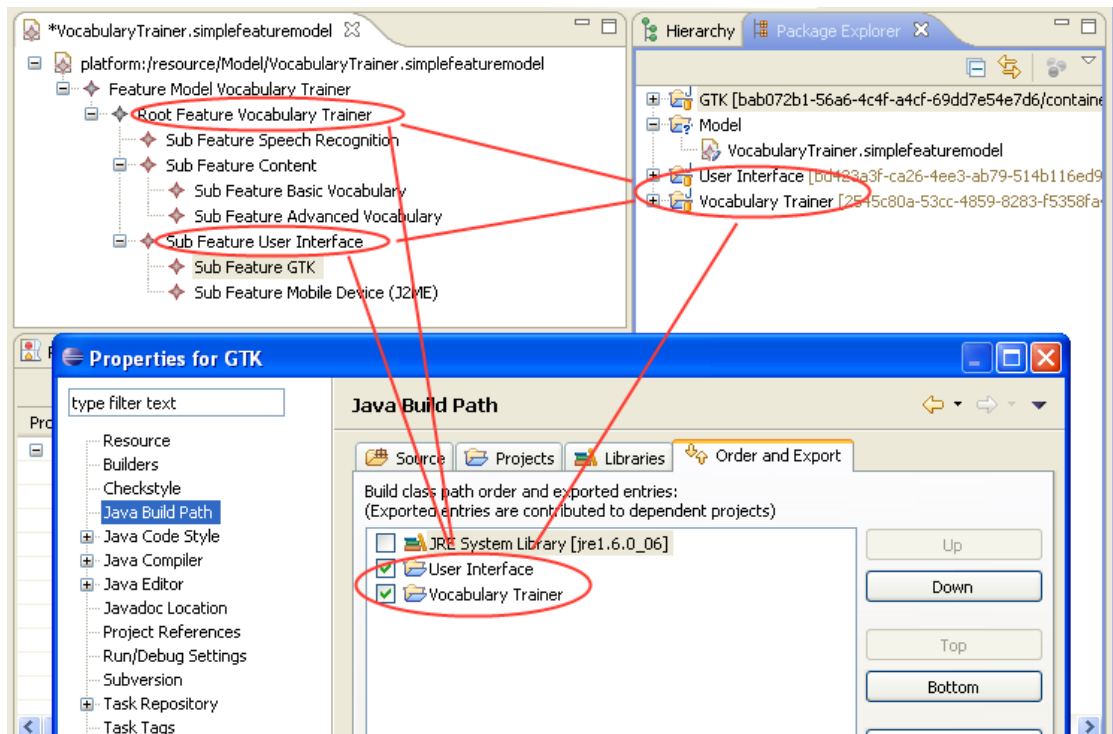


Figure 3.13: Feature Dependency Mapping

Now, we add some java code into the projects. In the Vocabulary Trainer we will implement an abstract class `VocabularyTrainerMain.java`, on which all products should be based. The User Interface will be implemented as an interface `IUserInterface.java`, which has some methods. GTK will contain `GTKComponent`, which will implement `IUserInteraction`. After creation of these three files, we check them in into SVN. The evolution of the projects will be done with usual Subversion Checkin/Checkout cycles. E.g. Adding new classes to some features and check in to SVN.

Product Creation

Now we create products from our exemplary Vocabulary Trainer product line. Products are defined using a product model. The editor for the product model has the same look and feel as the feature model editor. To create a product

model, select *File* → *New* → *Other...* and select *SimpleProductModel Model* in the category *EMF – Model Versioning*. A similar wizard to the creation wizard of the feature model will appear, allowing the definition of a product model file. In our example, we create the file *VocabularyTrainer.simpleproductmodel* and place it in the same Eclipse project as the feature model, i.e. project 'Model'. In the wizard the 'model object' selected for the products must be 'Product Model'.

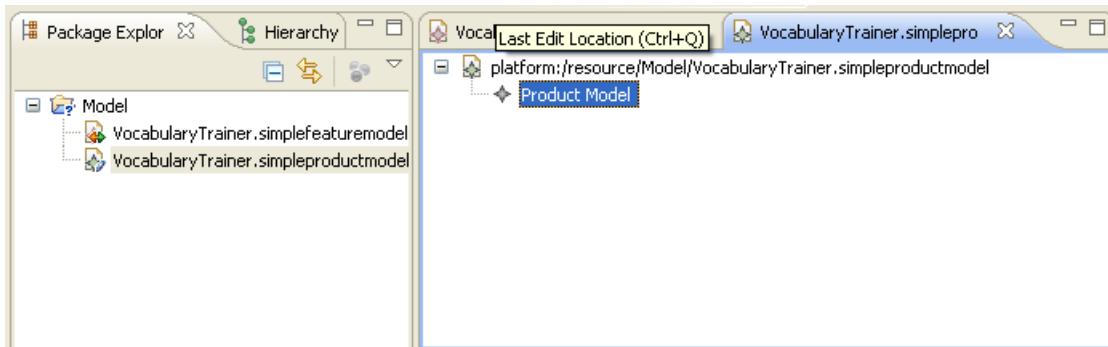


Figure 3.14: Empty Product Model

The product model consists of one root node 'Product Model', which contains all product definitions. A product is a selection of one or more features from the feature model. Before we can select features, we have to select the feature model that is used for product definition. This can be done via the context menu *LoadResource...* and *Browse Workspace...* and then choosing the file *VocabularyTrainer.simplefeaturemodel*. Similar to features, new products can be created via context menu *New Child* → *Product...*. For our example we create three products, *Free Edition* and *ProfessionalDesktopEdition* and *ProfessionalMobileEdition* as depicted in Figure 3.15. Each product incorporates a different set of features from the software product line. For example, the *Free Edition* includes the *GTK* and the *Basic Vocabulary* features. Parent features are implicitly selected.

Each product has again a set of properties. *Description*, *Id*, *Name*, *Version*, *Container Repository Link*, *Container Version* and *Implementation Allowed* have the same semantics as the same attributes for features. In addition each product has properties for selected *Features* and *Referenced Version*. The selected features are the features that are incorporated into building the product. The referenced version describes which versions of features are selected into the product.

If one clicks on the field to set the property features, a dialog will appear that allows to add or remove features to the product. The selected features will show up as nodes below the product node. Note, that features should not be deleted from the product model for deselecting them, because this would delete these features from the feature model. To remove features from a product, one uses the dialog described above. After selecting some features, the property *Referenced Version* is updated automatically to the version of the feature model the selected feature belongs to. This reference version uniquely identifies all

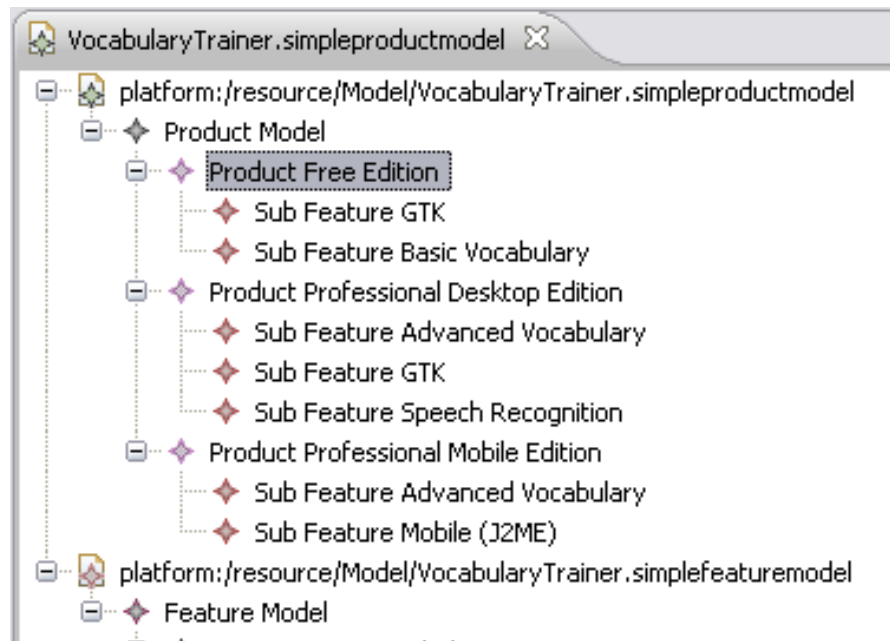


Figure 3.15: Vocabulary Trainer Product Model

versions of all features that reside in the feature model.

3.3.3 Future Work

The prototype for Feature-Driven Versioning provides traceability for features, products and artefacts. Currently the prototype supports only coarse-grained traceability for artefacts, i.e. the features are traced to files as the abstraction level for artefacts. More fine-grained information, e.g. traceability to elements that are contained in a model such as an UML Model, is currently not provided. This traceability information can prove valuable to the analysis of the evolution of a software product line. However, such information requires domain-knowledge about the model in question in order to understand the structure of the model and infer traceability links to specific model elements. Such knowledge can not be obtained using current software configuration management systems, as these systems are agnostic to their content and also only operate on the levels of files. However, building on the Feature-Driven Versioning Prototype as a solid framework for information regarding the evolution of a software product line, such fine-grained traceability information can be inferred by specialized traceability tools and then correlated to the traceability information in evolution. The integration of the information can be performed in the ATF Repository developed in the AMPLE project.

```

<?xml version="1.0" encoding="UTF-8"?>
<ss:Workbook xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet"
             xmlns:p="urn:TraceViewPlugin">
  <ss:Styles>
    <ss:Style ss:ID="1"><ss:Font ss:Bold="1"/></ss:Style>
  </ss:Styles>
  <ss:Worksheet ss:Name="SGN">
    <ss:Table>
      <ss:Column ss:Width="220"/><ss:Column ss:Width="100"/>
      <ss:Row ss:StyleID="1">
        <ss:Cell><ss:Data ss:Type="String">Simple global numbers</ss:Data></ss:Cell>
      </ss:Row>
      <ss:Row/>
      <ss:Row ss:StyleID="1">
        <ss:Cell><ss:Data ss:Type="String">Name</ss:Data></ss:Cell>
        <ss:Cell><ss:Data ss:Type="String">Value</ss:Data></ss:Cell>
      </ss:Row>
      <ss:Row>
        <ss:Cell><ss:Data ss:Type="String">Total number of nodes</ss:Data></ss:Cell>
        <ss:Cell><ss:Data ss:Type="String">65</ss:Data></ss:Cell>
      </ss:Row>
    </ss:Table>
  </ss:Worksheet>

```

Figure 3.16: Example of the output of native Excel export.

3.4 Exports

To allow interaction between the traceability framework and external tools, we created some export facilities:

XML: XML is a standard that allows to transfer data between a wide range of applications. It is a natural candidate for exporting about any data. We defined a XML exporter to export three kind of data: trace information from the repository, the result of all the metrics in 3.2, and traceability matrix (a square matrix where all links between artefacts are indicated).

DOT file (graphviz): The Graphviz tool, [EGK⁺02], can be used to visualize state machines or architectures. It also proposes several layout algorithms for graph and is able to compute connected parts, merge several graphs, compute size of paths, compute transitive reduction, and so on.

Excel (native): Using XLS transformation rules, we can convert the XML export to native MS-Excel format and import it directly into Excel. As an example, a short snippet of the metrics file containing the global numbers table in Excel-XML can be found in Figure 3.16.

3.5 Extractors

As well as export facilities, the trace repository requires import facilities to be able to interact with external tools. We defined three extractors which are briefly described below.

Java: This plugin imports data from Java projects. Recognized artefacts are packages, classes and methods; recognized traceability links are: import (between packages) and defines.

More artefacts and links could be recognized relatively easily since we created a complete Java parser.

XML: As stated in the previous section, XML is a natural candidate for exchanging about any data. We created a simple XML extractors that can read artefact types, link types, artefacts (with properties) and links from an XML file.

Configuration Management Extractor: This extractor synchronizes data provided by the configuration management and evolution approach discussed in Section 3.3. The approach relies on Subversion as a back-end for storing versioning information. Thus the Subversion system already contains data regarding the evolution of the product line. However, Subversion does not provide easily usable query mechanism. Therefore we provide an extractor that transforms the information into an explicit representation in an ATF repository. Figure 3.17 depicts a metamodel of the information that is extracted from the configuration management system. Currently the extractor imports traceability information between fea-

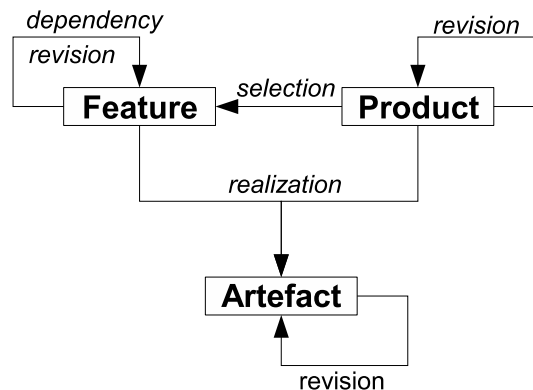


Figure 3.17: Overview of traceability information provided by configuration management

tures and artifacts, but does not incorporate the additional information regarding products, i.e., the selection of features into products and the artefacts attributed to products. Also the extractor currently only reads the latest version of the feature model from the Subversion repository. Therefore, to get the complete traceability graph the extractor must be run after each check-in of the model into the repository. The extractor requires the project that contains the feature/product model as an

entry-point to know in which repository these models are shared. The project is looked up by its name, with a default value of 'Model' for the project. Other project names may be provided via Eclipse preferences under *Team* → *Product and Feature Model Provider*.

For now, the XML formats for import and export have not been unified because they were developed separately with different purposes in mind. It is clearly a short term objective to unify these two format so that the traceability framework can import the data it exports.

3.6 Future Work

The plugins are available as prototype for testing. Other partners in the project (presumably from other work package) should test them to see if they suit their needs. Our development plans for these plugins are focused on creating a global understanding of how information can flow between the various extractors and be integrated in the repository. For example, how could the java extractor and the subversion extractor collaborate to provide traceability of java code over various versions of this code.

New extractors should be needed by partners from other work package to deal with the entire product line development process. Again, it is not yet clear how all these extractors could collaborate.

Work on advanced queries and visualization could be done too.

Finally, there is still a question on how well this prototype would scale up and if it could support the amount of traceability links to be expected on a real product line.

4 Planned Extension

In this chapter, we describe two future extensions to the traceability framework.

4.1 Traceability of Design Decision Rationale

In this section we describe the envisioned tool support for the traceability of design decision rationale in the presence of uncertainty.

4.1.1 Introduction

The AMPLE project envisions the explicit handling of uncertainty during SPL development. We are focusing on handling uncertain information on the rationale used to resolve design decisions. For this purpose, we have defined in [GKN⁺07] a meta-model that conceptualizes the kind of Design Decision Rationale (DDR) in which we are interested. This meta-model comprises elements from argument-based rationale methods, problem-solving approaches and quality evaluation methods. The meta-model accommodates the treatment of uncertainty in the assumptions made by the developers while taking design decisions. Uncertainty is represented by utilizing techniques from fuzzy set theory.

For performing traceability in the presence of uncertainty, we aim at tracing the design decision rationale to other development artefacts [AGG⁺08]. The traces related to design decision rationale instances must be stored along with inter or intra traceability relationships. For example, the design decision rationale should be traced to other decisions, or from and to other artefacts, such as requirements and architectural models. The traceability of design decisions in SPL is necessary for improving the understanding of the important contextual factors that impact the quality of the SPL and variability management. It also should facilitate the analysis of the influences of the uncertainty on the development of the SPL. We intend to utilize the tool support for the sake of performing rationale management, change impact analysis and root-cause analysis.

The next subsections describe the main tool modules to be developed for the purpose of managing and tracing design decision rationale. These modules will have interfaces with each other, on which functionalities related to each of them can be called by the other.

4.1.2 Tool Support for Rationale Management

The tool support for rationale management regards a language editors for the design decision rationale metamodel [GKN⁺07], a management tool and a set of analyzers.

- **Design Rationale Modelling Language.** The implementation of a domain specific language for specifying design decision rationale and the tool support for editing and visualizing the design models is ongoing. With this language we will be able to specify the models of design decision rationale and the uncertainty capture during SPL engineering.
- **Rationale Management.** This part of the tool concerns the functionalities that model and edit the design decision rationale and its elements. The main functionalities which belong to this module are: triggering of trace capture(from specified traceability rules); evolution of design models defined on the design rationale modeling language; design decision rationale interpretation; and monitoring of the invalidation of assumptions made for a design decision.
- **Uncertainty Analyzer.** The uncertainty analyzer comprises the algorithms for reasoning on uncertainty and the decision optimizers.
- **Traceability Analyzer.** The traceability analyzer is the module that interfaces with the plug-in for traceability and relates the detailed specification and visualization of design decision rationales with their traceability.

4.1.3 Tool Support for Traceability of DDRs

The second tool module envisions the support for traceability of design decision rationales. This module will be developed as an Eclipse plug-in which makes use of the extension points of the graphical front-end for ATF, presented in Chapter 2. We have, therefore, identified the following extensions:

- **Trace Extractor.** We need to extend the ATF to be able to extract the DDRs generated in the Rationale Management tool.
- **Trace Register.** The DDR must be registered as a traceable artefact and we must create registers to all kinds of trace dependencies from DDRs to other traceable development artefacts. In addition to the DDRs, the other traceable artefacts to be registered are: concerns, architectural elements, variation artefacts and implementation elements. Examples of trace dependencies are the ones from design decisions to the previously mentioned traceable artefacts.
- **Trace Query.** Extensions are necessary for processing queries over the traces in the repository which involve the design decision rationale and its elements. Examples of such specific queries are:

- Trace development artefacts by design decision;
 - Trace design decisions by development artefact;
 - Trace design decisions by design decision (query over chains of design decisions);
 - Trace the evolution of a design decision;
 - Change impact analysis due to changing design decisions;
 - Change impact analysis on design decisions due to changing development artefacts;
 - Root-cause analysis for changing development artefacts;
- **Trace View.** We may extend the trace views to be able to perform the following:
 - Visualization of evolution metrics of design decisions by means of tables and graphs;
 - Visualize the SPL development process as a chain of decisions;

In addition to these extensions, we envision calculating metrics of design decisions over other decisions and development artefacts. Examples of such metrics are the diffusion and concentration of design decisions and the measure of interlacing of design decisions.

4.2 GAMBLE: A Generalized Framework for Traceability Link Rejuvenation

4.2.1 Introduction

Recording traceability information during the software engineering life cycle is a valuable activity. This information can prove helpful in comprehending large, complex systems, as well as verifying that each planned requirement of the system was, in fact, implemented and furthermore adequately tested. However, non-trivial systems are typically prone to some form of *evolution*. With requirements tending to change over time, emergence of agile methods [HC01], and possible incorporation of new technologies [KSR07, vDD04, DKTE04, FTK⁺05, KETF07, TFDK04], software may undergo many alterations such as refactoring [Fow99, Ker04], architectural restructuring, design modifications, etc. Unfortunately, many of these kinds of changes may invalidate traceability information that has been recorded about the software. Even seemingly innocuous changes to any element of the traced artifacts are highly likely to have negative, cascading ramifications to the corresponding trace information. As such, the trace information is considered *fragile* in an evolving artifact base, and effort must be made to *maintain* the information with each artifact alteration.

In this section, we present a preliminary design for an approach that is intended to mechanically alleviate the burden associated with maintaining trace information over the lifetime of a software product. The proposed approach plans to offer a general framework for traceability link rejuvenation (GAMBLE) which provides automated mechanisms to help with the task of maintaining trace information as the traced artifacts evolve.

Proposed Methodology

The GAMBLE approach proposes to mitigate the task of traceability maintenance by *inferring* the true intentions of the developer in mapping particular artifact transitively to a concern the software is to ultimately realize. We propose that it does so by adapting the use of a generalized *concern graph* [RM02] which models relationships amongst traceable artifacts, as well as the developer's intentions in mapping (or transitively mapping) particular elements to a concern. Concern graphs have been commonly used in describing and tracking concerns within source code and are planned to be adopted in GAMBLE to uncover the essence of traceable artifacts mapped to a particular concern. Vertices of the graph represent traceable artifacts, while directed arcs represent relationships between those artifacts. Trace information will then be compared with graph elements; elements realizing a particular concern are denoted as *enabled* w.r.t. to that concern. Next, *intention patterns* expressing general *shapes* of acyclic, finite paths that include enabled elements are to be derived. The patterns are then themselves analyzed in order to associate a *confidence*, a real number in the interval of $[0, 1]$, in the pattern uncovering future traceability links in an evolved version of the software.

4.2.2 General Proposal Overview

The approach presented here is separated into two distinct phases: *analysis* and *rejuvenation*. The analysis phase (phase I) is responsible for deriving intentional patterns, as well as associating an accurate confidence to them, in the original version of the artifact base (e.g., requirements, architecture, design, implementation). The rejuvenation phase (phase II), on the other hand, is responsible for inferring new links in a revised (or evolved) version of the artifact base.

Phase I: Correlation Analysis

Figure 4.1 depicts a flowchart that corresponds to the planned analysis phase of GAMBLE. The process begins each time trace information is either created or modified. The existing artifact base is then analyzed in order to build the concern graph, i.e., artifacts and their relations. Next, GAMBLE is to then associate the trace information with vertices and arcs in graph, thereby enabling these elements with respect to the given trace information. Then, intentional patterns

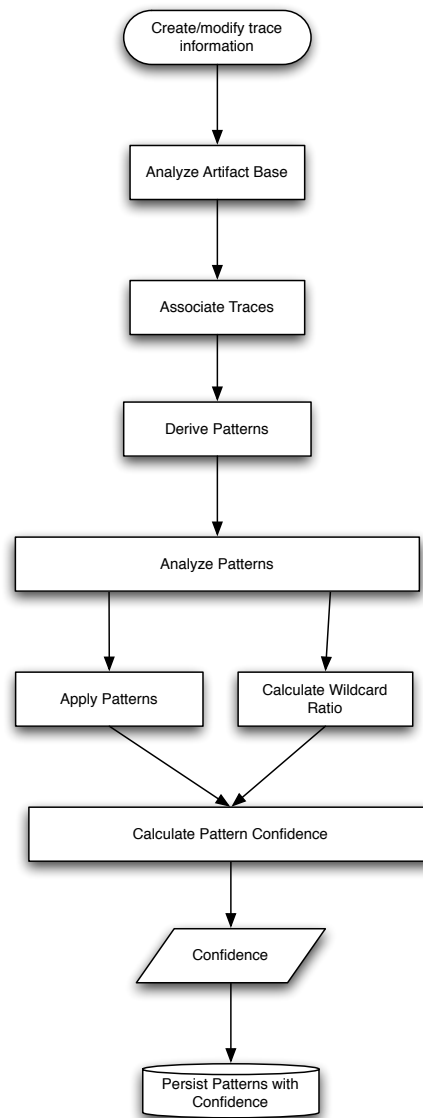


Figure 4.1: GAMBLE Phase I: Correlation analysis flowchart.

are to be produced from finite, acyclic paths in the graph. During the pattern derivation process, various wildcard elements are to be placed into the patterns so that they would match in future artifact base versions. The ratio of wildcard elements to concrete elements, i.e., its *abstractness*, is then combined with how well the patterns match the *current* version of the artifact base. We plan to utilize the principle of locality here and, thus, hypothesize that future versions of artifacts mapped to a concern will exhibit similar properties as the current version. These two metrics combine to form a pattern confidence. The patterns, along with their confidence, are lastly persisted for use in later versions of the artifacts base where trace information will require rejuvenation.

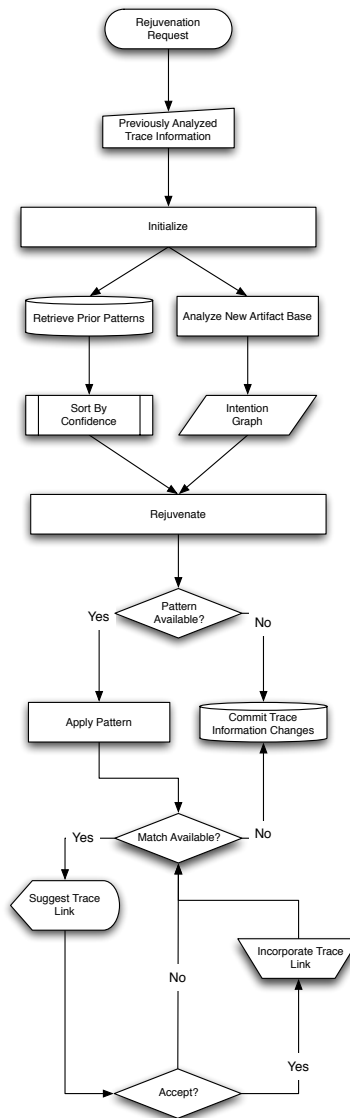


Figure 4.2: GAMBLE Phase II: Link rejuvenation.

Phase II: Trace Link Rejuvenation

Figure 4.2 portrays a flowchart that corresponds to the planned rejuvenation phase of GAMBLE. The process is to begin by the user requesting that trace information be rejuvenated after any form of evolution of the artifact base. At this point, the previously analyzed information from phase I will be regurgitated in order to apply the formerly derived patterns to the new artifact base version. Initializing this process would also involve building a concern graph for the new version. Suggested trace links are to be produced by first sorting the regurgitated patterns by their corresponding confidence decreasingly and then applied to the *newly* created concern graph. GAMBLE is then to proceed to make suggestions to the user until the trace information has updated to reflect the new changes in the artifact base.

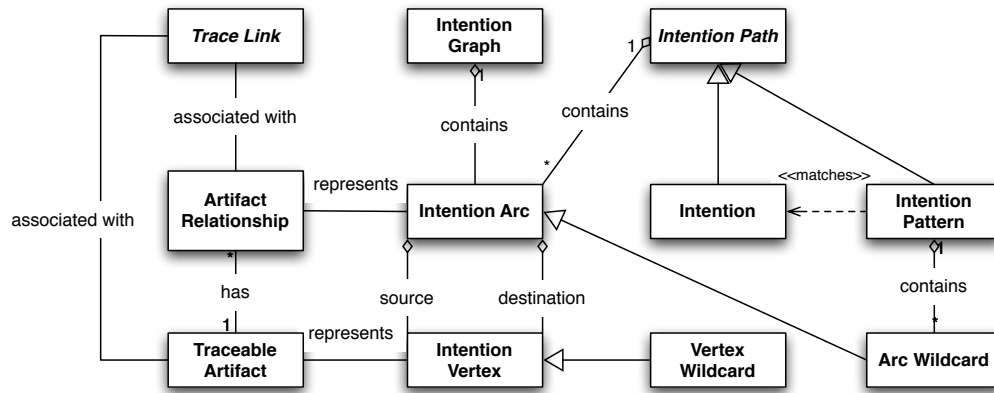


Figure 4.3: GAMBLE rejuvenation meta-model.

4.2.3 Concern Graph Topology

The meta-model planning corresponding to GAMBLE's general concern graph is depicted in Figure 4.3. The portrayed meta-model depicts relationships between elements of the concern graph, traceable elements contained in the evolving artifact base, and trace links. As previously mentioned, a traceable artifact will be represented by a vertex in the graph; relationships between artifacts will be represented by directed arcs. A single artifact may have multiple relationships of varying types with multiple artifacts. Thus, a concern graph would manifest itself as a directed multigraph of labeled arcs. Each arc would be associated with a source and target vertex, as well as label depicting the relationship between the two.

Recall that each finite acyclic path in the directed graph would represent an *intention* of where a trace link may apply. Informally, a developer *intention* is an aim or goal the developer has in mind when creating and/or maintaining software artifacts to realize (or help realize as in the case of, e.g., design) a particular requirement [FH92]. Intention patterns are to be derived from paths in the concern graph by replacing certain elements (vertices and/or arcs) with wildcard elements used to match new elements in the next version. The positions and semantics of the wildcards are to be decided by analyzing where the trace link applies in the original version of the artifact base.

4.2.4 Proposed Extensible Framework Architecture

Phase I Architecture

Figure 4.4 demonstrates the overall, extensible planned architecture for phase I to be provided to clients. We envision GAMBLE being utilized by traceability plugins, such as the ones discussed in this deliverable, in particular, by utilizing extension points in the Eclipse¹ plugin architecture. The numbered circles in the diagram indicate such planned extension points. Clients, i.e., traceability

¹<http://eclipse.org>

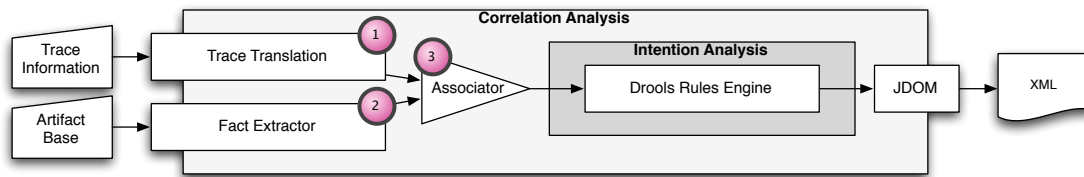


Figure 4.4: GAMBLE Phase I planned extensible architecture.

plugins, provide GAMBLE with three functional modules, a Trace Translator, a Fact Extractor, and an Associator. These modules may either be off-the-shell (i.e., commercial), or custom made for GAMBLE. Nonetheless, each module would serve as an input translation into the framework.

Extension point 1, Trace Translation, referred to as EP_1 , acts as the input processor for the traceability information. Traceability may come in many forms from many different kinds of plugins. Thus, EP_1 translates the traceability information into a form that can be understood by the Associator (EP_3) can understand, and corresponding match against the output of the Fact Extractor (EP_2).

Extension point 2, Fact Extractor, referred to as EP_2 , also acts as input processor, taking input as a representation of the artifact base. EP_2 derives facts about the artifact base, i.e., entities and their relationships to be fed into EP_3 . These information is ultimately used to build the concern graph discussed in the previous section.

Extension point 3, Associator, referred to as EP_3 , serves to associate the trace link information produced by EP_1 with the facts, i.e., entities and relationships, produced by EP_2 . Essentially, as depicted in the meta-model portrayed in Figure 4.3, a trace link associates an element of an artifact produced in a previous phase with an element of an artifact produced in a later phase, the transitive source being that of a concern. Thus, these elements may consist of either entities or relationships as produced by EP_2 .

The associated data emanating from EP_3 serves as input the internal Intention Analysis (IA) module provided by GAMBLE. The IA module is responsible for building the concern graph from the input information, associating graph elements according the trace information, and finally deriving intention patterns from the graph and trace links. The actual path matching is planned to be implemented via the Drools² rules engine, which provides not only an efficient solution but also performance benefits in cases where the tool is consequently run for multiple trace information against a common artifact base. Once the patterns have been derived, we plan to store them for latter use in phase II via an eXtensible Markup Language (XML) format, possibly with the aid of a Java Domain Object Model (JDOM)³ translation service.

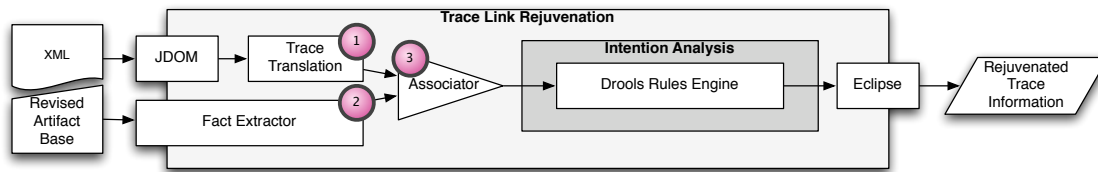


Figure 4.5: GAMBLE Phase II planned extensible architecture.

Phase II Architecture

Figure 4.5 demonstrates the plans for the overall, extensible architecture for phase II to be provided to clients. Input to the second phase would consist of the XML file containing the analysis results from phase I (see Section 4.2.4, as well as the *new* version of the artifact base. Notice that the same extension points are used here in phase II as was in phase I, specifically, EP₁, EP₂, and EP₃. The extension points feed into the Associator which matches *old* trace links with the facts about the *new* (or revised) artifact base. IA would then be repeated using this information. Finally, the results would be displayed to the user, possibly via the Eclipse framework. The display may include a table of suggested trace links sorted by decreasing order of their confidence.

4.2.5 Specific Approach and Implementation: Pointcut Rejuvenation

Although development of a general traceability link rejuvenation framework is still in its conceptual stages, a specific approach has been developed to rejuvenate pointcut expressions in Aspect-Oriented [KLM⁺97] software. Aspect-Oriented Programming (AOP) has emerged to reduce the scattering and tangling of crosscutting concern (CCCs) implementations. This is achieved through specifying that certain behavior (advice) should be composed at specific execution points (join points) which are quantified via pointcut expressions (PCEs). A PCE does so by logically connecting various predicates over static and dynamic program characteristics like method/field names and execution control-flow. An optimal PCE is one that truly conveys the *essence* of where a CCC applies in the base-code (the code under the influence of advice) so that not only are current join points selected but future ones as well.

Recall that advice is created in order to materialize the implementation of a concern that crosscuts the underlying base-code; thus, the PCE bound to the advice should quantify over the join points that correspond to this CCC. Similar to traceability links, PCEs can be viewed as a mapping between the concern realized by the advice and where the concern applies in the base-code. Thus, PCEs likewise require similar maintenance tasks in the face of software evolution. In fact, this task has been coined in this paradigm as the *fragile*

²<http://www.jboss.org/drools>

³<http://jdom.org>

pointcut problem [KS04], and can manifest itself such circumstances with join points incorrectly falling in or out of the scope of the PCEs.

Designing a robust PCE is often considered a “dark-art” with multiple design choices available to the developer. For example, if all method executions within a class named `Foo` need to be advised, two different strategies could be employed:

- (i) a generic PCE could be specified that quantifies over all methods (e.g., `execution(* Foo.*(..))`), OR
- (ii) each method could be enumerated individually (e.g., `execution(* Foo.methodA())`).

Deciding which strategy is best in order to balance robustness, correctness, and precision is non-trivial task. Apart from simple PCEs like the ones just mentioned, it is often impossible to ascertain prior to making maintenance changes whether or not the PCE is in fact robust. In the following sections, we will overview the motivation and algorithmic design for an automated approach to rejuvenating PCEs in the midst of AO software evolution.

Motivating Example

Figure 4.6 shows an example control system for a hybrid-powered vehicle which has two different power sources: a diesel engine (line 19) and electric motor (line 25). Calls to `DieselEngine.increase(Fuel)` (line 21) or `ElectricMotor.increase(Current)` (line 27) increase the vehicle’s speed (line 2) with the `HybridAutomobile` notified to compute the new speed (lines 6–7, 12–13).

Suppose now that certain highways contain a feature that notifies hybrid vehicles of the speed limit. As such, the code portrayed in Figure 4.6 is augmented by an aspect `SpeedingViolationPrevention` (Figure 4.7) to prevent speeding. It does so via `around` advice (lines 1–3). The PCE (line 2) specified to compose this advice selects join points corresponding to the execution of `DieselEngine.increase(Fuel)` and `ElectricMotor.increase(Current)`. Class `Energy` (not shown) is an abstract super class which both classes `Fuel` and `Current` (also not shown) extend. The type pattern `Energy+` is a wildcard that denotes object references of type `Energy` and its subclasses.

Further suppose that the base-code must evolve to add a new fuel cell energy source. Consequently, a `FuelCell` class (Figure 4.8) is created. Requests to increase power from the `FuelCell` require passing a numerical parameter (e.g., `double`) to a method (line 38) representing the amount of energy the fuel cell is to generate. Intuitively, the `SpeedingViolationPrevention` aspect should also apply to the execution of this method. However, the PCE listed on line 2, Figure 4.7 fails to capture this new but *semantically equivalent* join point. Although the new method’s signature is consistent with the other join points with only the parameter type differing, i.e., `double` is a primitive type that could not hold references to type `Energy` or any of its sub-classes, this difference causes the PCE not to match this method. It would be helpful to developers to *automatically* discover such join points that may be overlooked when manually updating PCEs to reflect new

```

1 class HybridAutomobile {
2   private double overallSpeed;
3
4   //Sets the new speed for changes in fuel.
5   public void notifyChangeIn(Fuel fuel) {
6     this.overallSpeed +=
7       fuel.calculateDeltaInMPH(this);
8     /* Update attached observers ... */
9
10    //Sets the new speed for changes in electricity .
11    public void notifyChangeIn(Current current) {
12      this.overallSpeed +=
13        current.calculateDeltaInMPH(this);
14      /* Update attached observers ... */
15
16    public double getOverallSpeed() {
17      return overallSpeed;}}
18
19 class DieselEngine {
20   private HybridAutomobile car;
21   public void increase(Fuel fuel) {
22     // ...
23     this.car.notifyChangeIn(fuel);}}
24
25 class ElectricMotor {
26   private HybridAutomobile car;
27   public void increase(Current current) {
28     // ...
29     this.car.notifyChangeIn(current);}}
30
31 class Dashboard {
32   private HybridAutomobile car;
33   public void update() {
34     // ...
35     this.display(car.getOverallSpeed()); }}

```

Figure 4.6: Hybrid automobile example.

```

1 Object around() :
2   execution(void increase(Energy+))
3   { /* ... */ }

```

Figure 4.7: Speeding prevention aspect.

changes in the base-code. In the following sections, we will continue to use the hybrid-powered vehicle example to show how our proposed approach can be used to automatically identify such new join points.

Algorithm

In this section, we present our inferencing algorithm that serves as a mechanism for unveiling developers' underlying intent in capturing certain program elements in a PCE. The algorithm utilizes a concept similar to that of a *concern graph* [RM02] extended with several elements found in current Java languages, e.g., annotations⁴, and adapted for use with AOP. Concern graphs have been

⁴During our empirical studies, we found that augmenting concern graphs with program elements found in current Java languages to be especially usefully in uncovering the essence of

```
36 class FuelCell {
37     private HybridAutomobile car;
38     public void increase(double amount) {
39         // ...
40         Current current =
41             this.generateCurrent(amount);
42         this.car.notifyChangeIn(current);}}
```

Figure 4.8: A new fuel cell class.

previously used to discover, describe, and track concerns in evolving source code [Rob06]. Our goal, however, is to exhaustively exploit rich, structural relationships between elements corresponding to captured join points, extract general patterns related to this data (as inspired by [DBWR07]), and finally apply these patterns to later versions of the software in order to accurately recover PCEs.

Assumptions The algorithm presented in this section works under a key assumption that the initial PCE to be analyzed and later rejuvenated is specified correctly. Specifically, we assume that advice is created in order to materialize the implementation of a concern that crosscuts the underlying base-code; thus, the PCE bound to the advice should quantify over the join points that correspond to this CCC. In fact, we treat the bound PCE as a mapping between the concern realized by the advice and where the concern applies in the base-code. This assumption is crucial in the ability of the algorithm to successfully infer new join points that should be incorporated in a revised version of the PCE.

For the sake of presentation, we further make several simplifying assumptions about the underlying source code to be analyzed; we discuss in Section 4.2.5 how much of these are relaxed in our implementation.

- We assume that inter-type declarations (static crosscutting) is not used in the analyzed aspects.
- Although it is possible for a PCE to capture join points associated within an advice body (possibly the one it is bound to), we adopt the perspective that aspects are indeed separate from the base-code; advice may only apply to join points associated with classes, interfaces, and other Java types.
- We assume that we are able to statically identify all references to program entities contained within the base-code. This assumption could be invalidated through the use of reflection and custom class loaders.
- We assume that the original source code successfully compiles under an AspectJ Development Tools⁵ (AJDT) 1.6 compiler.

join points underlying a certain PCE. The reason being is that annotations, in particular, are used, surprisingly, rather often as a mechanism to specify where a CCC should apply.

⁵<http://www.eclipse.org/ajdt>

ω	a join point shadow; code corresponding to a join point
\mathcal{A}	a piece of advice
\mathcal{A}_{pce}	a pointcut bound to advice \mathcal{A} ; a set of join point shadows
$\mathcal{A}_{pce'}$	a subsequent revision of \mathcal{A}_{pce}
\mathcal{P}	the original program, the underlying base-code
\mathcal{P}'	a subsequence revision of program \mathcal{P}
$\Omega_{\mathcal{P}}$	the set of join point shadows contained within program \mathcal{P}
$CG_{\mathcal{P}}^+$	a finite graph representing structural relationships between program elements in \mathcal{P}
π	an acyclic path (sequence of arcs) in $CG_{\mathcal{P}}^+$; an <i>intention</i>
$\Pi_{\mathcal{P}}$	a set of acyclic paths derived from $CG_{\mathcal{P}}^+$
$\hat{\pi}$	an intention <i>pattern</i> derived from π , possibly containing wildcards
$\hat{\Pi}_{\mathcal{P}}$	a set of intention patterns derived from $CG_{\mathcal{P}}^+$

Figure 4.9: Algorithm formalism notation.

Recall that a join point refers to a well-defined point in the execution of the base-code; thus, the definition of a join point is dynamic in nature. A *join point shadow*, on the other hand, refers to base-code corresponding to a join point [XR08], i.e., a point in the program text where the compiler may actually perform the weaving [MKD03]. Whether or not the base-code is actually advised at that point is dependent on (i) advice being applicable at that point, and (ii) possible dynamic conditions being met. As such, we consider a given program \mathcal{P} as consisting of a set of join point shadows $\Omega_{\mathcal{P}}$ (see Figure 4.9) that *may or may not* be currently under the influence of advice⁶. Moreover, a given piece of advice \mathcal{A} specifies its applicability to a base program \mathcal{P} via its bound pointcut expression \mathcal{A}_{pce} , which selects a subset of shadows contained within \mathcal{P} , i.e., \mathcal{A} applies to \mathcal{P} at $\Omega_{\mathcal{P}} \cap \mathcal{A}_{pce}$. Therefore, each $\omega \in \Omega_{\mathcal{P}} \cap \mathcal{A}_{pce}$ specifies *where* \mathcal{A} should apply to \mathcal{P} but does not specify *when*. That is, we assume that no dynamic conditions are associated with \mathcal{A}_{pce} ; thus, we utilize solely static information in our analysis. Section 4.2.5 discusses how our implementation conservatively relaxes this assumption so that PCEs utilizing dynamic conditions may nevertheless be used as input to our tool.

Lastly, we assume that we can accurately resolve the declaration of a particular piece of advice across varying versions of the software. This assumption is important to our analysis since, in AO languages like AspectJ, advice is considered to be anonymous, which may make it difficult to track it in subsequent versions.

Pointcut Rejuvenation Figure 4.10 depicts the *Rejuvenate* function, the top-level pointcut rejuvenation algorithm that drives the approach. Input to the function is a pointcut \mathcal{A}_{pce} from the original program \mathcal{P} to be recovered as a result of the new version of the program \mathcal{P}' . Conceptually, applying this function to the example given in Section 4.2.5, we would take \mathcal{A}_{pce} to be the PCE declared on line 2, Figure 4.7, \mathcal{P} to be the sequence of classes depicted in Figure 4.6, and

⁶This definition differs slightly from those given in the literature.

```

function Rejuvenate( $\mathcal{A}_{pce}, \mathcal{P}, \mathcal{P}', d$ )
1:  $CG_{\mathcal{P}}^+ \leftarrow BuildGraph(\mathcal{P})$  /*Construct the extended concern graph for the original program*/
2:  $\hat{\Pi}_{\mathcal{P}} \leftarrow CreatePatterns(\mathcal{A}_{pce}, CG_{\mathcal{P}}^+, d)$  /*Derive intention patterns relevant to the PCE from the graph of the original program*/
3:  $CG_{\mathcal{P}'}^+ \leftarrow BuildGraph(\mathcal{P}')$  /*Construct the extended concern graph for the revised program*/
4:  $\hat{\Pi}_{\mathcal{P}'}$   $\leftarrow CreatePatterns(\Omega_{\mathcal{P}'}, CG_{\mathcal{P}'}^+, d)$  /*Derive all possible intention patterns from the graph of the revised program*/
5:  $\hat{\Pi}_{\mathcal{P} \cap \mathcal{P}'} \leftarrow \hat{\Pi}_{\mathcal{P}} \cap \hat{\Pi}_{\mathcal{P}'}$  /*Intersect the patterns derived from the old version with the ones from the new version.*/
6:  $S \leftarrow MakeSuggestions(\hat{\Pi}_{\mathcal{P} \cap \mathcal{P}'}, CG_{\mathcal{P}'}^+)$  /*Create a set of suggestion, confidence pairs*/
7:  $\mathcal{A}_{pce'} \leftarrow \emptyset$  /*The rejuvenated PCE to be returned*/
8: for all  $(\omega, c) \in Sort(S)$  do /*For all suggestion, confidence pairs by descending confidence*/
9:   Suggest( $\omega, c$ )
10:  if Selected( $\omega$ ) then
11:     $\mathcal{A}_{pce'} \leftarrow \mathcal{A}_{pce'} \cup \{\omega\}$ 
12:  end if
13: end for
14: return  $\mathcal{A}_{pce'}$ 

```

Figure 4.10: Top-level rejuvenation algorithm.

\mathcal{P}' to be \mathcal{P} concatenated with the `FuelCell` class found in Figure 4.8. Parameter d represents the *maximum analysis depth* which serves to restrict the depth of structural program relationships analyzed, thus limiting the length (in the number of arcs) of the patterns produced by the algorithm. Line 1 constructs an adaptation of a concern graph $CG_{\mathcal{P}}^+$ using program elements from the original program \mathcal{P} . We now specify the graph more precisely; Section 4.2.5 discusses how the graph was generated in our prototype implementation.

Concern Graphs A representation similar to that of a concern graph is adapted and extended in our approach to help uncover the essence of shadows captured by a particular PCE. Each finite, acyclic path in the directed graph represents a developer’s *intention* of where a CCC may apply in the source code. Informally, a developer *intention* is an aim or goal the developer has in mind when creating and/or maintaining programming elements to realize a particular requirement [FH92]. For instance, consider the code of the hypothetical hybrid automobile example given in Section 4.2.5. Here, the developer has written a piece of advice (Figure 4.7, lines 1–3) that is *intended* to “advise the executions of methods that are responsible for contributing to overall speed of the vehicle in order to bypass them under certain conditions” (I_1). To carry out this across the software that are responsible for this behavior, and writes a suitable PCE (e.g., Figure 4.7, line 2) that captures the executions of these methods. Of course, exactly how this

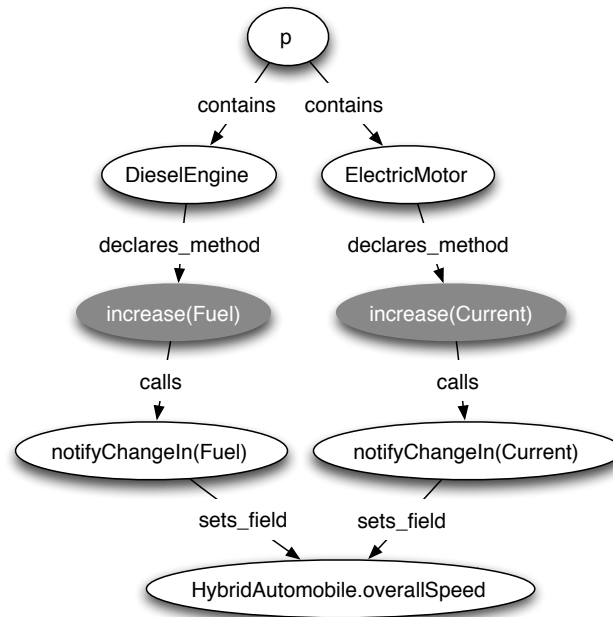


Figure 4.11: A subset of $CG_{\mathcal{P}}^+$ computed from the motivating example.

intention is encoded may not be unique and is largely dependent on the expressiveness of the available PCE language, *as well* as the developer’s expertise with that language.

Relating this to our example, the intention I_1 has been encoded using the PCE `execution(void increase(Energy+))`. We can rephrase this intention in another way, e.g., “to advise the executions of methods which possibly call methods that possibly write (or set) the field `HybridAutomobile.overallSpeed`” (I_2). Although both intentions I_1 and I_2 share a common goal in advising the same set of method executions, the latter expresses the intention using various low-level program elements and structural characteristics existing at compile time. Thus, we can represent I_2 diagrammatically as portrayed in Figure 4.11, which depicts a subset of an extended concern graph computed from the motivating example given in Section 4.2.5. Here, I_2 is encoded in terms of elements of the graph manifested as two finite, acyclic paths $\text{increase(Fuel)} \rightsquigarrow \text{overallSpeed}$ and $\text{increase(Current)} \rightsquigarrow \text{overallSpeed}$. The highlighted vertices denote executions of the represented methods as being “selected” (or enabled) by the encoding.

Specification We now specify the extended concern graph more formally utilizing the notation in Figure 4.12.

Definition 1 An extended concern graph $CG_{\mathcal{P}}^+$ constructed from program \mathcal{P} is a labeled multidigraph consisting of a 4-tuple $CG_{\mathcal{P}}^+ = (V, A, R, \ell)$ where

- $V = \phi_{\mathcal{P}} \cup \mu_{\mathcal{P}} \cup \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \iota_{\mathcal{P}} \cup \gamma_{\mathcal{P}} \cup \nu_{\mathcal{P}}$ is a set of vertices representing the declarations of program elements contained in \mathcal{P} ⁷, e.g., packages,

⁷The program parameter is implicit in this definition in order to simplify the presentation.

$\phi_{\mathcal{P}}$	$\{f \mid f \text{ is a field declared in } \mathcal{P}\}$
$\mu_{\mathcal{P}}$	$\{m \mid m \text{ is a method declared in } \mathcal{P}\}$
$\psi_{\mathcal{P}}$	$\{c \mid c \text{ is a class declared in } \mathcal{P}\}$
$\varepsilon_{\mathcal{P}}$	$\{e \mid e \text{ is an enumeration type declared in } \mathcal{P}\}$
$\iota_{\mathcal{P}}$	$\{i \mid i \text{ is an interface declared in } \mathcal{P}\}$
$\nu_{\mathcal{P}}$	$\{a \mid a \text{ is an annotation type declared in } \mathcal{P}\}$
$\gamma_{\mathcal{P}}$	$\{p \mid p \text{ is a package declared in } \mathcal{P}\}$

Figure 4.12: Intention graph construction formalism notation.

classes, interfaces, enumeration types, annotations, methods, fields,

- $A = \{(u, v) \mid u, v \in V \wedge \exists [r \in R \mid r(u, v)]\}$ is a multiset of arcs connecting vertices in V ,
- R (see definition 2) is a set of binary relations depicting structural relationships existing amongst program elements in \mathcal{P} at compile time,
- $\ell: A \rightarrow R$ (see definition 3) is a labeling function labeling arcs with the relationships in R that elements represented by their source and target vertices, respectively, satisfy.

The set of vertices V of $CG_{\mathcal{P}}^+$ represent declarations of various program element, e.g., packages, classes, interfaces, enumeration types, annotations, methods, fields⁸. The set of arcs A connect vertices in V depending on the truth value of relations found in the set R when applied to the source and target vertices, respectively. The relations in R , details of which are specified more formally in definition 2, represent various structural relationships, e.g., method calls⁹, field accesses. Many such kinds of relationships may exist, however, for simplicity, we mainly focus on several popular relationship types as previously utilized in the literature [DBWR07, BGKV06, RM02, Rob06]. Arcs are then labeled with the satisfied relations via the labeling function ℓ , which is specified more formally in definition 3. Notice that, in this example, the analysis performed utilizes solely static information, in particular, class hierarchical analysis (CHA) [DGC95] is used to identify method call relationships. Section 4.2.6 touches upon future work which may potentially result in a more accurate estimate of the truth values associated with such relationships.

Definition 2 *A set of relations R of binary predicates, i.e., relations, over pro-*

⁸We do not consider local variables and other parameters in our analysis as crosscutting concerns tend to crosscut a larger granularity of programming elements.

⁹For simplicity of presentation, our formalism groups class instance creations, i.e., constructor calls, with method calls.

gram elements is the set

$$\begin{aligned}
 R = \{ & \text{GetsField}: \mu_{\mathcal{P}} \times \phi_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{SetsField}: \mu_{\mathcal{P}} \times \phi_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{CallsMethod}: \mu_{\mathcal{P}} \times \mu_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{OverridesMethod}: \mu_{\mathcal{P}} \times \mu_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{ImplementsMethod}: \mu_{\mathcal{P}} \times \mu_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{DeclaresMethod}: \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \iota_{\mathcal{P}} \times \mu_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{DeclaresField}: \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \iota_{\mathcal{P}} \times \phi_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{DeclaresType}: \psi_{\mathcal{P}} \cup \iota_{\mathcal{P}} \cup \nu_{\mathcal{P}} \times \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \iota_{\mathcal{P}} \cup \nu_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{ExtendsClass}: \psi_{\mathcal{P}} \times \psi_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{ExtendsInterface}: \iota_{\mathcal{P}} \times \iota_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{ImplementsInterface}: \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \times \iota_{\mathcal{P}} \cup \nu_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{ContainsType}: \gamma_{\mathcal{P}} \times \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \iota_{\mathcal{P}} \cup \nu_{\mathcal{P}} \rightarrow \mathbb{B}, \\
 & \text{Annotates}: \nu_{\mathcal{P}} \times \gamma_{\mathcal{P}} \cup \nu_{\mathcal{P}} \cup \iota_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \psi_{\mathcal{P}} \cup \mu_{\mathcal{P}} \cup \phi_{\mathcal{P}} \rightarrow \mathbb{B} \}
 \end{aligned}$$

Definition 3 A labeling function $\ell: A \rightarrow R$ labels arcs corresponding to the predicates in R their constituent vertices satisfy such that

$$\ell(u, v) = \left\{ \begin{array}{ll}
 \text{GetsField} & \iff u \in \mu_{\mathcal{P}} \wedge v \in \phi_{\mathcal{P}} \wedge \text{GetsField}(u, v) \\
 \text{SetsField} & \iff u \in \mu_{\mathcal{P}} \wedge v \in \phi_{\mathcal{P}} \wedge \text{SetsField}(u, v) \\
 \text{CallsMethod} & \iff u \in \mu_{\mathcal{P}} \wedge v \in \mu_{\mathcal{P}} \wedge \text{CallsMethod}(u, v) \\
 \text{OverridesMethod} & \iff u \in \mu_{\mathcal{P}} \wedge v \in \mu_{\mathcal{P}} \wedge \text{OverridesMethod}(u, v) \\
 \text{ImplementsMethod} & \iff u \in \mu_{\mathcal{P}} \wedge v \in \mu_{\mathcal{P}} \wedge \\
 & \text{ImplementsMethod}(u, v) \\
 \text{DeclaresMethod} & \iff u \in \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \iota_{\mathcal{P}} \wedge v \in \mu_{\mathcal{P}} \wedge \\
 & \text{DeclaresMethod}(u, v) \\
 \text{DeclaresField} & \iff u \in \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \iota_{\mathcal{P}} \wedge v \in \phi_{\mathcal{P}} \wedge \\
 & \text{DeclaresField}(u, v) \\
 \text{DeclaresType} & \iff u \in \psi_{\mathcal{P}} \cup \iota_{\mathcal{P}} \cup \nu_{\mathcal{P}} \wedge v \in \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \iota_{\mathcal{P}} \cup \nu_{\mathcal{P}} \\
 & \wedge \text{DeclaresType}(u, v) \\
 \text{ExtendsClass} & \iff u \in \psi_{\mathcal{P}} \wedge v \in \psi_{\mathcal{P}} \wedge \text{ExtendsClass}(u, v) \\
 \text{ExtendsInterface} & \iff u \in \iota_{\mathcal{P}} \wedge v \in \iota_{\mathcal{P}} \wedge \text{ExtendsInterface}(u, v) \\
 \text{ImplementsInterface} & \iff u \in \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \wedge v \in \iota_{\mathcal{P}} \cup \nu_{\mathcal{P}} \wedge \\
 & \text{ImplementsInterface}(u, v) \\
 \text{ContainsType} & \iff u \in \gamma_{\mathcal{P}} \wedge v \in \psi_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \iota_{\mathcal{P}} \cup \nu_{\mathcal{P}} \wedge \\
 & \text{ContainsType}(u, v) \\
 \text{Annotates} & \iff u \in \nu_{\mathcal{P}} \wedge v \in \gamma_{\mathcal{P}} \cup \nu_{\mathcal{P}} \cup \iota_{\mathcal{P}} \cup \varepsilon_{\mathcal{P}} \cup \psi_{\mathcal{P}} \cup \mu_{\mathcal{P}} \\
 & \cup \phi_{\mathcal{P}} \wedge \text{Annotates}(u, v)
 \end{array} \right.$$

```

function CreatePatterns( $\mathcal{A}_{pce}$ ,  $CG_{\mathcal{P}}^+ = (V, A, R, \ell)$ ,  $d$ )
1:  $T \leftarrow \emptyset$  /*The set of patterns to be returned, initially empty*/
2: for all  $v \in V$  do /*For all vertices in  $CG_{\mathcal{P}}^+$ */
3:   if EnabledVertex( $v$ ,  $\mathcal{A}_{pce}$ ) then /*If associated with the PCE*/
4:     for all  $\pi \in PathsThrough(v, d)$  do /*For all acyclic paths of length  $\leq d$ 
      passing through  $v$ */
5:        $\hat{\Pi} \leftarrow ExtractVertexPatterns(\pi, v)$  /*Obtain vertex patterns from  $\pi$  using
       $v$ */
6:        $T \leftarrow T \cup \hat{\Pi}$ 
7:     end for
8:   end if
9: end for
10: for all  $(u, v) \in A$  do /*For all arcs in  $CG_{\mathcal{P}}^+$ */
11:   if EnabledArc( $u, v$ ,  $\mathcal{A}_{pce}$ ) then /*If associated with the PCE*/
12:     for all  $\pi \in PathsAlong(u, v, d)$  do /*For all acyclic paths of length  $\leq d$ 
      along  $(u, v)$ */
13:        $\hat{\Pi} \leftarrow ExtractArcPatterns(\pi, u, v)$  /*Obtain arc patterns from  $\pi$  using
       $(u, v)$ */
14:        $T \leftarrow T \cup \hat{\Pi}$ 
15:     end for
16:   end if
17: end for
18: return  $T$ 

```

Figure 4.13: Intention pattern creation algorithm.

Intention Patterns Once $CG_{\mathcal{P}}^+$ is constructed, the next step is to identify *intention patterns* that express general *shapes* of paths (i.e., intentions) within the graph from elements associated with shadows currently selected by the given PCE. Returning to the *Rejuvenate* function depicted in Figure 4.10, *intention patterns* of a maximum length of d are derived (line 2) from finite, acyclic paths in $CG_{\mathcal{P}}^+$ which are relevant to the input PCE. The purpose of these patterns is to approximate the *essence* of the developer’s intentions behind the original PCE.

The invoked function *CreatePatterns*, whose definition is depicted in Figure 4.13, initializes a set T to be a empty set of patterns to be returned at line 1. The algorithm then proceeds to cycle through each vertex (line 2) and arc (line 10), searching for graph elements that are considered *enabled* (or selected) by the given PCE. It does so via a vertex-pointcut association relation *EnabledVertex*: $V \times \mathcal{P}(\Omega_{\mathcal{P}}) \rightarrow \mathbb{B}$ and a arc-pointcut association relation *EnabledArc*: $A \times \mathcal{P}(\Omega_{\mathcal{P}}) \rightarrow \mathbb{B}$, respectively, where \mathcal{P} denotes the power set operation and \mathbb{B} the set of boolean values, determines whether a graph element is associated with a given PCE.

Intuitively, *EnabledVertex*, given a vertex v and a PCE \mathcal{A}_{pce} , is true iff there exists a shadow $\omega \in \mathcal{A}_{pce}$ such that ω corresponds to the program element e represents. Likewise, *EnabledArc*, given either an arc (u, v) and a PCE \mathcal{A}_{pce} , is true iff there exists a shadow $\omega \in \mathcal{A}_{pce}$ such that ω corresponds to the pro-

gram element relationship $\ell(u, v)$ represents. For instance, if a PCE contains a method execution shadow for a method x , a vertex in $CG_{\mathcal{P}}^+$ representing x is considered enabled w.r.t. the PCE. Conversely, if a PCE contains a method call shadow from a method y to a method z , then an arc between the vertices representing y and z with the label *CallsMethod* is considered enabled w.r.t. the PCE. Relating this to our motivating example given in Section 4.2.5, the PCE `execution(void increase(Energy+))` captures the execution of two methods, namely, `DieselEngine.increase(Fuel)` and `ElectricMotor.increase(Current)`. As such, we assign dual semantics to different kinds of graph elements in order to relate them to PCEs. In this case, vertices representing method declarations also happen to represent the shadow corresponding to each method's execution. To illustrate this notion, the subset of $CG_{\mathcal{P}}^+$ depicted in Figure 4.11 portrays the vertices representing these methods as shaded, denoting that the *Enabled* relation is true for the combination of each method and the given PCE.

We now specify the *EnabledVertex* relation more formally.

Definition 4 For an extended concern graph $CG_{\mathcal{P}}^+ = (V, A, R, \ell)$, a vertex $v \in V$, and a PCE $\mathcal{A}_{pce} \in \mathcal{P}(\Omega_{\mathcal{P}})$, we have that $EnabledVertex(v, \mathcal{A}_{pce}) \iff v \in \mu_{\mathcal{P}} \wedge \exists [\omega \in \mathcal{A}_{pce} \mid \omega = \mathbf{execution}(v)]$.

As definition 4 depicts, a vertex is considered enabled w.r.t. to a given PCE iff the vertex represents a method whose corresponding execution join point shadow is currently being advised by the PCE. Note that $\mathbf{execution}(v)$ denotes the method execution join point shadow corresponding to the method represented by the vertex v .

We now specify the *EnabledArc* relation more formally.

Definition 5 For an extended concern graph $CG_{\mathcal{P}}^+ = (V, A, R, \ell)$, an arc $(u, v) \in A$, and a PCE $\mathcal{A}_{pce} \in \mathcal{P}(\Omega_{\mathcal{P}})$, we have that

$$\begin{aligned}
 EnabledArc(u, v, \mathcal{A}_{pce}) \iff & \ell(u, v) = GetsField \wedge \\
 & \exists [\omega \in \mathcal{A}_{pce} \mid \omega = \mathbf{get}(v) \ \&\& \ \mathbf{withincode}(u)] \\
 & \forall \ell(u, v) = SetsField \wedge \\
 & \exists [\omega \in \mathcal{A}_{pce} \mid \omega = \mathbf{set}(v) \ \&\& \ \mathbf{withincode}(u)] \\
 & \forall \ell(u, v) = CallMethod \wedge \\
 & \exists [\omega \in \mathcal{A}_{pce} \mid \omega = \mathbf{call}(v) \ \&\& \ \mathbf{withincode}(u)]
 \end{aligned}$$

As definition 5 depicts, an arc is considered enabled w.r.t. to a given PCE iff either

- (i) the arc is labeled as a field read access *and* there exists a shadow in the given PCE s.t. the shadow represents a *field get* join point on the field represented by the target vertex v (denoted by $\mathbf{get}(v)$), and the shadow is located *within* the body of the method represented by the source vertex u (denoted by $\mathbf{withincode}(u)$), or

- (ii) the arc is labeled as a field write access *and* there exists a shadow in the given PCE s.t. the shadow represents a *field set* join point on the field represented by the target vertex v (denoted by $\text{set}(v)$), and the shadow is located *within* the body of the method represented by the source vertex u (denoted by $\text{withincode}(u)$), or
- (iii) the arc is labeled as a method call *and* there exists a shadow in the given PCE s.t. the shadow represents a *method call* join point of which the called method is that of the method represented by the target vertex v (denoted by $\text{call}(v)$), and the shadow is located *within* the body of the method represented by the source vertex u (denoted by $\text{withincode}(u)$)

As future work, we plan to incorporate more AO shadow types into this scheme, e.g., `handler()`.

Once determined that a vertex v is indeed enabled by the given PCE, *CreatePatterns* (line 4) traverses each acyclic, finite path π of length $\leq d$ in $CG_{\mathcal{P}}^+$ passing through v . E.g., one such path in the case of the example given in Figure 4.11, when taking $e = \text{increase}(\text{Fuel})$ and $d = 2$ would be $\text{increaseFuel}(\text{Fuel}) \xrightarrow{cm} \text{notifyChangeIn}(\text{Fuel}) \xrightarrow{sf} \text{HybridAutomobile.overallSpeed}$, where the labels cm and sf refer to the satisfied binary relations in R , $\text{CallsMethod}: \mu_{\mathcal{P}} \times \mu_{\mathcal{P}} \rightarrow \mathbb{B}$ and $\text{SetsField}: \mu_{\mathcal{P}} \times \phi_{\mathcal{P}} \rightarrow \mathbb{B}$, respectively. The traversal process is similar for arcs (line 12), except that paths along the arc are considered, as opposed to passing through a specific vertex.

In the case of vertices, the function then proceeds (line 5) to obtain intention patterns from the path π under consideration using the enabled vertex as a guide, doing so via a helper function $\text{ExtractVertexPatterns}: \Pi_{\mathcal{P}} \times V \rightarrow \mathcal{P}(\hat{\Pi}_{\mathcal{P}})$. This function *extracts* a vertex-based intention pattern; a vertex-based *pattern* is similar in nature to the *path* it is extracted from except that certain vertices are replaced with *wildcard* elements. Wildcard elements are used for matching patterns with other paths in the graph in order to ultimately obtain suggested shadows. A vertex on the path may be replaced by a *vertex wildcard*, which only matches other vertices. A vertex wildcard may be *enabled* or *disabled* depending on its position in the path relative to the input vertex. Shadows represented by vertices matched by *enabled* wildcards are those that eventually become suggested.

In the case of arcs, the function proceeds on line 13 to obtain intention patterns from the path π under consideration using the enabled arc as a guide, doing so via a helper function $\text{ExtractArcPatterns}: \Pi_{\mathcal{P}} \times A \rightarrow \mathcal{P}(\hat{\Pi}_{\mathcal{P}})$. This function *extracts* an arc-based intention pattern; an arc-based *pattern* is similar in nature to the *path* it is extracted from except that certain vertices *and* arcs are replaced with *wildcard* elements. An arc on the path may be replaced by an *arc wildcard*, which only matches other arcs. Similar to a vertex wildcard, an arc wildcard may be *enabled* or *disabled* depending on its position in the path relative to the input arc. Shadows represented by arcs matched by *enabled* wildcards are those that eventually become suggested.

Intuitively, the functions works by replacing combinations of elements along

the path with wildcards depending on the position of the enabled graph element. Relating this concept to the example subset graph given Figure 4.11, suppose π , the path under consideration, is the one previously considered, namely, $\text{increaseFuel(Fuel)} \xrightarrow{cm} \text{notifyChangeIn(Fuel)} \xrightarrow{sf} \text{HybridAutomobile.overallSpeed}$, and suppose the enabled vertex is the first vertex $\text{increaseFuel(Fuel)}$. Thus, in this case, *ExtractVertexPatterns* would produce a set consisting of a single vertex pattern $\hat{\pi}$, namely, $?^* \xrightarrow{cm} ? \xrightarrow{sf} \text{HybridAutomobile.overallSpeed}$ where $?$ denotes a disabled wildcard and $?^*$ denotes an enabled wildcard. *CreatePatterns* finally returns the set of all such patterns T on line 18, Figure 4.13.

Returning to the *Rejuvenate* function depicted in Figure 4.10, lines 3 and 4 perform similar actions as the previous two, however, they do so for new version of the base-code \mathcal{P}' . Also notice that at line 4, patterns are derived by enabling *all* shadows contained in the revised program ($\Omega_{\mathcal{P}'}$), thus, all possible intention patterns are derived from $CG_{\mathcal{P}'}^+$. Then, at line 5, the patterns derived from the old version of the base-code are intersected with all possible patterns derived from the new version. The intersection $\hat{\Pi}_{\mathcal{P} \cap \mathcal{P}'}$ represents *surviving* set of patterns between the two versions, thereby removing patterns containing program elements that no longer exist in the new version of the base-code. On line 6, a set S of suggested shadow, confidence¹⁰ pairs is created to serve as suggested shadows to be included in a new version of the PCE. S is constructed by matching each intention pattern in the surviving set $\hat{\Pi}_{\mathcal{P} \cap \mathcal{P}'}$ with the the graph of the revised base-code $CG_{\mathcal{P}'}^+$.

We now define the pattern to path matching scheme of our approach more formally.

Definition 6 *A pattern $\hat{\pi}$ matches a path π iff*

- *for each vertex u along π at position i there exists a vertex v in $\hat{\pi}$ at the same position i in which either $u = v$ or v is a wildcard,*
- *for each arc (p, q) along π at position j there exists an arc (s, t) in $\hat{\pi}$ at the same position j in which either $\ell(p, q) = \ell(s, t)$ or (s, t) is a wildcard.*

In respect to a graph $CG_{\mathcal{P}}^+$, we define the equivalency relation over vertices of $CG_{\mathcal{P}}^+$ to be that of the traditional equality relation, i.e., that u and v refer to the *same* vertex. As for arcs, the function ℓ refers to the labeling function introduced in definition 1, thus, $\ell(p, q) = \ell(s, t)$ denotes that each arc satisfies the same structural relation. In terms the graph subset portrayed in Figure 4.11, the pattern $?^* \xrightarrow{cm} ? \xrightarrow{sf}$ would match $\text{increase(Fuel)} \rightsquigarrow \text{overallSpeed}$ and $\text{increase(Current)} \rightsquigarrow \text{overallSpeed}$. Also note that the same pattern would also match the path $\text{increase(double)} \rightsquigarrow \text{overallSpeed}$ had we augmented the graph in Figure 4.11 with the revised version of the base code portrayed in Figure 4.8 from our motivating example. Furthermore, notice that increase(double) would be represented by a vertex that would match an enabled wildcard element in the pattern, thus, this method would be suggested to be included in a new version of the PCE.

¹⁰The *confidence* factor is inspired by [DBWR07].

$$error_{\alpha}(\hat{\pi}, \mathcal{A}_{pce}) = \begin{cases} 0 & \text{if } |Match(\hat{\pi}, Paths(CG_{\mathcal{P}}^+))| = 0 \\ 1 - \frac{|\mathcal{A}_{pce} \cap Match(\hat{\pi}, Paths(CG_{\mathcal{P}}^+))|}{|Match(\hat{\pi}, Paths(CG_{\mathcal{P}}^+))|} & \text{otherwise} \end{cases} \quad (4.1)$$

$$error_{\beta}(\hat{\pi}, \mathcal{A}_{pce}) = \begin{cases} 1 & \text{if } |\mathcal{A}_{pce}| = 0 \\ 1 - \frac{|\mathcal{A}_{pce} \cap Match(\hat{\pi}, Paths(CG_{\mathcal{P}}^+))|}{|\mathcal{A}_{pce}|} & \text{otherwise} \end{cases} \quad (4.2)$$

$$abs(\hat{\pi}) = \begin{cases} 1 & \text{if } |\hat{\pi}| = 0 \\ 1 - \frac{|\hat{\pi}| - |\mathcal{W}(\hat{\pi})|}{|\hat{\pi}|} & \text{otherwise} \end{cases} \quad (4.3)$$

$$confidence(\hat{\pi}, \mathcal{A}_{pce}) = 1 - error_{\alpha}(\hat{\pi}, \mathcal{A}_{pce})(1 - abs(\hat{\pi})) + error_{\beta}(\hat{\pi}, \mathcal{A}_{pce})abs(\hat{\pi}) \quad (4.4)$$

Figure 4.14: Pattern attribute equations.

A confidence c paired with each suggested shadow ω is a real number in the interval $[0, 1]$ that represents the degree to which we believe a revised version of the original PCE to be applicable to the shadow ω . The confidence of a suggested shadow is inherited from the pattern in which it was produced and is derived from 3 components as depicted in Figure 4.14. We refer to each of these components as *pattern attributes* in respect to a PCE \mathcal{A}_{pce} , which is the PCE to be rejuvenated.

The first attribute is the $error_{\alpha}$ rate, depicted in equation (4.1), which is a ratio of the number of shadows captured by both the PCE \mathcal{A}_{pce} and the pattern $\hat{\pi}$ when applied to finite, acyclic paths in the graph $Paths(CG_{\mathcal{P}}^+)$ to the number of shadows captured by solely the pattern. The α signifies the metric's association with the rate of type I (or α) errors which relates to the number of false positives produced by the pattern. Conceptually, the $error_{\alpha}$ rate quantifies the pattern's ability in matching *solely* the shadows contained within the PCE; the closer the $error_{\alpha}$ is to 0, the more likely the shadows matched by the pattern are also ones contained within the PCE. It refers to the *quality* of results that the pattern is likely to produce in the future. A pattern with a low $error_{\alpha}$ is one that expresses a strong relationship amongst join points captured by the PCE; we would expect *future* join points to exhibit *similar* characteristics. Naturally, if a given pattern does not match any shadows, its corresponding $error_{\alpha}$ rate is 0.

The second pattern attribute in respect to a given PCE is the $error_{\beta}$ rate, depicted in equation (4.2), which is a ratio of the number of shadows captured by both the PCE \mathcal{A}_{pce} and the pattern $\hat{\pi}$ when applied to finite, acyclic paths in the graph $Paths(CG_{\mathcal{P}}^+)$ to the number of shadows captured by solely by the given PCE. The difference between $error_{\alpha}$ and $error_{\beta}$ is subtle but important; the β signifies the metric's association with the rate of type II (or β) errors which re-

lates to the number of false negatives produced by the pattern. Conceptually, the $error_{\beta}$ rate quantifies the pattern's ability in matching *all* of the shadows contained within the PCE; the closer the $error_{\beta}$ is to 0, the more likely the pattern is to match all the shadows contained within the PCE. It refers to the *quantity* of *correct* results that the pattern is likely to produce in the future. A pattern with a low $error_{\beta}$ expresses properties similar to the ones expressed by the given PCE, *whether or not* those properties are common to the captured shadows. Naturally, if the given PCE does not contain any shadows, the pattern's corresponding $error_{\beta}$ rate is 1 since it could not possibly match *any* of the join points contained within PCE.

As portrayed by function *CreatePatterns* in Figure 4.13, a pattern $\hat{\pi}$ is derived from a path π by replacing concrete elements in the path with wildcard elements. Wildcard graph elements may match a number of elements contained in the graph as detailed previously. When predicting a pattern's future ability to rejuvenate a given PCE, we would like to take into account its *abstractness* (abbreviated by *abs*), i.e., the ratio of the number of constituent wildcard elements to concrete elements. Let $|\hat{\pi}|$ denote the number of unique elements (vertices and arcs), including wildcards, contained within pattern $\hat{\pi}$. Moreover, let $\mathcal{W}(\hat{\pi})$ denote the multiset projection of wildcard elements contained in pattern $\hat{\pi}$. Likewise, $|\mathcal{W}(\hat{\pi})|$ projects the number of wildcard elements contained within pattern $\hat{\pi}$. Then, the *abs* of a pattern $\hat{\pi}$, which is independent of any particular PCE, is given by equation (4.3). Note that an empty pattern has no concrete elements, thus, we consider such a pattern to be completely abstract, i.e., having an abstractness of 1.

The corresponding intuition behind a pattern *abstractness* attribute is that patterns containing many wildcard elements are more likely to match a greater number of concrete graph elements and vice versa. Thus, we would like to combine the α and β error rates of a pattern by use of a weighted mean weighted by the pattern's abstractness. The intuition behind the chosen weighting scheme is as follows. A pattern that is very abstract (i.e., containing many wildcards) is typically less likely to hone in on shadows that are *only* contained within the given PCE. Conversely, a pattern that is less abstract (i.e., more *concrete*, containing fewer wildcards) is less likely to cover all shadows contained within the given PCE. The combined metrics are used to derive a *confidence* pattern attribute, portrayed in equation (4.4), which is a convenient, single metric to judge the *confidence* we have in the pattern being useful in detecting accurate shadows to be included in a future, rejuvenated version of the corresponding PCE. The closer a pattern's confidence is to 1, the more likely it will produce accurate suggestions in the future.

Returning to the *Rejuvenate* function depicted in Figure 4.10, line 7 commences the final rejuvenation process by initializing the new PCE $\mathcal{A}_{pce'}$ to be returned by the function as the empty set of shadows. Then, for each shadow, confidence pair sorted by decreasing confidence (line 8), the suggestion is presented to developer along with its confidence (line 9). The selected shadows (line 10) are then augmented to the rejuvenated PCE $\mathcal{A}_{pce'}$ (line 11) and returned (line 14).

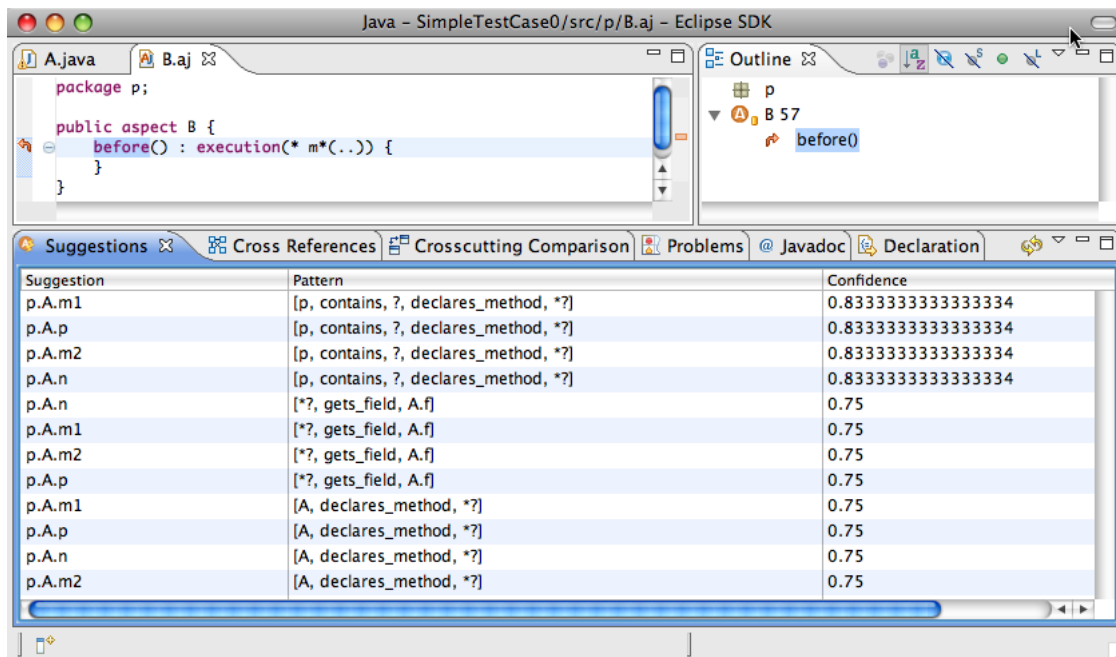


Figure 4.15: The suggestions view of the rejuvenate pointcut tool.

Implementation

We implemented the algorithm presented in Section 4.2.5 as a plug-in to the popular Eclipse IDE. Eclipse abstract syntax trees (ASTs) with source symbol bindings were used as an intermediate program representation. The intention graph is constructed with the aid of a JayFX fact extractor, extended for Java 1.5 and AspectJ, which generates facts (using CHA) pertaining to structural properties and relationships, e.g., field accesses, method calls. Furthermore, we leveraged the AJDT compiler for the implementation of the *Enabled* relation described in Section 4.2.5, which associates PCEs with both vertices and labeled arcs depending on the captured shadows. The actual path matching described in Section 4.2.5 is implemented via the Drools rules engine.

To increase applicability to real-world applications, we relaxed several assumptions described in Section 4.2.5. For example, we conservatively assume that dynamic advice, i.e., advice bound to a pointcut containing run-time predicates, is always applied. If the tool encountered any inter-type declarations or any other form of the static crosscutting, the associated aspect was still processed but these constructs were ignored.

Figure 4.15 portrays a screen capture of the suggestions view from our research prototype implementation called REJUVENATE POINTCUT [KR08]. As shown, the tool is built as an extension to the AJDT and is coupled with other refactoring tools in Eclipse. The suggestions view contains 3 columns. Column *suggestion* refers to the join point shadow being suggested for the selected advice. The method signatures shown here symbolize the `!execution` join point shadow that corresponds to the method. Column *pattern* displays an informational description of the pattern in which the suggestion was produced from. Column *confi-*

dence, of which the list is sorted by in decreasing order, is the confidence value produced from equation (4.4) given the pattern listed in the previous column.

Our current tool research prototype is publicly available for download¹¹. In its current state, the tool presents the user with the new suggested join points for manual integration, however, in future versions of the tool, once the selection is final, the pointcut will be rewritten using existing refactoring support [BGK01] adapted for AspectJ constructs. A thorough evaluation of the prototype can be found in [KGRX08].

4.2.6 Conclusion and Future Work

Recording traceability information during the software engineering life cycle is, indeed, a valuable activity. Equally valuable is the ability to accurately correct invalidated traceability information as the underlying artifact base evolves. We have presented a preliminary design of GAMBLE, a general approach that is intended to mechanically alleviate the burden associated with maintaining trace information over the lifetime of a software product.

Although development of GAMBLE is currently underway, a specific approach and implementation has been developed, namely, REJUVENATE POINTCUT, an automated approach which limits the problems associated with pointcut fragility by assisting the developer in rejuvenating pointcuts as the base-code evolves. Future work involves further generalizing concepts used in REJUVENATE POINTCUT for use in GAMBLE, as well as a complete and thorough empirical evaluation of these concepts.

¹¹<http://tinyurl.com/6fdz55>

Conclusion

The AMPLE project aims at combining AOSD and MDD techniques to address variability at every stage in the Software Product Line (SPL) engineering life cycle. Although not directly apparent from this mission statement, management of traceability is core to the project and transverse to all its activities. Traceability overlaps the entire SPL process and it intersects and shares dependencies with other work-packages, in particular, those focusing on specific development steps during SPL development (WP1, WP2, and WP3). It is the responsibility of WP4 to provide the traceability platform that will be utilized by the other work packages to store tracing information and build up the trace model.

In these report, we described our implementation of a prototype version of the AMPLE Traceability Framework. This framework allows to create, retrieve, search, visualize and generally manage trace links between any artefacts. The framework was built on top of the Eclipse platform for generality, it offers base mechanisms for the management of trace links and extension points that allow advanced users to implement their own features.

The framework in its current state is only a prototype. A, revised, new version is planned in the light of initial experiments. New functionalities are also envisioned.

Bibliography

- [AGG⁺08] Nicolas Anquetil, Birgit Grammel, Ismenia Galvao, Joost Noppen, Safoora Shakil Khan, Hugo Arboleda, Awais Rashid, and Alessandro Garcia. Traceability for Model Driven, Software Product Line Engineering. In *ECMDA Traceability Workshop Proceedings*, pages 77–86, 2008. ISBN 978-82-14-04396-9.
- [Ber70] Claude Berge. *Graphes et hypergraphes*. Dunod, 1970.
- [BGK01] Dirk Bäumer, Erich Gamma, and Adam Kiezun. Integrating refactoring support into a Java development tool. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.
- [BGKV06] Mathieu Braem, Kris Gybels, Andy Kellens, and Wim Vanderperren. Automated pattern-based pointcut generation. In *Software Composition*, 2006.
- [CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.
- [DBWR07] Barthélémy Dagenais, Silvia Breu, Frédéric Weigand Warr, and Martin P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *Int. Conf. Automated Software Engineering*, 2007.
- [DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Eur. Conf. Object-Oriented Programming*, 1995.
- [DKTE04] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java programs to use generic libraries. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2004.
- [ecl08] Eclipse modeling tools. <http://www.eclipse.org/downloads/packages/>, 2008.
- [EGK⁺02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz — open source graph drawing tools. *Lecture Notes in Computer Science*, 2265:483–484, 2002.

- [FH92] Stephen Fickas and B. Robert Helm. Knowledge representation and reasoning in the design of composite systems. *IEEE Trans. Softw. Eng.*, 1992.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- [FTK⁺05] Robert Fuhrer, Frank Tip, Adam Kiežun, Julian Dolby, and Markus Keller. Efficiently refactoring Java applications to use generic libraries. In *Eur. Conf. Object-Oriented Programming*, 2005.
- [GKN⁺07] Ismenia Galvao, Safoora Shakil Khan, Joost Noppen, Jean-Claude Royer, and Andreas Rummeler et. al. Definition of a traceability framework (including the metamodel and the modelling of processes and artefacts to allow traceability in the presence of uncertainty) for spls. Technical report, The AMPLE Project, 2007.
- [HC01] J. Highsmith and A. Cockburn. Agile software development: the business of innovation. *Computer*, 2001.
- [KCH⁺90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis FODA Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University Pittsburgh, PA., 1990.
- [Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.
- [KETF07] Adam Kiežun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for parameterizing Java classes. In *International Conference on Software Engineering*, 2007.
- [KGRX08] Raffi Khatchadourian, Phil Greenwood, Awais Rashid, and Guoqing Xu. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. Technical Report COMP-001-2008, Lancaster University, Lancaster, UK, August 2008.
- [KLM⁺97] Gregor Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Eur. Conf. Object-Oriented Programming*, 1997.
- [KR08] Raffi Khatchadourian and Awais Rashid. Rejuvenate pointcut: A tool for pointcut expression recovery in evolving aspect-oriented software. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008.
- [KS04] C. Koppen and M. Stoerzer. PCDiff: Attacking the fragile pointcut problem. In *Eur. Int. Workshop on Aspects in Software*, 2004.

- [KSR07] Raffi Khatchadourian, Jason Sawin, and Atanas Rountev. Automated refactoring of legacy Java software to enumerated types. In *Int. Conf. Software Maintenance*, 2007.
- [ME08] R. Mitschke and M. Eichberg. Supporting the evolution of software product lines. In *ECMDA Traceability Workshop (ECMDA-TW) Proceedings*, pages 87–97, 2008.
- [MKD03] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *International Conference on Compiler Construction*, 2003.
- [RM02] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *International Conference on Software Engineering*, 2002.
- [Rob06] Martin P. Robillard. Tracking concerns in evolving source code: An empirical study. In *Int. Conf. Software Maintenance*, 2006.
- [TFDK04] Frank Tip, Robert Fuhrer, Julian Dolby, and Adam Kiezun. Refactoring techniques for migrating applications to generic Java container classes. Technical Report RC 23238, IBM T.J. Watson Research Center, February 2004.
- [vDD04] Daniel von Dincklage and Amer Diwan. Converting Java classes to use generics. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2004.
- [XR08] Guoqing Xu and Atanas Rountev. Ajana: a general framework for source-code-level interprocedural dataflow analysis of aspectj software. In *Int. Conf. Aspect-Oriented Software Development*, 2008.