André Luís Sequeira de Sousa

Traceability Support in Software Product Lines

Lisboa
2008

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Traceability Support in Software Product Lines

André Luís Sequeira de Sousa (aluno nº 26823)

Orientadora: Prof. Doutora Ana Maria Dinis Moreira
Co-orientadores: Prof. Doutor Vasco Amaral, Doutor Uirá Kulesza

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do grau de Mestre em Engenharia Informática.

Lisboa
2008

Acknowledgments

To my wife, parents and family for all their support on my educational path.
To all my teachers and colleagues that accompanied me during my academic path.
To all members of the FCT/UNL AMPLE research group with whom I had the pleasure of working, especially Uirá Kulesza, Mauricio Alferez, Antonielly Rodriguez, and João Santos.
To professors Ana Moreira and Vasco Amaral from Departamento de Informática (FCT/UNL) for the orientation provided during this stage of my master thesis. To professor João Araújo for also providing his guidance and council.
Also to my Comunidade Neocatecumenal, for all their support and prayers when they were needed the most.

Resumo

A rastreabilidade está a tornar-se uma qualidade indispensável de qualquer sistema de software moderno. A complexidade no desenvolvimento de software é de tal ordem que, se não contarmos com boas técnicas e ferramentas, torna-se rapidamente um fardo demasiado pesado, sendo dificil ligar os artefactos de software aos seus requisitos originais.

Os modernos sistemas de software são constituídos por um grande número de artefactos (modelos, código, etc.). Qualquer alteração introduzida num artefacto pode repercutir-se por vários componentes. Avaliar este impacto é uma tarefa árdua, dispendiosa e propensa a erros. Esta complexidade inerente ao desenvolvimento de software é aumentada no contexto de Linhas de Produtos de Software. A rastreabilidade pretende responder a este desafio, ligando os artefactos de software que são utilizados, de forma a descobrir as influências que eles exercem entre si.

A nossa proposta passa por especificar, desenhar e implementar um *Framework de Rastreabilidade* que forneça uma solução de rastreabilidade para linhas de produtos, ou a possibilidade de o estender para outros cenários de desenvolvimento. O trabalho desta dissertação de mestrado é desenvolver um framework extensível, utilizando tecnologias de Desenvolvimento Orientado a Modelos. Pretendemos ainda fornecer buscas básicas e avançadas, e vistas desenhadas para satisfazer as necessidades de cada utilizador.

Palavras-chave: Rastreabilidade, Engenharia de Linhas de Productos de Software, Desenvolvimento Orientado a Modelos.

Abstract

Traceability is becoming a necessary quality of any modern software system. The complexity in modern systems is such that, if we cannot rely on good techniques and tools it becomes an unsustainable burden, where software artifacts can hardly be linked to their initial requirements.

Modern software systems are composed by a many artifacts (models, code, etc.). Any change in one of them may have repercussions on many components. The assessment of this impact usually comes at a high cost and is highly error-prone. This complexity inherent to software development increases when it comes to Software Product Line Engineering. Traceability aims to respond to this challenge, by linking all the software artifacts that are used, in order to reason about how they influence each others.

We propose to specify, design and implement an extensible *Traceability Framework* that will allow developers to provide traceability for a product line, or the possibility to extend it for other development scenarios. This MSc thesis work is to develop an extensible framework, using Model-Driven techniques and technologies, to provide traceability support for product lines. We also wish to provide basic and advanced traceability queries, and traceability views designed for the needs of each user.

Keywords: Traceability, Software Product Line Engineering, Model-Driven Engineering.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. Introduction

In software development, traceability is becoming a necessary characteristic of any system, as it supports software management, evolution and validation [54]. The IEEE Standard Computer Dictionary [38] defines traceability as:

1. *The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match;*

2. *The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement that it satisfies*.

This definition is strongly influenced by the requirements management community, which were the originators of traceability [1]. Gotel and Finkelstein define requirements traceability as [32]:

...the ability to describe and follow the life of a requirement, in both a forward and backward direction; i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases.

In other words, traceability allows identification of the artifacts implementing a given requirement or the originating requirement of a given software artifact. For instance, a function programmed in C++ can be traced back through the elements that led to its implementation, all the way up to the requirement that motivates the existence of that particular piece of source code.

However, it is becoming clear that using traceability which is only concerned with requirements is not sufficient. In Aspect-Oriented Software Development (AOSD) [5],

traceability of crosscutting concerns could be used to guarantee the consistency of requirements [68]. In Model-Driven Engineering (MDE) [66] traceability links could be used to associate the target models created by model transformations to their source models and vice-versa. Some authors are beginning to regard traceability in a much broader scope, where traceability is seen as any relationship that exists between artifacts involved in the software life cycle [1].

With software systems growing in complexity and size, traceability has become a necessity for software developers. Proper traceability support can yield improvements in software quality and eliminate much of the overhead attached to tasks such as performing covering analysis, change impact analysis and other related tasks. Another benefit is that using automated and formally defined methods of capturing trace information lowers the chance of occurring errors and the costs (economic, time, organizational, etc.) inherent to performing this tasks by hand [32].

This thesis work is being developed in the context of Software Product Lines (SPL) Engineering, more specifically, for the support of traceability in SPL, filling the gap in this area. We will specify, design and implement a traceability framework for this software development methodology. For that purpose, several technologies were assessed, with a special focus on Model-Driven Engineering (MDE) for suitability in achieving our goals.

1.1 Problem Description

In modern software industry, the complexity inherent to the development of a software system can become an unsustainable burden if not properly dealt with. Understanding the relationships between the different artifacts used in software development plays a major role in ensuring that the delivered system meets the stakeholders' needs [57]. It is within this context that traceability problems can occur when the artifacts found in a solution (the system implemented) cannot be easily matched against the corresponding set of problem features that are being addressed [34].

Without proper traceability methods, any change introduced, either in one of the requirements (e.g., new market rules), intermediate specification model, or in any of the solution artifact (e.g., source code file is edited) might lead to the inability to verify if the requirements are still being satisfied (forward traceability), or if the artifact is still implementing any requirement (backward traceability).

The traceability problem requires a solution that allows the developers to reason about the relationships between the elements of the different spaces (problem space vs. solution space), in order to efficiently maintain and evolve a software system.

This problem becomes even more complex in a SPL environment, where the existing complexity to traceability (a great deal of artifacts to be dealt with) is augmented by the increasing number of different variants that can be produced by the product line. This adds the necessity to provide not only traceability between the many artifacts that compose a system, but also to distinguish between the artifacts that belong to a variant that is created by simply choosing a configuration features.

Many approaches for SPL development have been proposed over the years [9, 11, 19, 20, 26, 31, 59, 61]. The vast majority of these approaches offer the possibility to specify the commonality and variability inherent to a product line and some of them even offer complete solutions for establishing an SPL [11, 59, 61]. The most common method for achieving this is through the use of feature models, initially presented by Kang et al. [44], which have been extended and adapted to include several

improvements [14, 21, 35]. Other proposals have also emerged, such as the Orthogonal Variability Model [59] or modeling features with UML [31]. These approaches are usually focused on combining variability information contained in a variation model (e.g., feature model) with requirements information contained in an appropriate model (e.g., use cases) which is an important aspect of product line engineering.

The main problem is that most of them do not provide any traceability support other than combining elements from different domains. For instance, validating a feature implementation (e.g., a design class) against its requirements is a time consuming and error-prone task without a solution that provides traceability support throughout the entire SPL life cycle. Without traceability techniques, performing change impact analysis comes at a huge cost. Reasoning about the rationale associated with an artifact, information that is necessary for performing tradeoffs, is also very difficult to accomplish without proper traceability support. Finally, coverage analysis can also be performed quite easily with traceability mechanisms. Checking if all the requirements are being met or if an artifact is traceable to any requirement becomes a question of simply submitting the right query and browsing the results.

The majority of the existing approaches does not address these issues or do so in a trivial way. Another important aspect is the appropriate tool support for an effective traceability solution. Most approaches do not provide a complete solution or do so in a strictly theoretical fashion without supporting tools, which must be an essential part of any traceability framework.

1.2 Limitations of Existing Approaches

Several approaches and tools have been developed over the years to address the traceability problem in software development. Almost all of the currently available tools [30, 37, 39, 65, 69] have been implemented for traceability in Single-Systems development. They are capable of managing the trace links between requirements and other software artifacts, however they do not address some key issues in product lines such as managing variability and linking it to the artifacts used throughout the SPL lifecycle. Recently, there has been as effort to integrate some of these traceability tools with SPL tools such as *pure::variants* [61] and GEARS [11], but although this enables to trace from feature models to other artifacts, there are still many useful and important trace queries that are left unanswered. For instance, trace links could be exploited to perform feature interaction detection or to discover how the artifacts of a product variant relate to another product variant. This is currently not possible to do using the tools currently available. Another problem is that commercial traceability tools are closed and cannot be adapted to extend their base capabilities, which makes it impossible to take an existing tool and adapt it to the SPL traceability scenario.

Some approaches to address traceability in SPL have been proposed by several authors. Some authors prefer to model variability using a model created for that purpose, and to create traces from variability elements to other artifacts [59]. Other authors focus on traceability for certain aspects of SPL, keeping it comprehensive but not exhaustive [7, 47]. The problem with the majority of these approaches is the lack of appropriate tool support which limits its use in a real software development scenario.

1.3 Work Context

AMPLE [60] is an RTD project funded by the European Union. AMPLE combines renowned academic and industrial expertise from UK, Portugal, France, Spain, Netherlands and Germany to provide a holistic software product line development methodology that improves modularization and traceability of variability. The list of participants includes:

- Lancaster University, UK
- Universidade Nova de Lisboa, Portugal
- Darmstadt University of Technology, Germany
- ARMINES and Ecole des Mines de Nantes, France
- University of Twente, The Netherlands
- Universidad de Malaga, Spain
- HOLOS, Portugal
- SAP AG, Germany
- Siemens AG, Germany

The project coordinator is Prof. Dr. Awais Rashid of Lancaster University. The coordinator for the FCT/UNL team is Prof. Ana Moreira. The project began in October 2006 and has 3 years of duration, until September 2009.

The aim of AMPLE is to provide a Software Product Line development methodology that offers improved modularization of variations, their holistic treatment across the software lifecycle and maintenance of their (forward and backward) traceability during SPL evolution. Currently, there is a big gap between research in requirements analysis, architectural modeling and implementation technology, and the industrial practice in SPL engineering. Furthermore, the focus tends to be on the design and code level when variations need to be identified, managed and analyzed from the very early stage of requirements engineering. Architecture models are related to requirements models in an ad-hoc fashion and implementation tends to rely on pre-processors which are inadequate substitute for proper programming language support for variability. Nor is there any systematic traceability framework for relating variations across a SPL engineering lifecycle.

AMPLE will combine AOSD and MDE techniques to not only address variability at each stage in the product line engineering lifecycle but also manage variations in associated artifacts such as requirements documents. Furthermore, it aims to bind the variation points in various development stages and dimensions into a coherent variability framework across the lifecycle thus providing effective forward and backward traceability of variations and their impact. This makes it possible to develop resilient yet adaptable SPL architectures for exploitation in industrial SPL engineering processes.

Figure 1.1 shows the work package breakdown of AMPLE. This thesis work is being developed under the WP4 umbrella of the AMPLE Project. The WP4 is concerned with developing a framework for backward and forward traceability in product line engineering.

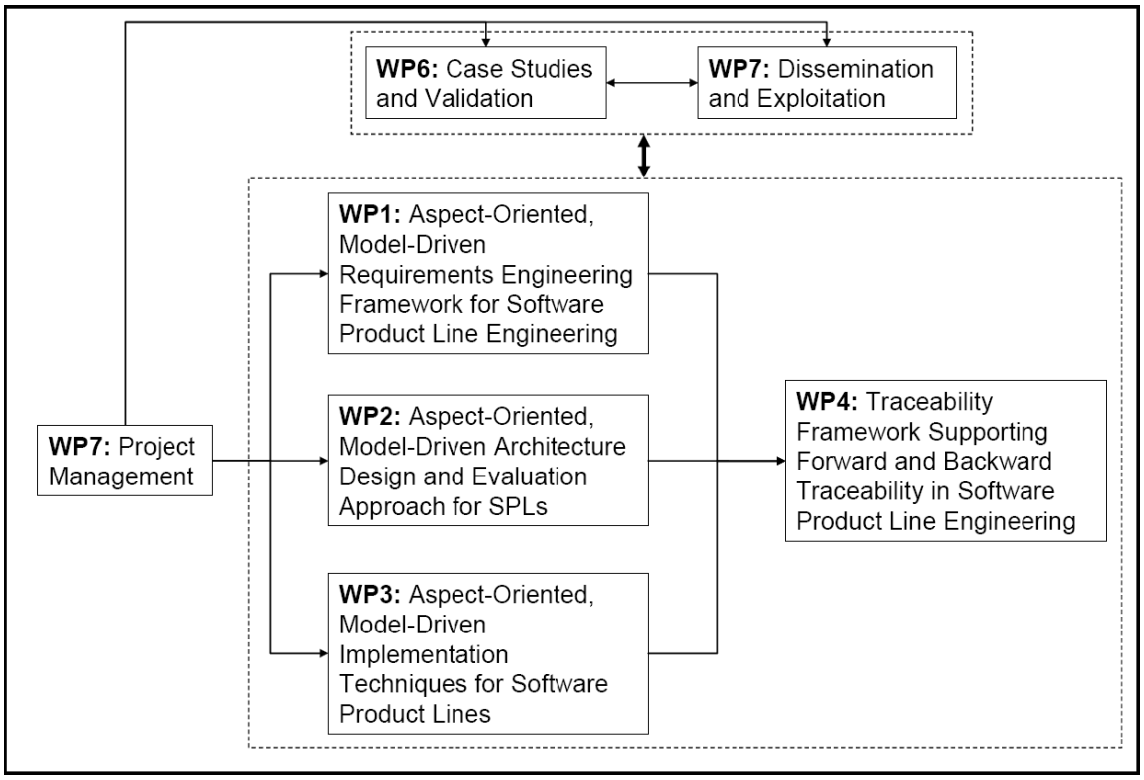


Figure 1.1 – Graphical representation of work packages in the AMPLE project

1.4 Proposed Solution

The solution that we propose is to specify an extensible Model-Driven Traceability Framework that allows for the definition and implementation of traceability mechanisms and tools for SPL. In a first stage, only traceability between features and requirements artifacts will be provided, but implementing a solution for other artifacts should be straight forward. This framework will be composed of four main modules, as shown in Figure 1.2.

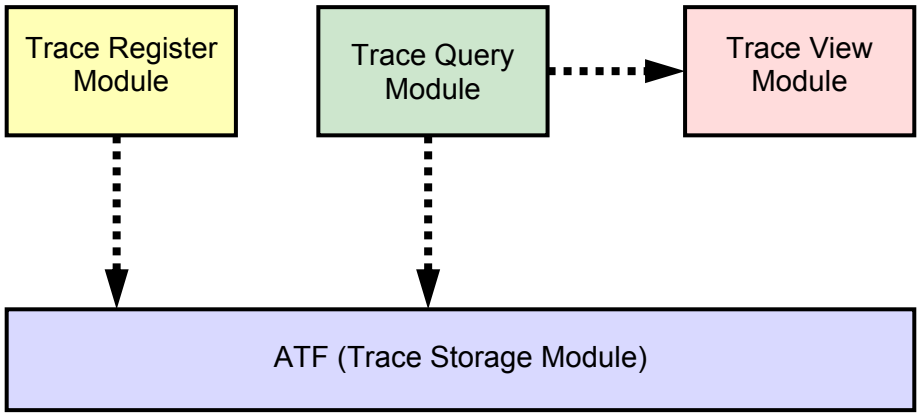


Figure 1.2 – Traceability Framework architecture overview

The "Trace Register Module" is responsible for creating, updating or deleting trace links between the different artifacts. It can be used to manage the links stored in the repository. The "Trace Query Module" provides means to select a subset of trace links,

from the traceability repository. It is used for submitting basic queries, like selecting the artifacts linked to a feature, or more complex queries, like coverage analysis and change impact analysis. The "Trace View Module" is responsible for supplying a graphical or textual view for the trace queries submitted. Finally, the "ATF (Trace Storage Module)" module is in charge of storing the trace links, the trace artifacts and associated information, such as the rationale, in a persistent way.

The main idea behind this framework is to use the variability model as the base reference for traceability. It should allow the developers to perform traceability from the variability model to other models. Another important aspect is that the framework should be highly adaptable and evolvable, to satisfy the needs of different users. Initially, we will be using the feature models (modelled in FMP [4]) as our variability model and a use cases model as the requirements model, but these will be variation points for this framework. The framework will allow changing either the variability model or the requirements model or any other model used in a SPL, in order to suit each developer's needs. For instance if one wishes to use viewpoints, it can be used instead of the default use cases model with a simple adaptation to the framework. This goal is achieved by using a traceability metamodel under development for the AMPLE's WP4, which will be explained in more detail in Chapter 3.

1.5 Document Structure

The remainder of this document is divided in six chapters and four annexes. Chapter 1 is this introduction. Several concepts regarding the traceability problem were presented with a special focus on traceability for Software Product Lines.

Chapter 2 presents some related work. First the general concepts regarding traceability are discussed. We introduce some notions and methods on how to describe and use traceability to aid software development. The chapter also presents some of the existing traceability solutions for MDE and SPL, and the state-of-the-art in terms of research in this field. A survey of some traceability tools that were evaluated in the context of this thesis is also discussed.

Chapter 3 presents our proposal for addressing the main problems discussed in Chapter 1 and Chapter 2. We are proposing a Model-Driven framework to address traceability in software development, especially in SPL development, an area where we found a big gap. In this chapter we describe the general architecture of the framework, the implementation that was achieved, and the necessary steps to instantiate the framework's hotspots. We also discuss our plans of evolving the framework to a scenario of "Black Box" instances development, to ease the process of instantiating the framework hotspots.

In Chapter 4 we discuss how we plan to use traceability techniques to address problems that arise during the SPL development stages. We describe three main problems that affect SPL development: covering analysis, change impact analysis and detection of feature interaction. We present the strategies that we have developed for addressing these problems and our plans to implement them as instances for our framework.

Chapter 5 presents a case study based on a home automation product line. This case study is based on "Smart Home" case study provided by Siemens[1]. We validated our framework against this case study by demonstrating how we could implement an

[1] Case study provided in the context of AMPLE project.

instance to provide detection of feature interactions. This kind of problem is not addressed in other traceability tools and remains untackled in the approaches discussed in Chapter 2.

Chapter 6 concludes this document by presenting the contributions and our plans for future work.

Chapter 2. Related Work

In this chapter we will discuss the work related to traceability and how it as been applied in the fields of Model-Driven Engineering (MDE) and Software Product Lines (SPL). We begin by introducing some definitions for traceability. Some usage scenarios, techniques on how to achieve traceability solutions and trace queries are also described in the first section. We also discuss the importance that traceability has in software development, has it provides means to address the growing complexity of software systems [34]. The following sections of the chapter discuss traceability solutions in the domains of MDE and SPL. We present a brief introduction to these subjects, and analyze several approaches that have been proposed for traceability in these areas of research. Each approach is evaluated by a set of criterion and the results are discussed at the end of each section. The final section presents the results of the traceability tools survey that was conducted as part of this thesis work. We also evaluated each tool against a set of criterion and present our results in the end of the chapter.

2.1 Traceability

With the increase of the software complexity, software engineers have realized that the assessment of the impact that a change in one requirement introduces in the rest of the system is a critical task [34]. When a requirement changes, it may introduce a conflict with the rest of the requirements of a system. For instance, if the system is required to evolve to provide enhanced security (e.g., using stronger encryption algorithms and larger encryption keys) it may conflict with the existing requirement that addresses the response time that the system should meet.

If these conflicts go undetected, it may lead to a system that fails to meet the requirements of the stakeholders, meaning that the software system might not be good enough for its intended use [63]. It therefore becomes necessary to perform the task of detecting how requirements influence each other, and how they are implemented, so that

when a change is introduced in either a requirement, or in a software artifact, it is possible to assess the impact that reverberates on the rest of the software system. In his book "*Software Requirements: Objects, Functions and States*", Davis classifies traceability information in four types [22]:

- **Backward-from traceability** links requirements to their sources in other documents or people.
- **Forward-from traceability** links requirements to the design and implementation components.
- **Backward-to traceability** links design and implementation components back to requirements.
- **Forward-to traceability** links other documents (which may have preceded the requirements document) to other relevant requirements.

An important aspect is also the ability to define traceability between requirements themselves. Davis does not seem to clearly define this relationship, but Kotonya and Sommerville [50] state that by extending the backward-from and forward-to traceability in order to allow links between the same document (the requirements document), it is possible to cover this concern.

Even though traceability as been defined to include links to the source of a requirement (backward-from traceability), in practice, it is usually maintained between requirements themselves and between a requirements and its design artifacts. Gotel et al. [32] give a comprehensive picture of the "requirements traceability problem" and discuss the necessity to perform traceability all the way up to the source of a requirement (people, other requirements, documents, standards, etc.). They introduce the notion of pre-RS (pre-Requirements Specification) traceability and post-RS (post-Requirements Specification) traceability. This distinction is necessary because there are different needs in terms of the information that is dealt with in each level and the problems that arise.

Post-RS traceability provides the ability to trace requirements from, and back to, a baseline (the requirements specification). Any change introduced in the requirements specification must be propagated though the linked elements. The pre-RS traceability consists on the ability to trace requirements from, and back to, their originating statement(s), from which requirements are produced with the information collected from the existing sources, and combined in a single requirements specification. Changes in this requirements production process must be reflected in the requirements specification, and vice-versa.

To define traceability links, one of the most common methods used is by means of traceability tables. Figure 2.1 shows a traceability table example for a system with six requirements. Each requirement is listed in the vertical and horizontal axes of the table, and the cells are used to mark the relationships between them. These tables can also be used to mark any other kind of traceability relationship. This would require adding the traceable artifact to both axes.

The proper way to read this table is by navigating its cells and interpreting the information depending on whether we are reading a row or a column. A mark in the row of a requirement indicates that this requirement is depending on the marked requirements. For instance, R3 in Figure 2.1 is dependent on R4 and R5. On the other hand, by navigating the column we can find which requirements are depending on a

specific requirement. We can also see that requirement R2 has R4 as dependent. Using this approach, it is possible to perform change impact analysis simply by navigating the column of the requirement that was changed. If a change was introduced to R5, by going down the R5 column, we could find that R2 and R3 are dependent requirements, and therefore the impact on R2 and R3 of the change introduced on R5 can be assessed.

	R1	R2	R3	R4	R5	R6
R1			*	*		
R2					*	*
R3				*	*	
R4		*				
R5						*
R6						

Figure 2.1 – A simple traceability table (taken from [50])

These tables however, are not scalable. For a small number of requirements, it is possible to navigate the table with relative ease to perform the necessary analysis, but for a large number of requirements (hundreds or thousands) the table becomes unmanageable. To address this problem, an alternative to traceability tables can be used. Instead of using a table, it is possible to use two traceability lists to provide the same information. One list traces the "depends-on" traceability, while the other provides the "dependents-off" traceability. Figure 2.2 shows an example of these lists.

	Depends-on
R1	R3, R4
R2	R5, R6
R3	R4, R5
R4	R2
R5	R6

	Dependents-off
R2	R4
R3	R1
R4	R1, R3
R5	R2, R3
R6	R2, R5

Figure 2.2 – "Depends-on" and "Dependents-off" traceability lists (adapted from [50])

Traceability lists have the advantage of being more compact and easy to read than traceability tables. If one wishes to find the requirements that are depending on a given requirement it is a matter of searching the "dependents-off" list. For instance it is easy to see that R3 as R1 as dependent. The inverse is also straightforward. By navigating the "depends-on" list, we can see that R5 is dependent on R6. The only drawback of using traceability lists is that the information is duplicated, leading to problems of maintaining the information in a coherent state in both lists.

Finally another important aspect of traceability is the ability to perform queries on traceability information and viewing the results returned. The great majority of the authors do not provide any implementation of the approaches proposed. This is usually left for the developer of the system to either implement the traceability himself, or to be

performed by traceability tools [13, 30, 37, 39, 69]. Many different solutions can be used to achieve the same goal. Traceability tables can be implemented using a simple spreadsheet, or a relational database. Traceability lists can also be implemented using a relational database, a spreadsheet, a simple text file, or implemented in many programming languages (e.g., the standard Java Class Library [67] already contains a Hashtable implementation that could be used to implement this solution).

Depending on the solution, the desired information can either be extracted manually (e.g., going though a spreadsheet) or automatically using predefined or custom queries (e.g., using SQL with a relational database). There are several kinds of traceability queries that can be performed. Traceability queries can occur at two different levels:

- **Inter-level** Queries traceability between artifacts at different development levels (e.g., requirement with design artifact or design artifact with source-code).
- **Intra-level** Queries traceability between artifacts at the same level of development (e.g., requirements models with requirements documents).

The type of queries can also be divided between simple and complex types. Simple queries usually involve choosing an artifact or set of artifacts and viewing the related artifacts. The type of information returned should also be chosen (which kind of artifacts, etc.). The results can also be viewed in a variety of formats. It is possible to see it represented in a simple textual report, a graph or a table. Figure 2.3 shows an overview of the concepts that address traceability.

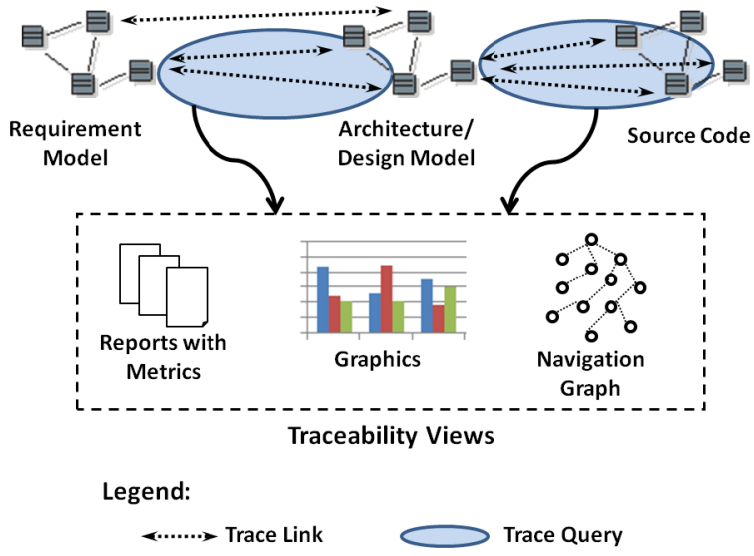


Figure 2.3 – Traceability basic concepts (taken from [71])

More complex and sophisticated traceability queries can also be provided to the developers. These might include requirements coverage analysis, which aims to ensure that architecture, design and implementation artifacts cover specific requirements. This could be useful in satisfying the need of showing that the resulting system met the contractual agreements. Another important query might be the change impact analysis, mentioned previously, which allows the discovery of artifacts that are affected by a change introduced in a requirement. As mentioned in Chapter 1 the most common form

of traceability is performed between requirements themselves and/or the software artifacts that realize them. However, Aizenbud-Reshef et al. suggest a broader definition of traceability [1]. The authors consider traceability as "any relationship that exists between artifacts involved in the software-engineering life cycle". This definition includes elements that fall out of the scope of requirements traceability but are essential for new paradigms and methodologies that have emerged in recent years (e.g., Model-Driven Engineering [66] or Software Product Lines [19]), which will be discussed in more detail in the following sections.

2.2 Model-Driven Engineering

Model-Driven Engineering (MDE) refers to the systematic use of models as first-class entities throughout the software development process, where the software lifecycle is considered to be a chain of model transformations [28]. The main purpose of MDE is to provide the developers with methodologies that use models, raising the level of abstraction of creating software [66]. The goal of MDE is to make the process of creating new software automatic and to ease necessary changes in a rapidly changing environment by using model transformations. Model-Driven Engineering is sometimes referred to as Model-Driven Software Development or Model-Driven Development [66][2].

According to Stahl and Völter [66], the idea of modeling, as in MDE's point of view, is not exactly new, and has been used in software development for documenting the inner structure of software. Developers would then review each step of the development process to check the models for consistency and correct possible mistakes. Another approach is reverse engineering that is possible in many UML tools. However this approach is merely source code visualization in UML syntax. Visually it may be clearer and more understandable, but in essence, the abstraction level of these models is the same as the source code itself.

Model-Driven Engineering offers a significantly more effective approach: models are abstract and rigorous at the same time. Abstractness does not stand for vagueness, but for compactness and reduction to the essence [66]. The difference between old modeling techniques and modern MDE is that the new vision is not to use models only as simple documentation to aid in software development, but use them as input/output for computer based tools implementing precise operations. MDE models have the exact meaning of program code in the sense that the bulk of the final implementation, not just the class and method skeletons, can be generated from them. Models are no longer only documentation, but they become actual parts of the software.

Figure 2.4 shows the MDE process of model refinement and the relationship of each model with the developed system. The vertical arrows demonstrate the refinement achieved in each step, by means of a model transformation, from more abstract models, into more concrete ones. Since all the models are representations of the same system, each transformation step should preserve the intended meaning of the source model, while adding new details to the resulting model [28].

Bézivin compares the evolution of object technology in the past period with the new proposals and claims of MDE [10]. The basic principle in Object-Oriented technology was: "Everything is an object". This had a great impact in driving the technology in the direction of simplicity, generality and better integration. With the

[2] The acronym MDE will be used in this dissertation from now on.

appearance of MDE we seem to have entered in a shifting phase in the software development process, and "Everything is a model" is promising to replace the OO guiding principle [10]. What seems to be important now is capturing a particular view (or aspect) of a system in a model and that each model is written in the language of its metamodel.

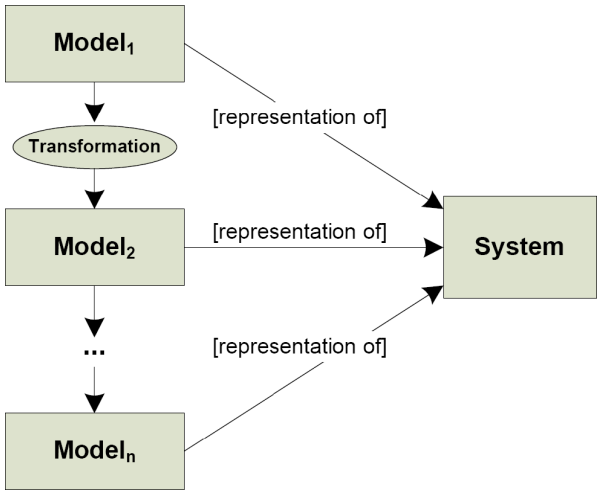


Figure 2.4 – Model refinement in MDE (taken from [28])

2.2.1 Traceability in Model-Driven Engineering

When put in practice, the MDE process follows some general principles of software development, like iterative development, separation of concerns, reverse engineering and refactoring [28]. Since the system is developed as a series of model transformations until a system implementation is achieved, any change introduced in a model should be propagated throughout the rest of the models. To maintain a complete integrity in the system's models, the changes should be propagated to models that were derived from the changed model but also to the models that originated the model that was changed.

To effectively perform the necessary changes, one must solve the problem of knowing which models are related to the changed model. Traceability in MDE can be used to address this issue, providing a solution to this potentially complex problem. By using trace links that associate elements of different models, and due to transitivity inherent to model transformations, it is possible to completely identify all the elements that are affected by a change in an element. For instance, if a design class is changed, it is possible to trace that change all the way down to the C++ class that implements it. In the same way, it may be possible to trace all the way up to the requirement that derived a particular set of classes. It is then possible to analyze the impact of this change in those elements. A great deal of research as been developed and many approaches have been proposed to solve the problem of representing, capturing and querying the trace information during MDE development. They will be discussed next.

Event-Based Traceability (EBT) is a method for automating the generation and maintenance of trace links [16]. With this method, the requirements and other artifacts are not connected directly as in other approaches. A publish-subscribe mechanism based on the Observer design pattern [29] is used to perform traceability. Instead of using direct links, these are established by an event service that uses information retrieval

techniques to extract links between the registered artifacts. This system is composed of three main components: the event server, the requirements manager and the subscriber manager. The requirements manager handles the requirements and is capable of publishing the changes that are performed in a requirement as events in the event server. The event server is used to manage the links between a requirement and its dependent artifacts. It is also responsible for receiving change notifications and sending messages to the subscriber manager of the affected artifacts. Each artifact has a subscriber manager, responsible for handling the messages received by the artifact and for executing the necessary steps to properly handle the message received. The subscriber manager is also responsible for registering the artifact in the event server. The main drawback of this approach is concerning scalability. As a project grows, the event server can become a bottleneck for the system and it becomes hard to maintain a good performance.

Event-Based Traceability with Design Patterns (EBT_{DP}) is another approach by Cleland-Huang and Schmelzer [17]. It builds on top of EBT, but defines a different process for dynamically tracing non-functional requirements (NFR) to design patterns. This approach consists of two distinct phases, depicted in Figure 2.5. In the first phase, which occurs during construction of the system, the initial user-defined traceability links are established. In the second phase, which occurs during the ongoing maintenance and refinement of the system, fine-grained links are dynamically generated. As NFRs are elicited during early software development stages, the relationships between them and design patterns are discovered. The elements that compose a design pattern (models and code) are grouped in a cluster and a trace link is established between that cluster and the related NFR. This decreases the number of links established between design artifacts and non-functional requirements.

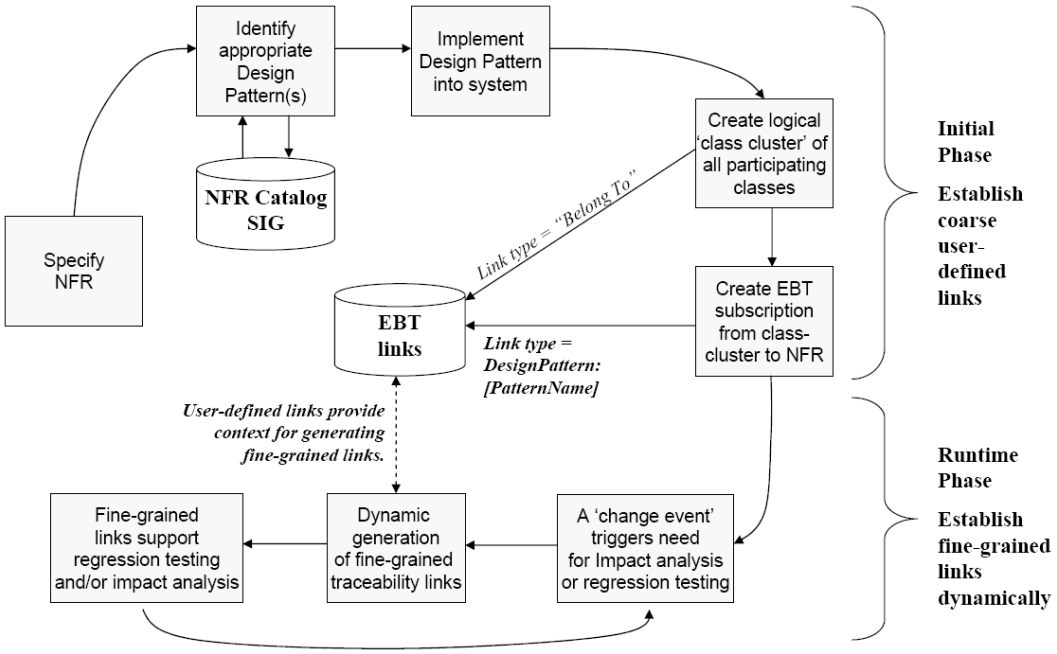


Figure 2.5 – The EBT_{DP} process (taken from [17])

In the runtime phase, the well defined descriptions of a design pattern allow the automatic and dynamic generation of code during runtime, with the pleasant side-effect

of enabling automatic generation of fine-grained links. This characteristic increases the maintainability and the expressiveness of the method. The current techniques for detection of implemented design patterns are not precise enough to support a good level of traceability, but this approach promises to improve the precision and achieve a good level of dynamical link generation.

Goal Centric Traceability is a proposal to effectively maintain non-functional requirements thorough the software life cycle made by Cleland-Huang[18]. The nonfunctional requirements and their dependencies are modeled using a Softgoal Interdependency Graph (SIG). The purpose of GCT is to allow the developers to assess how a change in a functional requirement affects the non-functional requirements of a system. According to the authors, it is possible to identify potentially impacted goals, to analyze the level of impact and to develop the desired strategy to minimize risks.

Figure 2.6 shows the phases of GCT: (i) Goal modeling; (ii) Impact detection; (iii) Goal analysis; (iv) Decision making. Goal modeling occurs during the elicitation specification. In this phase, non-functional requirements are modeled as softgoals in a SIG. During impact detection the traceability links between functional and non-functional requirements are created. A trace retrieval algorithm is used to return a set of potentially impacted goals, which are then evaluated by the user and any link that is incorrect is discarded. In the goal analysis phase the impact of a change is propagated though the related regions of the SIG in order to expose its effect in the system wide goals. Finally, during the decision making phase the stakeholders analyze the impact introduced by the change and evaluate if it should be implemented.

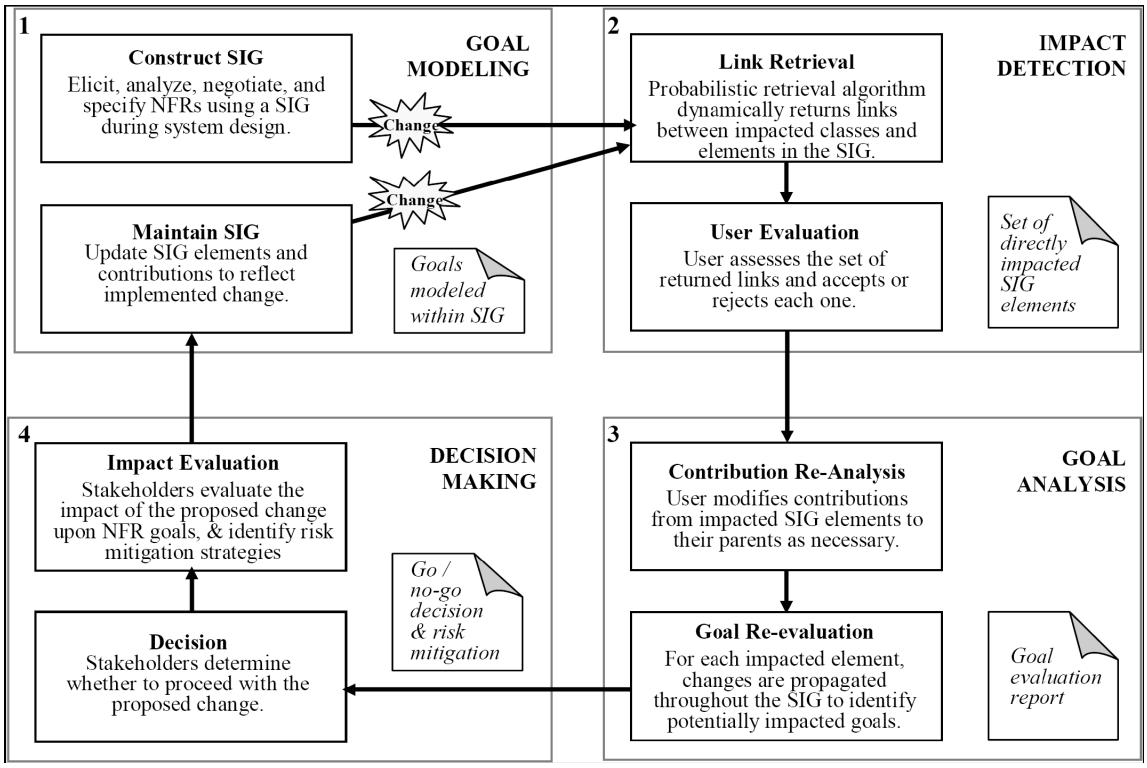


Figure 2.6 – Goal-Centric Traceability (taken from [18])

Ramesh and Jarke studied a wide range of traceability practices, and submitted them to scrutiny by using an empirical approach and focus interviews conducted in a series of

software organizations [62]. The result of this work was the creation of reference models that include the most important kinds of traceability links for the various software development elements, which reflect the actual needs of real users. The importance of this study is that it realized that different stakeholders have different traceability needs, and therefore should be presented with different reference models that respond to their specific needs. They also realized that traceability participants fell into two distinct categories, which are referred as high-end and low-end users.

Low-end users see traceability as an obligation imposed by project sponsors, and use simple traceability schemes as a means of modeling dependencies between requirements, components and compliance verification procedures. High-end users consider traceability to be an indispensable component of the software engineering process and will usually employ richer traceability schemes. The authors then proposed a basic traceability metamodel, shown in Figure 2.7, that expresses the trace entities used by high-end and low-end users and customize a set of reference models, contained within the scope of the trace metamodel, for each group. The reference model aimed at low-end users provides only a handful of relationship types that satisfies the needs of this group. The reference model for high-end users provides a much broader set of link types that allow the users to establish a much better rationale about the dependencies between the different elements.

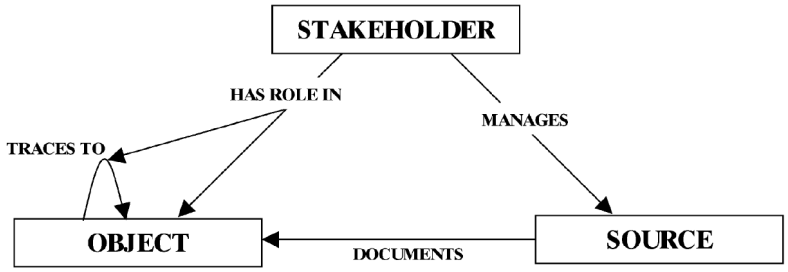


Figure 2.7 – Essential traceability metamodel (taken from [62])

Egyed presents an approach to traceability analysis [24]. This approach consists of establishing trace dependencies by analyzing some of the elements that constitute a software system: test scenarios, model elements (use cases, class diagrams, etc.) and source code. The approach also requires an observable and executable software system, which is its major drawback since it cannot be applied on the early phases of the software lifecycle and only when an implementation of the system is available. Other elements required by the approach are: a list of development artifacts; scenarios describing test cases or usage scenarios for the development artifacts; and a set of initial hypothesized traces linking artifacts and scenarios. Figure 2.8 shows the steps required by this approach.

The test scenarios are executed to observe the behavior of the system. By observing the execution of those scenarios it is possible to detect trace links between the scenarios and the source code that executes them. The user is also required to input a set of hypothesized traces between model elements and the test scenarios. With this information, it is possible to automatically perform trace analysis to extrapolate traces between model elements and scenarios, between model elements and source code, traces between model elements and also detecting inconsistencies and incompleteness. This approach reduces the complexity of generating and validating trace information, since it is only required to input the list of observed traces by running test scenarios and a set of hypothesized traces between those scenarios and model elements. The author

considers a footprint to be the source code that implements a model element. With this definition in sight, a footprint graph that represents the runtime behavior of the scenarios is created, and the algorithm proposed by the author generates trace links by analyzing how this graph relates to the hypothesized traces input and the elements to which they are linked.

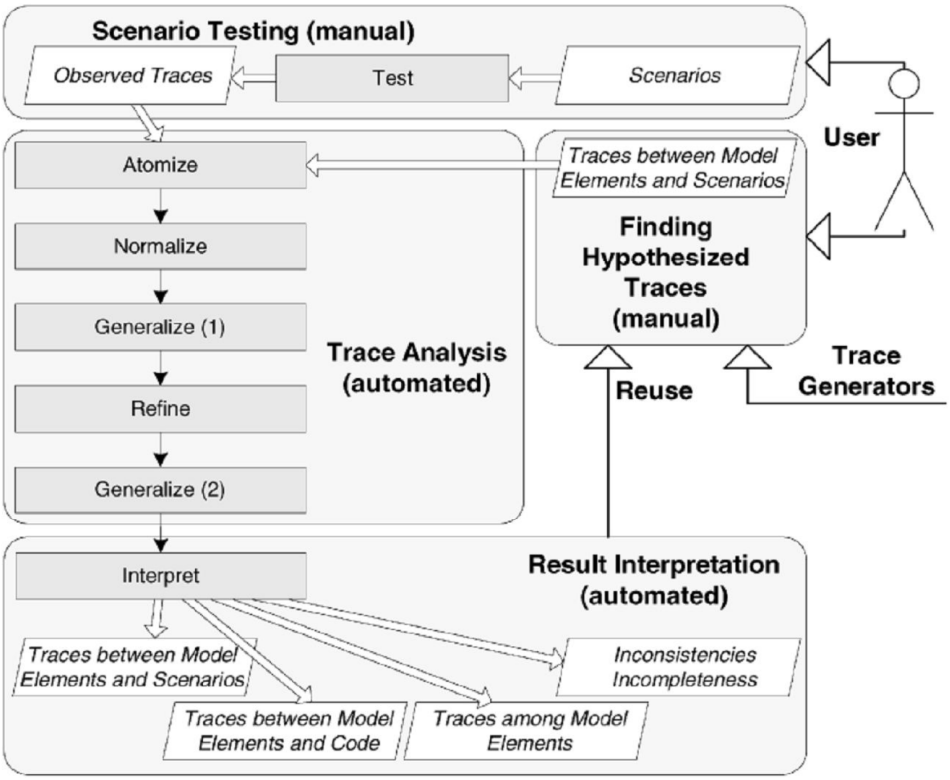


Figure 2.8 – Trace Analyzer overview (taken from [24])

Jouault shows a method for adding traceability to programs [42], written in the ATLAS Transformation Language (ATL) [43]. ATL is a model transformation language that has built-in support for traceability used in model transformations. This form of traceability however, is not maintained once a transformation is completed. Jouault also argues that a single transformation program can be used in several different contexts and as such, it may be required to generate different kinds of traceability information, depending on the execution context. The approach proposed is to consider the traceability information generated as an additional target model for the transformation program. Since the trace information is considered to be a model, the author introduced a simple traceability metamodel, shown in Figure 2.9, to allow the creation of trace links during model transformation.

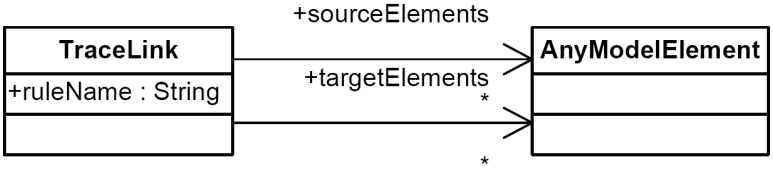


Figure 2.9 – Trace metamodel (taken from [42])

This approach uses the mechanisms already available in ATL to create traceability elements in the same manner that other target model elements are created. To integrate traceability in transformation programs, the developer simply needs to add a pattern element that will generate an external trace link in the traceability model. The drawback is that this ATL code must be manually added to the transformation program. But since transformation programs are models themselves, than an ATL program without traceability can be transformed in another ATL program that already includes this traceability code. The result was the creation of an ATL program called TraceAdder [42] that automatically inserts the traceability creation code in an ATL program.

Kolovos et al. presented an approach generating annotated models on-demand by merging the primary models with their correspondent trace models [49]. These annotated models contain traceability information that is useful for inspection purposes. The authors define two approaches for storing and managing traceability links. The first approach, called embedded traceability, the traceability elements are kept inside the target models they are referring. This approach simplifies the definition of traceability and helps in its understanding, but it pollutes the target model with many elements that do not belong there. Another disadvantage is that this approach can only be applied to represent intra-model traceability links. The second approach, called external traceability, creates trace links as elements in a separate model. For this approach to work, all the related elements must have a unique and persistent identifier that eliminates ambiguity problems when resolving external links. The advantage of this approach is that storing the traceability in links in separate models facilitates loose coupling between the models and the links. The down side is that id-based links are not human-friendly which goes against one of the goals of traceability, i.e., to assist modelers in decision making.

The authors argue that even though both approaches have disadvantages the external traceability model is more flexible and it allows managing intra-model and inter-model traceability, and present a technique of external traceability that overcomes the problem of user-friendliness [49]. This goal is achieved by automatic merging of traceability links (stored in separate models) with the elements to which they refer. This produces models annotated with traceability information. The Epsilon Merging Language (EML) [48] is used to perform the merging of models with traceability. This task is performed in two phases: matching and merging. The first phase establishes the correspondence between the source models (e.g., the traceability model and design model). The identified elements are then merged in the merging phase, resulting in a model annotated with traceability.

Falleri et al. defined a traceability framework that is especially suited for gathering traceability when chains of transformations are applied [27]. Their work is inspired by [42] and is implemented in the model oriented language Kermeta [70]. The authors argue that to trace model transformations, the two main concepts involved must be clearly defined: what is a model and what is a transformation. The first answer as been answered by the proposal of several metamodels that represent what is a model (e.g., MOF [56]). When it comes to model transformations a consensus has not been found yet, primarily due to the fact that a transformation language and the respective transformation metamodel are not independent. In order to provide traceability for any kind of model transformations, one must discard a concrete transformation model [27]. The authors then provided some definitions for what they consider to be a model and a

transformation. Based on those definitions they concluded that a model transformations trace is simply a bipartite graph with two kinds of nodes: source nodes and target nodes.

Figure 2.10 shows the specification for the basic elements of a transformation chain trace metamodel. In this metamodel, a transformation chain trace is represented by a trace. Every trace is an ordered set of steps, each one representing a single transformation (from a source model element to a target model element). The authors have implemented their framework using Kermeta that provides the following set of features: Generic traceability items; trace serialization; and trace visualization (achieved using Graphviz's Dot language [33]). The framework does have a major disadvantage. The trace generation code must be placed inside the transformation code, and as such, it becomes tangled with the transformation code itself, leading to an approach that is less extendable and adaptable.

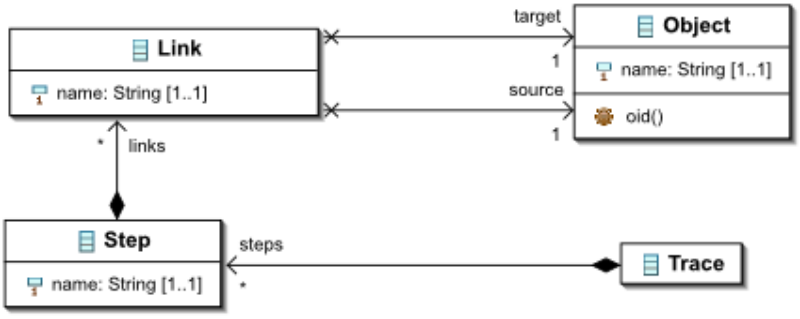


Figure 2.10 – Transformation chain trace metamodel (taken from [27])

2.2.2 Discussion

A survey in traceability approaches for MDE was conducted in the context of the AMPLE project [46]. Galvão and Goknil have also conducted a survey on traceability approaches for MDE and present their findings in [28]. Some of the approaches evaluated by these surveys were presented in the previous section. From all the approaches described in the surveys, we chose to discuss only the ones that, in our opinion, seem to provide a more complete solution (with regard to the evaluation criterion). The criteria used by authors for evaluating the performance of each approach were: *representation of traceability information*, *mapping between models*, *scalability*, *change impact analysis*, and *tool support*. These five criteria are summarized in Table 2.1. The representation criterion characterizes how each approach represents traceability information. The mapping criterion indicates if an approach is capable of generating inter or intra traceability, i.e., the links are established between models at different levels of abstraction. The scalability criterion analyzes if it is possible to apply the approach to a large system. The change impact analysis criterion evaluates how the approach includes support for detecting the impact of changes on the related artifacts. Finally the tool support criterion evaluates whether the approach provides any kind of tool support for facilitating traceability.

Traceability is becoming a major feature of any MDE approach, since it is intrinsically related with the main idea of Model-Driven Engineering of transforming abstract models into more concrete ones, until a system implementation is achieved. Trace links can play a crucial role in this process, since they allow the developers to discover how a change in a model should be propagated throughout the rest of the models. Without this kind of trace information evaluating the impact of a change is a

time consuming and error-prone task. The approaches summarized in Table 2.1 represent the state-of-the-art in traceability approaches for MDE.

Table 2.1 – MDE traceability approaches summary (adapted from [28] and [46])

Approach	Representation	Mapping	Scalability	Change impact analysis	Tool support
Cleland-Huang et al. [16]	event-based subscriptions	inter	no	Yes	Yes
Cleland-Huang and Schmelzer [17]	SIG graph and event-based subscriptions	intra and inter	no	Yes	yes
Cleland-Huang et al. [18]	SIG graph and traceability matrix	intra and inter	partially	Yes	Partially
Ramesh and Jarke [62]	Traceability metamodels	intra and inter	yes	yes	yes
Egyed [24]	Footprint graph	intra and inter	yes	yes	partially
Jouault [42]	Trace model using ATL	intra and inter	no	no	yes
Kolovos et al. [49]	metamodel in EML and trace model in UML	inter	partially	no	Yes
Falleri et al. [27]	Kermeta models	intra and inter	no	no	Yes

From this evaluation it should be pointed out that the path taken by some approaches offers great advantages. The external representation of the trace links, such as the one used in Kolovos et al. [49], that are later combined with the models they refer to, is a good method for implementing traceability, since it maintains the model decoupled from the traceability information. This satisfies the principle of separation of concerns, keeps the models clean and facilitates the evolution of the approach. Another important aspect discussed in [28] is that tool support is crucial for automating traceability generation in MDE, something that the majority of the approaches do not provide.

2.3 Software Product Lines

According to Clements and Northrop a Software Product Line (SPL), is a set of software-intensive systems sharing a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [19]. Each product in a SPL is developed by using the necessary components, taken from the core assets base, and tailoring them as necessary using pre-planned variation mechanisms such as parameterization or inheritance, adding any new components that may be required and assembling the collection according to the rules of a common, product-line-wide architecture. A prescribed way of software development allows a more economic approach. Building a new system becomes more a question of integration rather than implementation.

There are many approaches that, at first glance, could be confused with a SPL. One might even think that a Software Product Line is just a new name for older approaches. It is therefore important to describe what a SPL **is not** [19]:

- Fortuitous small-grained reuse;
- Single-System development with reuse;
- Just component-based development;
- Just a reconfigurable architecture;
- Releases and versions of single products;
- Just a set of technical standards.

It should be pointed out that many of the terms described above are present in a Software Product Line Engineering, and are used for assembling one (e.g., reconfigurable architectures, artifacts reuse) but the idea is that even though they are an integrating part of a SPL, they are not its definition. A product line is much more than just reusing code or integrating components. It is a classical example of: "The whole is greater than the sum of the parts" [63].

At its essence, a product line involves core asset development (also known as Domain Engineering) and product development (also known as Application Engineering) using the core assets, both under the supervision of technical and organizational management. Core asset development and product development from the core assets can occur in either order: new products are built from core assets, or core assets are extracted from existing products. Often, products and core assets are built in synergy with each other. Figure 2.11 illustrates this triad of essential activities.

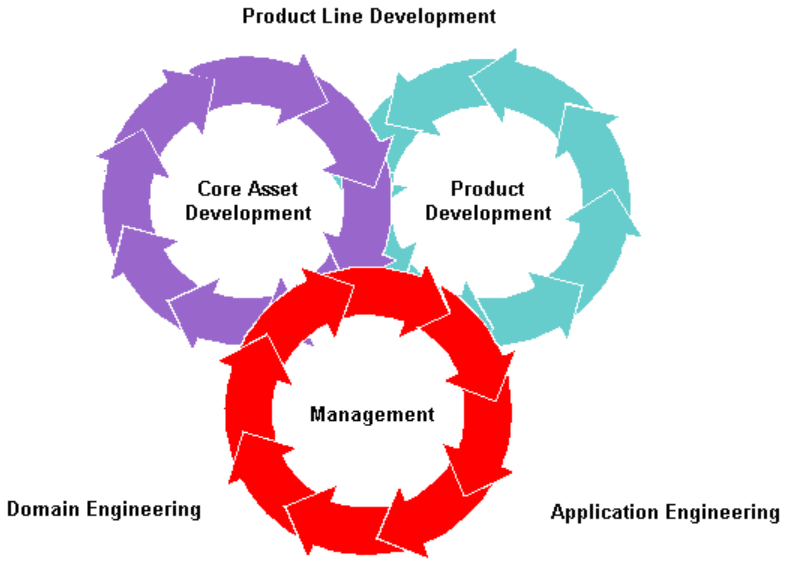


Figure 2.11 – Essential product lines activities (taken from [19])

Each rotating circle represents one of the essential activities. They are linked together and in perpetual motion, showing that all the activities are essential, inextricably linked, can occur in any order, and are highly iterative. The rotating arrows

indicate not only that core assets are used to develop products, but also that revisions of existing core assets or even new core assets might, and most often do, evolve out of product development. The diagram in Figure 2.11 is neutral in regard to which part of the effort is launched first. In some contexts, already existing products are mined for generic assets (perhaps a requirements specification, architecture, or software components) which are then migrated into the product line's core asset base. In other cases, the core assets may be developed or procured for later use in the production of products. Clements and Northrop [19], provide a very good insight into Software Product Lines principles and practices, and these three activities are discussed in great detail by the authors.

2.3.1 Traceability in Software Product Lines

Berg et al. state that the management of variability plays an important role in successful software product line engineering [8]. In fact, the all concept of SPL is based on the ability to derive different products from the same core assets. This variation can be achieved in many ways depending on goal to be achieved, or the level of abstraction being modeled, for instance a feature model can be used to model the product line scope, stating which products fall in or out of the product line family.

Due to the fact that any asset (a requirement, a test scenario, an architectural component, etc.) in a SPL can be included in several products, the impact that a change in one of these assets produces on the rest is even more complex to assess than in Single-System Engineering. This motivates the need for a universal variability management approach that is consistent and scalable and that provides traceability between variations at different levels of abstraction and across various generic development artifacts. The state-of-the-art in these approaches will now be discussed.

The existing work in traceability for Software Product Lines can be divided in three main categories [46]:

(i) variability on top approaches;

(ii) artifact level traceability approaches;

(iii) fine-grained traceability approaches.

Variability on top approaches use the variability model as the main reference for all traceability. The variability model sits on top of the software artifacts. It is used through all development stages and all artifacts are linked to some variation point. This case is illustrated in Figure 2.12. The traceability links point from artifacts (or parts of an artifact) to the variability model. This allows tracing the impact of a single variation point to all the artifacts that depend on him, allowing to easily analyze the impact of a change in the variability model. The downside is that since all links point strictly to the traceability model, that means that there is no traceability information between the other artifacts. This makes it hard to assess the impact of changes in, for example, design to implementation artifacts.

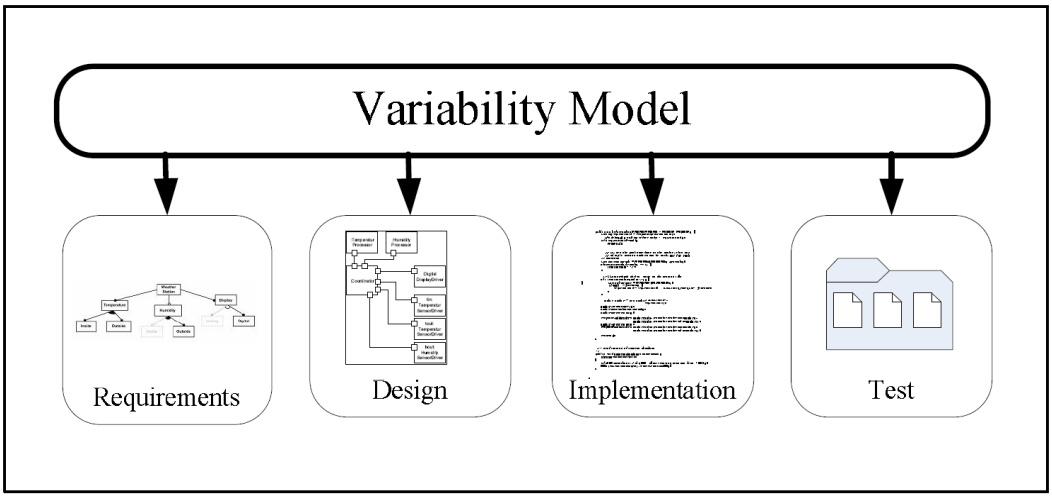


Figure 2.12 – Variability on top approach (taken from [59])

Orthogonal Variability Model (OVM) is described by Pohl et al. in the book "*Software Product Line Engineering: Foundations, Principles and Techniques*" [59]. They defined the variability of a software product line in a separate model, which is then related to other software development models. Traceability links are created between a development artifact (a requirement, a use case, a design model, etc.) and a variant, or variation point within the OVM. This relationship can be or an arbitrary granularity, e.g., variation points can be related to an entire design model, or to a single class. The multiplicities of these associations was defined by the authors as follows [59]:

- A development artifact can but does not have to be related to one or several variants (multiplicity 0..n).
- A variant must be related to at least one development artifact and may be related to more than one development artifact (multiplicity 1..n).
- A development artifact can but does not have to be related to one or several variation points (multiplicity 0..n).
- A variation point can but does not have to be related to one or more development artifacts (multiplicity 0..n).

A prototype variability model editor, called VARMOD [58], has been developed. In its current state, the tool only supports the visual representation of OVM models, without establishing any kind of links to real artifacts.

Berg et al. presented an approach to represent traceability in SPL, but without a real system experience and tool support [8]. They argue that traceability will improve the understanding of the system's variability, as well as better support for its maintenance and evolution. The authors focused on traceability of variability in SPL and state that establishing 100% of trace links between generic artifacts brings more benefits than relating only 80% of all the artifacts. In our opinion this supposition may not be valid, as empirical studies indicate that trying to capture all possible trace data without considering the actual project characteristics is most likely to fail [51]. In order to

capture 100 % of generic trace links one may need to ignore project specific characteristics, which may lead to a great number of unstructured and possibly unusable trace links. Another important aspect of this approach is the view that the authors have of the software engineering process. They understand that Single-System Engineering can be divided in two dimensions (the development process and the level of abstraction) and all development artifacts can be represented in these two dimensions. Variability however, adds a third dimension to explicitly capture the information regarding variability between the product line members. Figure 2.13 shows a conceptual model for traceability proposed by the authors that establishes the necessary mappings between all variation points with the artifacts represented in the two dimensional space (abstraction level and development process). The approach also lacks a description of how to define and store the trace links that are proposed by the authors.

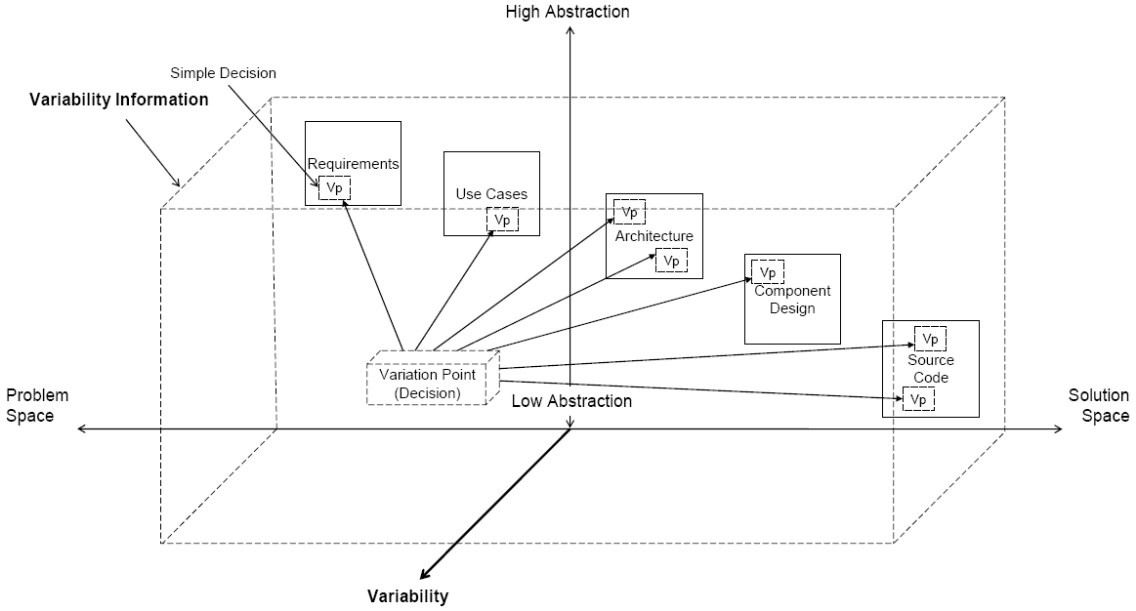


Figure 2.13 – Conceptual model for traceability (taken from [8])

Artifact level traceability approaches define directed and typed links between artifacts, on an artifacts level (these being usually files). This process is illustrated in Figure 2.14.

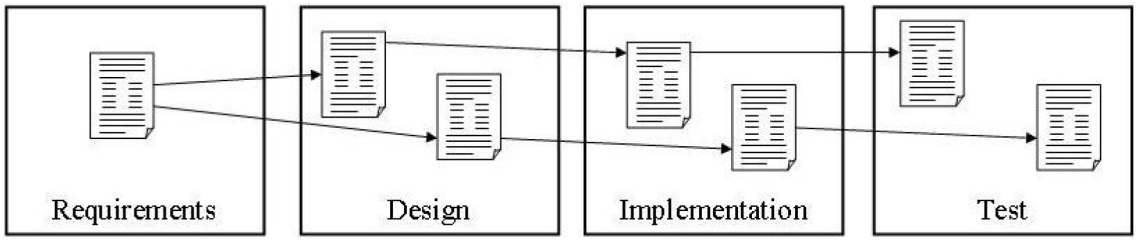


Figure 2.14 – Artifact level traceability approach (taken from [59])

Ajila and Kaba manage the evolution of the software product line recurring to traceability [2]. They identified three distinct sources of possible changes introduced in product lines: (i) changes in an individual product; (ii) changes in the entire product line; (iii) importing an architectural component from an individual product into the product line. Their work is more focused in defining a reference model to represent

horizontal and vertical traceability for evolution purposes. They do not discuss the actual means of handling traceability.

Knauber and Schneider describe an approach that enables traceability between the test case of a variant and its respective implementation [47]. The authors propose to implement variable features as aspects, and weave them to the core assets (the common features) of the product when needed. The test code is also supported by aspects, providing an automatic method for adapting it to the respective variant. Their approach shows that it is possible to combine the variable code with its test code into a single aspect. This provides traceability of a variable feature to its test cases.

Mergel et al. present an repository for product line assets [52]. The repository supports various kinds of information, for retrieval purposes. An asset header includes information about the identity of the asset, its qualification, administrative details, the work product category (e.g., domain model, source code) and the kind of representation. Relationships between the assets can also de defined, enabling traceability relationships between them. Relationships can be navigable in both directions, have an m:n cardinality, and a relationship type associated.

Jirapanthong and Zisman propose an approach with automatic generation of trace relationships [40]. The authors used FORM [45] that provides a reference model with two levels, two domains and specialized documents. XML Schemas for all these documents were also defined. At the core of this system lies a set of conditional rules, based in the XQuery language, that are responsible for parsing the XML documents and generate the relationships that exist between them. The authors use six different groups of relationships, and ten different kinds. A prototype tool called XTraQue [41] which implements the system described is also provided by the authors.

Fine-grained traceability approaches allow for the definition of fine-grained, directed and typed links between parts of an artifact, e.g., design elements in a diagram or function defined in a class. The methodology of these approaches is illustrated in Figure 2.15.

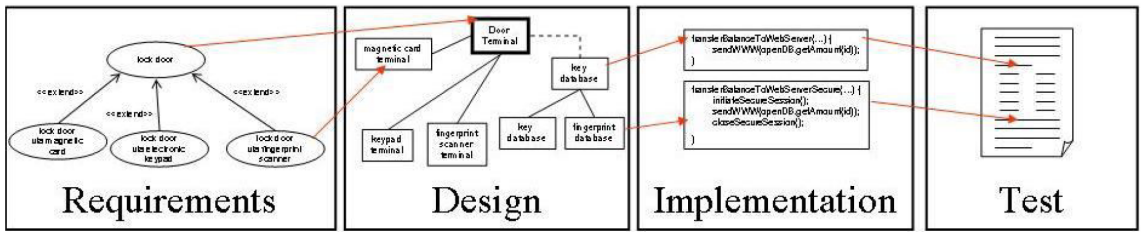


Figure 2.15 – Fine-grained traceability approach (taken from [59])

Zisman et al. present an approach that automatically generates and maintains bi-directional trace links between requirements specifications of product lines [73]. The approach is based in three types of specifications. The commercial requirements specification (CRS) is represented in natural language. The functional requirements specification (FRS) is represented recurring to use cases. The requirements object model

(ROM) is expresses in UML. The approach is capable of generating inter-requirements links and links between requirements and object models.

Luttikhuizen et al. describe a modeling concept for product families [51]. They introduce a traceability model that supports forward and backward traceability from requirements to refined requirements, architecture and test cases. The traceability model is based on the identification of requirements with requirement tags in the existing specification documents, and these relationships are modeled in several tables. The Traceability Table associates each requirement with one or more higher level requirements and vice versa. The Allocation Table associates each requirement with one or more lower level components. The Details Table associates each 'container' requirement (e.g., a use case or some general safety concept), with all its detail requirements. The use of tables might prove to be the major drawback of this approach. Form our experience, tables do not scale well and if the number or requirements is large enough, the table might prove to be unmanageable or quite hard to use in practice. This may prove to be the biggest disadvantage of the approach.

Bayer and Widen give a general overview of the traceability needs and integration regarding product lines [7]. The authors argue that traceability is a key aspect for the successful and sustained development and maintenance of a SPL infrastructure. They provide a list of traceability requirements that consist of:

1. Should be based in the semantics of models used in SPL;
2. Should be customized to capture only the relevant types of traceability;
3. Should be capable of handling variability;
4. It is preferable to have a small set of trace links;
5. As automated as possible.

The paper continues by describing how traceability is integrated in PuLSE [6]. PuLSE uses a general metamodel of development artifacts as the core of the product line architecture. This metamodel includes traceability links, and it is possible to create certain types of traces. Their approach consists of first customizing and tailoring this generic metamodel to a specific SPL context. On a second phase this customized metamodel becomes the basis for creating models and establishing trace links.

Moon et al. propose a metamodeling approach to trace variability between requirements and an architecture in SPL [53]. The approach is based on the principle that variation points may appear in all generic artifacts, since they are the realization of variability. This presents the need to capture the traceability between the variation points at least for requirements and architecture. The authors defined two metamodels for requirements and architecture, which extend the Reusable Asset Specification proposal recently adopted by OMG. The traceability between these artifacts is achieved by defining trace relationships between the two metamodels. The only variation mechanisms provided are wither common features or optional. Variability is represented as an attribute associated with an artifact (requirements, use cases and components)

2.3.2 Discussion

A survey in traceability approaches for SPL was conducted in the context of the AMPLE project [46]. Some of the approaches evaluated were presented in the previous section of this dissertation. The approaches chosen to be discussed in this dissertation were the ones that seem to provide a more complete traceability solution (with regard to the evaluation criterion). The survey also used a common set of characteristics for evaluating the performance of each approach. These criteria were: *representation of traceability information*, *mapping between models and granularity of relationships*, *scalability*, *change impact analysis*, and *tool support*. These five criteria are summarized in Table 2.2. The representation criterion characterizes how each approach represents traceability information. The mapping criterion indicates if an approach is capable of generating forward and backward traceability and the level of granularity (coarse-grained means that only an entire artifact is traceable, fine-grained traceability means that it is possible to trace parts of an artifact). The scalability criterion analyzes if it is possible to apply the approach to a large system. The change impact analysis criterion evaluates how the approach includes support for detecting the impact of changes on the related artifacts. Finally the tool support criterion evaluates whether the approach provides any kind of tool support for facilitating traceability. Some criteria were not possible to evaluate due to the lack of information in the available bibliography. They are marked as "not discussed" in Table 2.2.

Table 2.2 – SPL traceability approaches summary (adapted from [46])

Approach	Representation	Mapping and Granularity	Scalability	Change impact analysis	Tool support
Pohl et al. [59]	Directed link	Backward and forward, arbitrary granularity	yes	yes	no
Berg et al. [8]	Directed link	Fine-grained	not discussed	not discussed	no
Ajila and Kaba [2]	Directed link	Coarse-grained	not discussed	yes	ad-hoc tool set
Knauber and Schneider [47]	Directed link	Forward, coarse-grained	not discussed	not discussed	Junit and AspectJ
Mergel et al. [52]	Directed link and meta information	Backward and forward coarse-grained	yes	not discussed	not publicly available
Jirapanthong and Zisman [40]	Directed link	Backward and forward, coarse-grained	not discussed	not discussed	prototype (XtraQue)
Zisman et al. [73]	Directed link and meta information	Backward and forward, fine-grained for requirements	not discussed	not discussed	prototype
Luttikhuizen et al. [51]	Directed link and meta information	Backward and forward Fine-grained for requirements	not discussed	yes	no
Bayer and Widen [7]	Directed link	Backward and forward, arbitrary granularity	not discussed	not discussed	ad-hoc tool set
Moon et al. [53]	Directed link	Backward and forward, arbitrary granularity	yes	not discussed	not discussed

Traceability is considered by many researchers and practitioners as a very important concern for product line engineering. Some research work has been done on this issue, but a major drawback is the availability of appropriate tool support. Without tools, traceability tasks are performed manually and fail in achieving their goal. The purpose of traceability is to link the different artifacts used throughout the software lifecycle and to provide rationale about how they are linked. In traditional Single-Systems this is achieved by defining trace links between the elements of the two dimensions of development: inter traceability (elements at different levels of abstraction) and intra traceability (elements at the same level of abstraction). This trace information can then be the means by which one can prove that the delivered product is according to the agreed requirements, and to also prove the absence of unnecessary functionalities. In a product line scenario environment this is taken a step further, by adding a third dimension to the traceability process, orthogonal to the previous ones, to handle variability and its implications [3]. This new dimension introduces new challenges that are not present in traditional (non SPL) development. Another aspect relevant to SPL development is that due to the fact that product family members share a common set of assets, a change introduced in a core asset might affect a great deal of different variants from the family of products.

In the approaches described previously, some of the authors seem to prefer to model variability using a model created for that purpose, and to create traces from variability elements to other artifacts [8, 59]. The proposal of Orthogonal Variability Model [59] seems to provide a comprehensive and complete approach to address all the stages of SPL development, including the definition of trace links between the several artifacts used. However, in our opinion, this approach has the drawback of coupling the traceability information in the variability models. This imposes the need to use the OVM metamodel and establish trace links to it. In our opinion this limits the use of the approach as it binds the user to a metamodel, without giving him the freedom to choose how to represent the product line variability. It also incorporates traceability concepts inside the variability model, which goes against the principle of separations of concerns. We believe that traceability should be represented using a separate metamodel to facilitate an easier integration with different models (variability models, requirement models, architectural models, etc.) making the approach more reusable and easier to maintain and evolve.

Other authors prefer to focus on traceability for certain aspects of SPL, keeping it comprehensive but not exhaustive [7, 47, 53, 73]. In our opinion, the first solution seems to be more cohesive, because even though providing traceability to a certain aspect of SPL development might be useful in some cases, in the end an approach that facilitates the tracing of elements across the entire software lifecycle proves to be more desirable to the developer, as it allows a unified representation of trace information, instead of several different and separated solutions. All the approaches go towards the need to create tools that support traceability. However, a gap still exists in this area, since the majority of the approaches do not provide any tool support, or only a prototype implementation.

2.4 Traceability Tools

This section presents a survey on traceability tools. The tools presented here were evaluated by AMPLE's FCT/UNL research group. This survey was an activity relevant to the AMPLE project research, but also performed in the context of this thesis

preliminary work. The objectives of the survey were to investigate the current features provided by existing tools in order to assess their strengths and weaknesses, and identify possible fields of improvement. The tools were evaluated with regard to five criteria: *management of traceability links*, *traceability queries*, *traceability views*, *extensibility* and *support for SPL and MDE development*. We believe that these criteria are crucial for this kind of tools since they provide the basic support to satisfy traceability requirements (creation of trace information and querying it). Other important concerns regarding SPL development were also evaluated, since they evaluate the capacity of each tool for handling this paradigm of software development. In our opinion, this set of criteria can effectively evaluate if a tool responds to the problems described in Chapter 1. The management of traceability links criterion evaluates how each tool is capable of creating trace links (manual or automatic) and what kind of trace information is generated. The traceability queries criterion analyzes what type of basic search query is provided by each tool and if advanced queries are supported (coverage analysis and change impact analysis). The traceability view criterion characterizes the supported views for the traceability information stored by each tool. The extensibility criterion evaluates if a tool as any extension mechanisms. Finally, the support for SPL and MDE development criterion indicates if a tool as any mechanism that supports these software paradigms.

RequisitePro [37] is a powerful, easy-to-use requirements management tool that helps teams manage project requirements comprehensively, promotes communication and collaboration among team members, and reduces project risk. A RequisitePro project includes a database and optionally includes documents. The database is used to store the document types, requirement types and descriptors (attributes), discussions, information about requirement traceability and user/group security. The project and document templates used include the following structural information:

- Document types, such as glossary document, vision statement, and use cases (which outline how the system behaves);
- Requirement types, which are categories of requirements such as features, use cases, supplementary specifications, and so on;
- Requirement attributes, which describe the requirements in terms of priority, status, stability, and other characteristics that are user defined.

The tool supports manual creation of trace links and import from some file formats (Word and CVS). It creates forward and backward traceability and it is possible to create associations with parts of an artifact. It is possible to query requirements and trace links, and filter the desired information. The types of views supported for traceability are a Traceability Matrix View and a Traceability Tree View. It has no built-in mechanism for SPL or MDE.

Borland CaliberRM [13] is an enterprise software requirements management tool that facilitates collaboration, impact analysis and communication, enabling software teams to deliver on key project milestones with greater accuracy and predictability. CaliberRM also helps small, large and distributed organizations ensure that applications meet the needs of end-users, by allowing analysts, developers, testers and other project

stakeholders to capture and communicate the users' voice throughout the application lifecycle.

This tool classifies requirements as objects defined in a hierarchy and offers requirements validation and error detection during requirements specification. All the artifacts are stored in a central repository, which facilitates the collaboration, cooperation and communication between all members involved in the project. The trace links are transitive and need to be created manually by developers. It supports backward and forward traceability and allows the creation of links between a requirement and any other artifact (source code, test cases, use cases, design). Trace links can be queried by means of using filters which will only return the requested information. Impact change analysis and detection of suspect links (due to a change in requirements) is also provided. The views regarding traceability are a Traceability Matrix, Traceability Graph and several types of reports. It has no built-in mechanism for SPL or MDE support.

RaQuest [65] is a requirements management tool designed to integrate with Enterprise Architect a UML modeling tool developed by SparxSystems. RaQuest enables you to create lists of requirements, print them out, and export them as HTML or Word documents. Moreover, it is possible to display relationships or matrixes useful for analysis of change impact or coverage between requirements. The tool also allows the developers to relate requirements to Enterprise Architect items. Even though it was not originally designed to work with UML, it now includes functionalities that allow the transformation of requirements into UML diagrams smoothly. The creation of trace links is manual, but requirements can be imported from CVS, Excel or Word documents. It is possible to associate requirements with EA artifacts. The trace links provide forward and backward traceability. RaQuest supports change impact analysis and detect some types of inconsistencies between them. The views available for traceability are a Graph of dependencies, and traceability matrix. RaQuest does not support SPL development, but it allows the transformation of requirements into UML (as EA artifacts).

Telelogic DOORS [69] is one of the most important requirements management application. It provides many features to capture, track and manage user requirements. It also addresses the tracking and management of requirements throughout the project life cycle using a variety of features, such as views, links and traceability analyzes. The trace links can be created manually but the information can also be imported from many formats (Word, Excel, ASCII, Interleaf, and RTF). DOORS provides traceability of requirements throughout the project lifecycle, allowing forward and backward traceability and associations with parts of an artifact. Any data stored in the project's database can be queried, and the information returned can be filtered using a great number of options. It also provides change impact analysis and requirements coverage analysis. DOORS document hierarchies may be viewed graphically and traceability may be viewed as "tree" structures in the Traceability Explorer. No support for SPL or MDE is offered.

Contour [39] is a web based collaborative requirements management solution. Contour projects contain groups of artifacts such as requirements, use cases, test cases and many other types. Each project may have multiple groups and can be configured to meet each project's needs. Each group can contain as many artifacts of that type as necessary (e.g.,

the requirements group can contain many artifacts of type requirement). In addition user roles can be given access to projects. Contour provides a flexible way to setup data, organizing the several artifacts using the following objects:

- *Containers* - Solutions, products, projects, modules, etc. A high level collection.
- *Groups* - A collection of similar artifacts such as a group of requirements, documents, use cases, test cases, etc.
- *Folders* - A way of grouping artifacts.
- *Artifact* - A single item with a unique set of fields and behaviors. Examples of artifacts are requirements, use cases, test cases, etc.

Contour enables the creation of trace relationships between any combination of project artifacts and even to artifacts residing in other projects. The creation of relationship links is manual, but it is possible to be import artifacts from CVS or Word 2003 XML files. The tool provides forward and backward traceability, change impact analysis, suspect links detection, and general purpose queries to browse the information stored a project. Two traceability views are available to the user, a Traceability Matrix and a Trace Relationships Report. Contour does no support SPL or MDE.

GatherSpace [30] is a web based requirements management and use case tool for managing and sharing software requirements. It enables developers to focus on the requirements without concern for upgrades, infrastructure, and maintenance. It allows for the creation of multiple projects and the assignment of multiple users to the projects. Features, Requirements, Use cases and actors can all be modeled inside the tool. GatherSpace considers the software development cycle as a pyramid with three levels (shown in Figure 2.16).

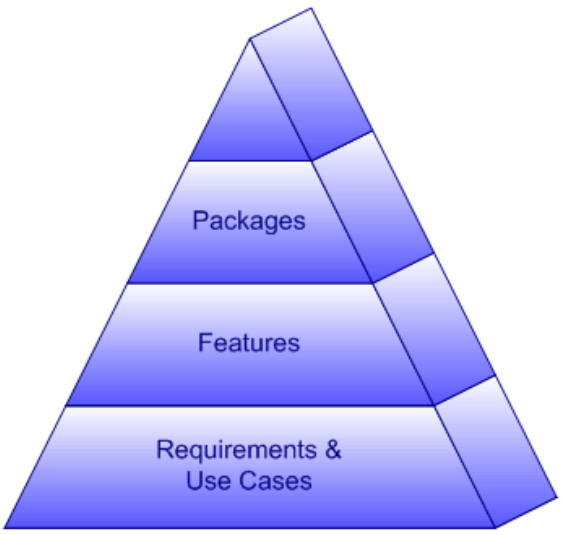


Figure 2.16 – The GatherSpace software requirements pyramid (taken from [30])

At the top level, reside the packages. Packages are the logical grouping of features. At the level below reside the Features. Features are the beginning point of constructing the project specifications. A feature is a simple description of something that the system

will do to solve the problem at hand. The requirements exist to support or supplement features. A requirement is a description of how a feature is carried out. Requirements can be associated with the features that they describe. Finally a use cases is created to specify the interaction of an actor with the system to achieve a goal. Use cases are associated with requirements and in turn with its feature. All the artifacts and links are created manually in GatherSpace. No type of query can be submitted and only coverage analysis is provided to the user. The only view available is a Traceability Report. This tool has no built-in mechanism for SPL or MDE support.

The Trace Analyzer [23] is a tool for generating and validating the traceability links among software models, source code, and test scenarios. Models may include any product relevant model elements such as requirements, architecture, (UML) design, and test scenarios. This tool works based on the commonality principle [25]:

"Commonality: if A is known to trace to some source code CA and B is known to trace to some source code CB then a trace dependency exists if CA and CB overlap."

Given a set of input models, source code and hypothesized trace links, the Trace Analyzer's primary task is to reason about the ownership of the source code by the different model elements. Its secondary task is to infer trace dependencies among the model elements based on the ownership information. Trace links are created automatically from the set of input data. No trace query of any kind is available, but the identification of some inconsistencies is possible. The tool provides several views, being the most important the Footprint Graph, the Model to Model view and a textual report. Trace Analyzer does not support SPL or MDE.

2.4.1 Discussion

To provide a complete and effective traceability solution, tool support is essential, since it allows system developers to use it in real software development contexts. Without tools, any approach is doomed to failure, because in manual traceability schemes it is not easy to maintain the trace information updated and it is very hard to reason and evaluate change impact analysis, coverage analysis, etc. A summary of the survey results is presented in Table 2.3. Some criteria were not possible to evaluate due to the lack of information in the available manuals. They are marked as "n.d." (not discussed).

The traceability tools survey that was presented in the previous section shows that none of them as support for SPL and only one implemented some MDE techniques [65]. Even though some of these tools provide very complete solutions to traceability for Single-System development, the lack of support for addressing variability and establishing trace links between software artifacts and variation points in a product line is the biggest downside to their use in SPL. Another problem is that it is not possible to adapt them, due to their closed source and copyright infringements. Even though some tools provide some extension mechanisms, like the possibility to define new templates for reports, in the bottom line, these extensions prove to be very basic and cannot be used to extend their capabilities to enable traceability for product lines.

Table 2.3 – Traceability tools summary

Tool	Creation of Trace Links	Mapping	Type of Query	Change Impact Analysis	Covering Analysis	Views	Extensible	Support for SPL and MDE
RequisitePro [37]	manual	part of artifacts, forward and backward	filters	yes	no	matrix, tree	no	no
Borland CaliberRM [13]	manual	requirements to artifacts, forward and backward	filters	yes	no	matrix, graph	yes	no
RaQuest [65]	manual, imports requirements from file	requirements to artifacts, forward and backward	n.d.	yes	no	matrix, graph	no	partial support for MDE
Telelogic DOORS [69]	manual, imports requirements from file	requirements to parts of artifacts, forward and backward	custom queries, filters	yes	yes	tree	yes	no
Contour [39]	manual, imports artifacts from file	artifact to artifact, forward and backward	filters	yes	no	matrix, report	yes	no
GatherSpace [30]	manual	requirement to other artifacts, forward	n.d.	no	yes	report	no	no
Trace Analyzer [23]	automatic	models to source code, models to models, forward	n.d.	no	yes	matrix, graph, report	no	no

2.5 Summary

In this chapter a state-of-the-art in traceability research and tools was presented. The chapter began by giving a general view of the concepts regarding traceability, and the advantages that trace methods bring to software development. Traceability can yield significant improvements to software developers, by providing information that aids in the detection of problems existing in a system (e.g., orphan code). The remaining sections introduced the topics of Model-Driven Engineering and Software Product Lines and analyzed some approaches that have been proposed in these domains. We have discussed the strengths and weaknesses of these approaches. The major problem found with the majority of these approaches is the lack of appropriate tool support, a critical component in any traceability solution. Another important topic discussed was a traceability tools survey which presented our findings regarding the current available tools. The greatest shortcoming that we found was the tools ability to handle SPL development. We found it to be nonexistent in the majority of the tools analyzed.

The next chapter will present our proposal for a Model-Driven Traceability Framework that was developed as the result of this master dissertation work.

Chapter 3. A Model-Driven Traceability Framework

In this chapter we present our proposal for a traceability framework that provides a solution for defining trace links in the context of product lines development. The traceability framework that we are proposing is meant to provide an open and flexible platform to define trace links between the different artifacts used during SPL development. The general structure of the framework was already discussed in Chapter 1. In the following sections we will describe the framework structure and implementation in more detail. The base of our framework is a traceability metamodel that will be presented to the reader. The reminder of the framework is built on top of this metamodel. We describe the architectural structure of the framework and how its several components are connected, along with the hotspots provided for instantiation to different development scenarios. The implementation that was achieved and the decisions that were made are also discussed. The final sections of the chapter present an instantiation example of the framework for a scenario of defining trace links between features (variability model) and use cases (requirements model).

3.1 Framework Description

Our proposal aims at providing an open and flexible platform to design and implement tools and methods that allow developers to define and store trace links between the different artifacts used during SPL development [64]. In this section, we will present and discuss the main topics regarding our SPL traceability framework. We begin by describing the traceability metamodel adopted by our framework (Section 3.1.1) followed by the framework's architecture, as well as the class diagram for the main modules (Section 3.1.2). Since SPL development was the focus of this work, another important point is the use the of the variability model as the main reference for tracing

the SPL artifacts. However, the design of the framework is generic enough, so that it may be applied to other software development scenarios.

The following main functionalities are provided by our traceability framework:

1. Creation and maintenance of trace links between a variability model and other software artifacts (UML models, architecture models, source code);
2. Persistent storage of trace links;
3. Searching for specific trace links using pre-defined or customizable trace queries;
4. Flexible visualization of trace query results using different types of trace views.

3.1.1 Traceability Metamodel

The traceability metamodel is the basis of the framework that we propose and is depicted in Figure 3.1. It is centered on the assumption that all trace information can be represented by a directed graph [64].

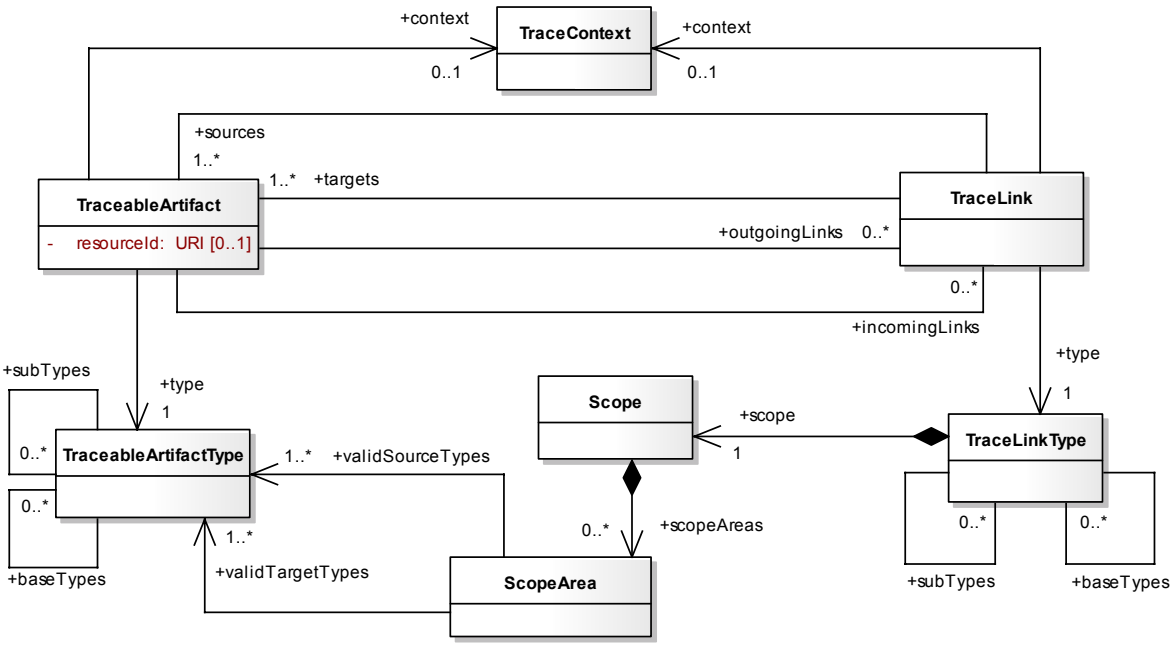


Figure 3.1 – Traceability metamodel

The main elements of the metamodel are the following:

- A TraceableArtifact represents a (physical) artifact that plays a role in the development cycle. The granularity of such artifact is arbitrary. It may represent a requirement, a UML diagram, an element inside a diagram, a class or a method inside a class. An artifact is unambiguously identified by a locator (resourceId), which describes where this artifact is located (such as in a file or a directory) and how it may be accessed.

- TraceLink is the abstraction for the transition from one artifact to another. An instance corresponds to a hyperedge[3] linking two artifacts in the trace graph. A transition is always directed; therefore a from-to-relation between artifacts is created by a trace link (between source and target artifacts).

- During the process of tracing information about the design of a software system, different artifacts of different types must be taken into account. For this reason each TraceableArtifact has an instance of TraceableArtifactType assigned. This type separates artifacts from each other. Artifact types may be grouped in a hierarchical manner, which mimics the concept of multiple inheritance, known from object orientation.

- Analogous to the type of an artifact, each link has a type because the relationship between two artifacts may differ. Examples for such types would be *contains*, *depends of* or *is generated from*. For this reason each instance of TraceLink is assigned to an instance of TraceLinkType.

- The existence of an artifact or the relationship from one artifact to another may be justified in some way. Not all artifacts and transitions would require such a justification, for example a "contains" transition is rather self explanatory. The attachment of additional information to artifacts and links can be modelled by attaching a TraceContext to relations and/or artifacts.

- Links of a certain type may only be valid between artifacts of a certain type. A link of type "contains" may be valid between a *Method* and a *Class*, but not between two *Architectural Models*. The narrowing of validity area of link types is modelled via the introduction of the elements ScopeArea and Scope.

3.1.2 Traceability Framework Structure

An architectural overview of the framework is show in Figure 3.2 as a UML component diagram, where the four main modules and their relationships are depicted. The framework has three hotspots that are meant to be instantiated to provide trace mechanisms for each desired scenario or stage of development. These extension mechanisms will be described in more detail in Sections 3.2 and 3.3. The ATF module is based on the traceability metamodel described in the previous section. It is responsible for providing a persistence mechanism for storage of trace information (trace artifacts, trace links, rationale, etc.), and for providing basic query and retrieval mechanism for accessing the stored information. This component was developed by SAP (one of the industrial partners in project AMPLE). The remaining framework modules were built on top of this persistence structure.

Inside the "Framework Core" package reside three hotspots of the framework: *TraceRegister*, *TraceQuery* and *TraceView*. The *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, 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

[3] A hyperedge is a set of vertices of a hypergraph [55].

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 this 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.

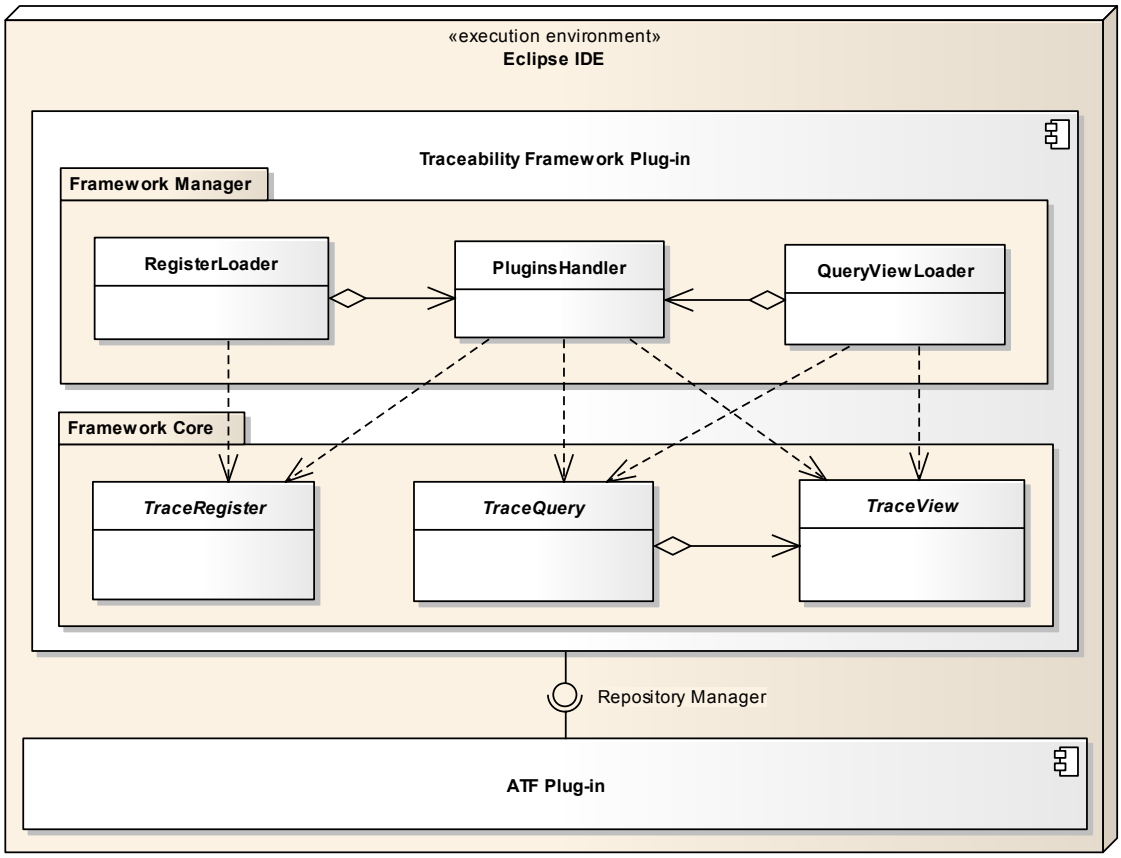


Figure 3.2 – Traceability Framework architecture overview

The workflow for defining new trace links using a register is shown in Figure 3.3. 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 residing in ATF. As mentioned previously, this step can be automatic, manual or a combination of both.

Figure 3.4 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 which type of artifacts to be queried). The chosen Trace Query will then retrieve the relevant links and artifacts from ATF and pass then to the chosen Trace View for visualization.

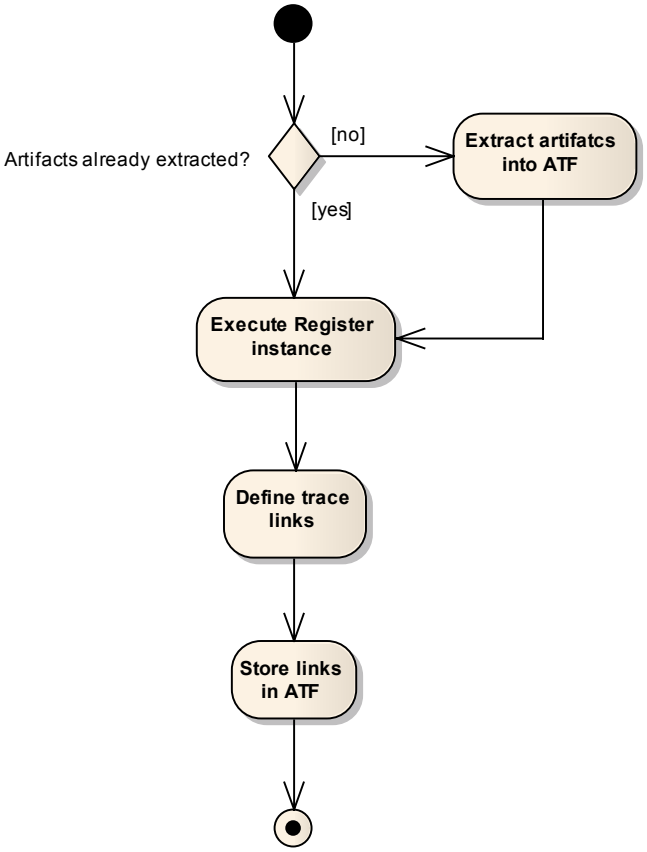


Figure 3.3 – Trace link definition workflow

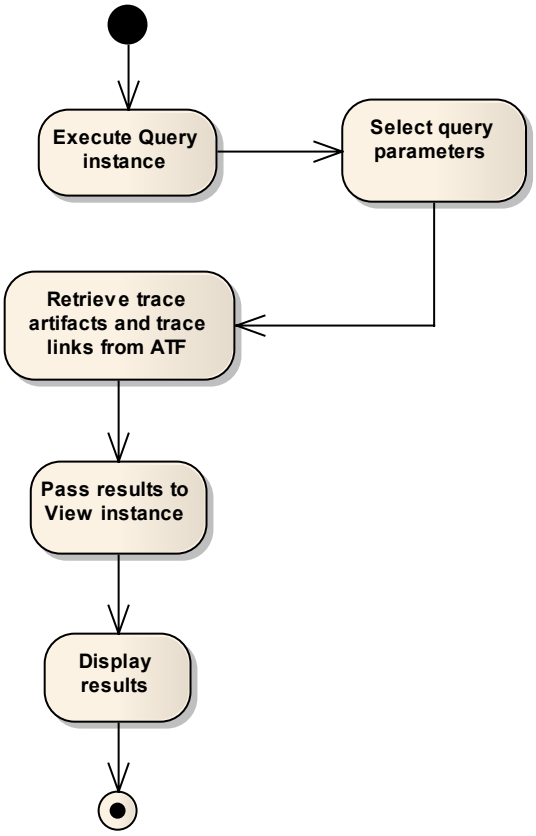


Figure 3.4 – Trace query and trace view workflow

3.2 Implementation

Part of this dissertation work was to implement a prototype version of the traceability framework. The technological platform chosen was Eclipse IDE, which was a requirement for the AMPLE project, where this work is included. As mentioned previously, the ATF module was implemented by one of the AMPLE partners. The traceability metamodel described in the previous section has been implemented using EMF [15]. On top of EMF, several reusable components were also used, such as, EMF Query[4] for information retrieval and Teneo[5] which was used as the abstraction layer between the EMF and the actual database layer that is used to provide persistence for the EMF model instances. ATF provides the following extension point:

- **net.ample.tracing.core.traceExtractor** – This extension point is used to plug in additional extractors to retrieve data into ATF.

The remaining modules were part of the work developed during this thesis. To achieve this goal, an Eclipse plug-in with the required extension points was implemented [12]. This mechanism allows framework developers to implement new instances of each hotspot by implementing an extension of a specific extension point, thus adapting the framework to the desired scenario. Figure 3.5 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). The extension points defined were:

- **net.ample.tracing.framework.core.traceRegister** – This extension point is used to plug in additional trace registers for establishing trace links between SPL artifacts. Implemented in *AbstractTraceRegister*.

- **net.ample.tracing.framework.core.traceQuery** – This extension point is used to plug in additional trace queries for performing new types of queries implemented in *AbstractTraceQuery*.

- **net.ample.tracing.framework.core.traceView** – This extension point is used to plug in additional trace views. Implemented in *AbstractTraceView*.

The schema for each extension point can be found in Annex 1 - , 2, 3 and 4. 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 of a particular instance (e.g., the method that performs the necessary operations for implementing the feature interaction query).

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 3.5) 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 just 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

[4] http://www.eclipse.org/modeling/emf/?project=query
[5] http://www.eclipse.org/modeling/emf/?project=teneo#teneo

path chosen, a trace register can be used to perform maintenance of trace artifacts and links.

Figure 3.5 also shows how the Framework Manager is used to load the framework instances. Each hotspot 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.

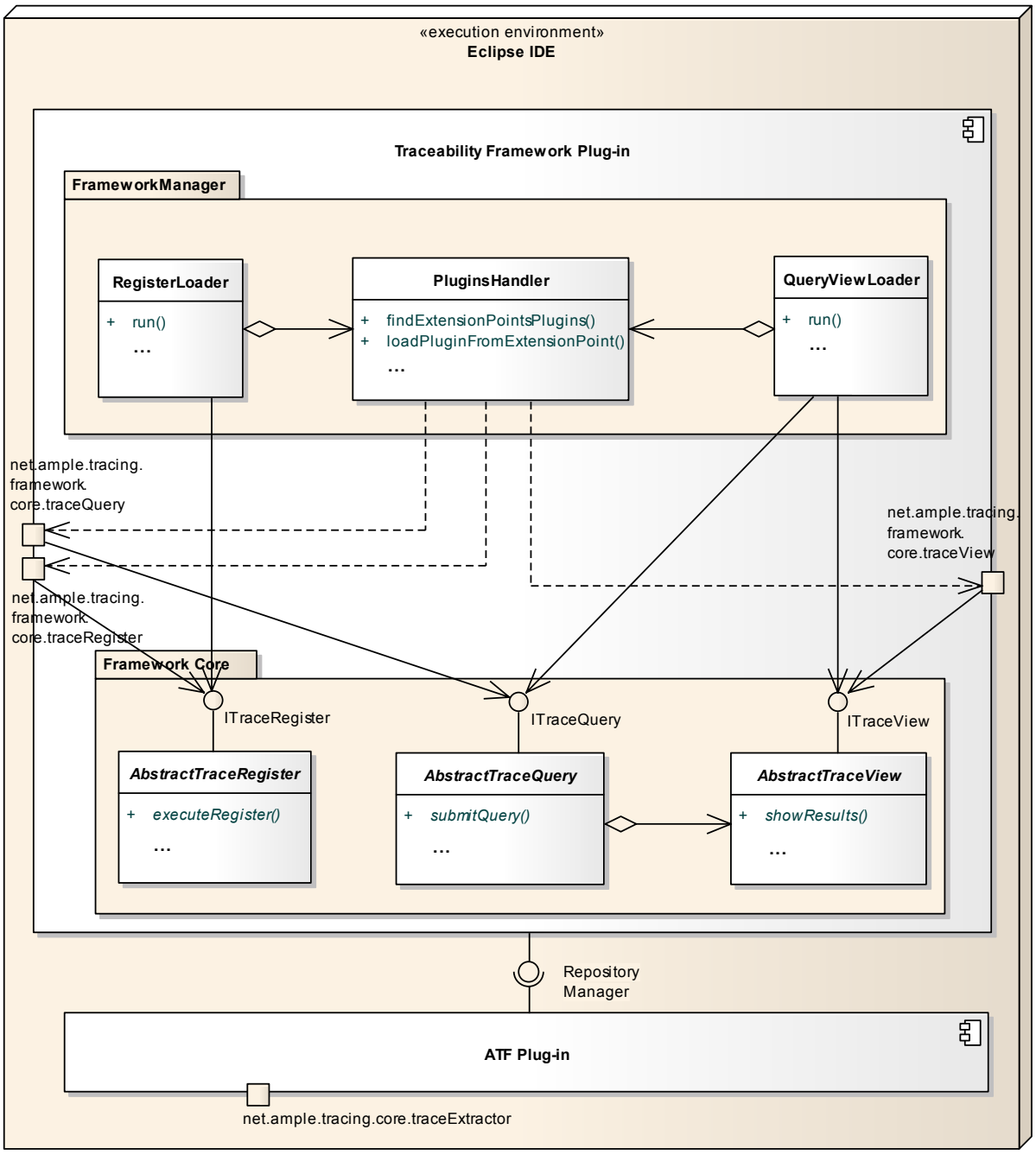


Figure 3.5 – Traceability Framework components diagram

Figure 3.6 illustrates the context menu provided by the framework, showing all the trace queries and trace views that are available. The new queries and views are automatically added to this list and the user is simply required to choose which instance is to be executed.

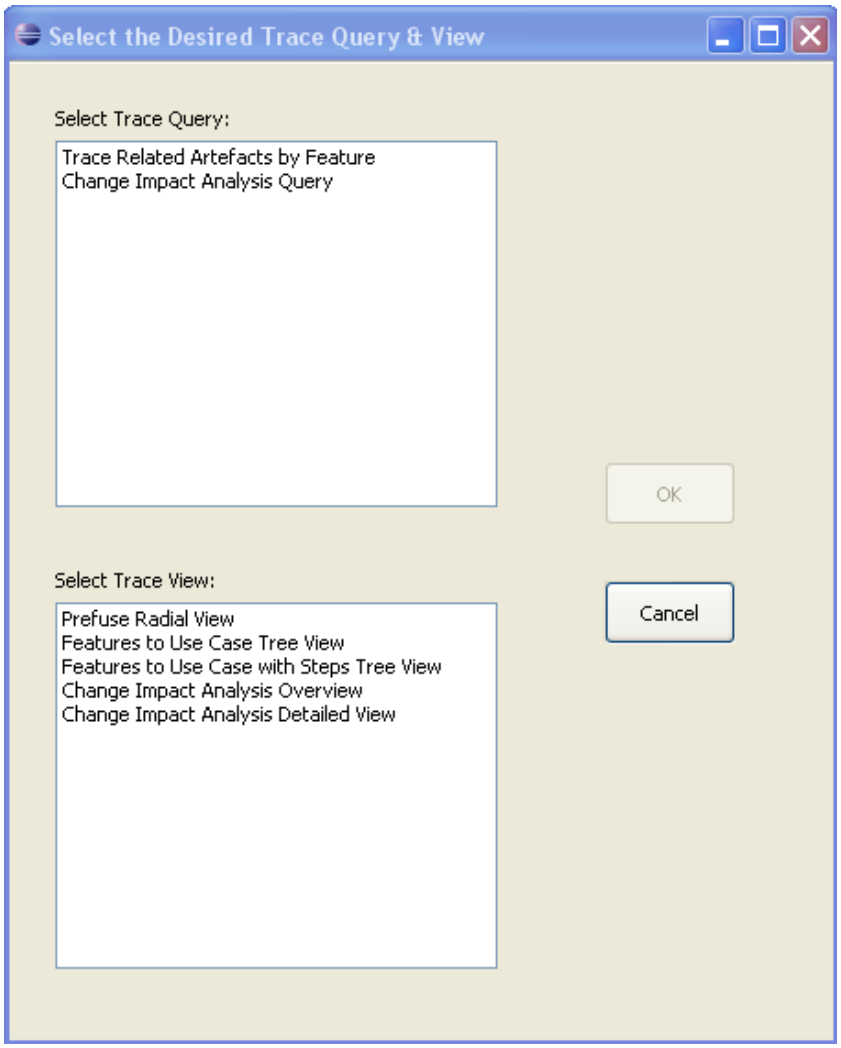


Figure 3.6 – Trace query and trace view selection window

3.3 Framework Instantiation

This section presents an instantiation of our framework that addresses the tracing between feature and use case models. Our aim is to illustrate how the framework can be used and extended to address concrete scenarios of traceability in SPL development. All the framework extension points, presented in Section 3.1.2, are illustrated in this instantiation. A detailed description of how the Eclipse extension points mechanism works and its usage will not be explained in detail in this dissertation, as it falls out of the scope of this work. Some books have been written addressing this subject, and many tutorials and articles are also available on the internet. The Traceability Framework User Guide can also be found in Annex 5 - , and it gives a detailed explanation of the framework usage and how to implement each hotspot instance as a new Eclipse plug-in.

The features used to specify the commonalities/variabilities were created using the Feature Modeling Plug-in (FMP) [4] that allows the creation of feature models and feature model configurations. The requirements were modeled using a use case model designed in Rational Rose [36]. The idea is to implement extensions as described in the previous section (using the ATF net.ample.tracing.core.traceExtractor extension point) to parse the input models. Once that step is concluded, the *TraceRegister* is executed to manually define trace links between the artifacts.

3.3.1 Extractor Instantiation

To provide an instance for parsing use case models created using Rational Rose, one must simply create an extension to the net.ample.tracing.core.traceExtractor and implement the respective Java class that will parse the input file and store the extracted information into the repository. ATF provides an abstract implementation of an extractor (*AbstractTraceExtractor*). The only method missing is the `run()` that will be invoked when this extractor is selected. Figure 3.7 depicts this instantiation.

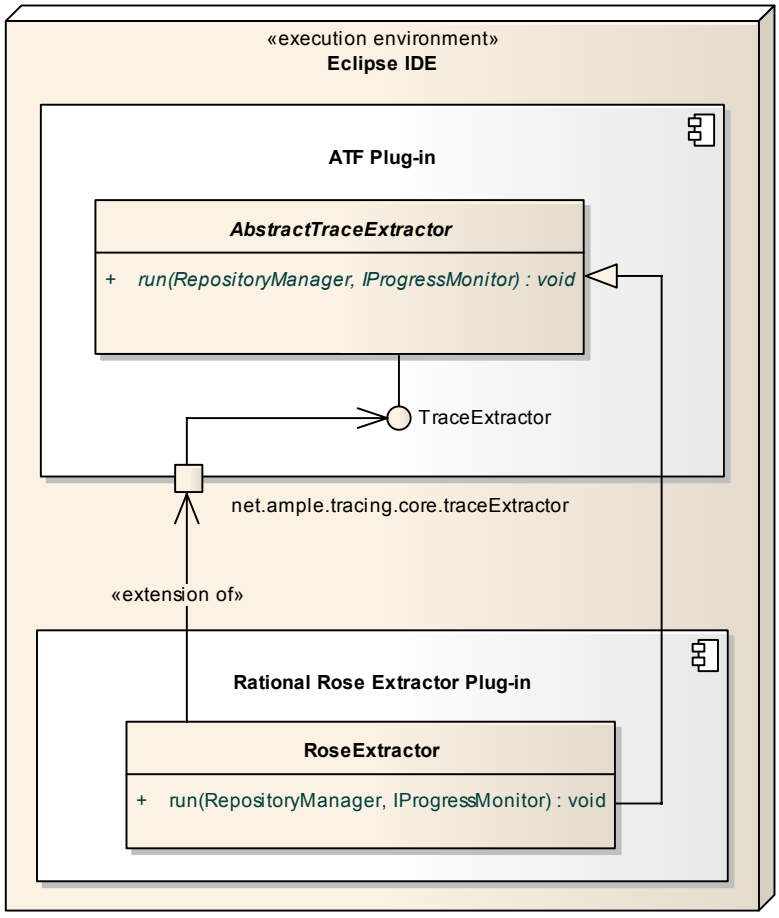


Figure 3.7 – Rational Rose extractor instance

The implemented Rational Rose extractor is shown in Figure 3.8. The new extractor can be added to the list of available extractors, and by choosing the run method it automatically performs the extraction of artifacts and stores them in the ATF repository. Extractors for parsing use case models from Enterprise Architect files, and

feature models from FMP files were also implemented. The instantiation mechanism follows the same principle described in this section.

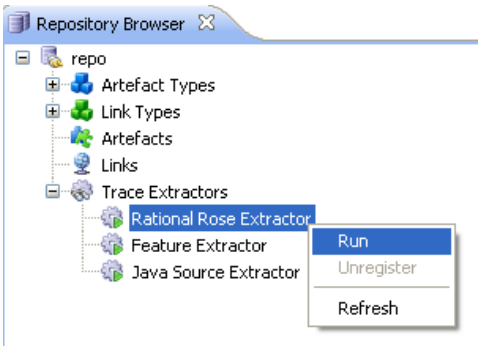


Figure 3.8 – Rational Rose extractor runtime

3.3.2 Register Instantiation

For our *TraceRegister* instance we have chosen to implement a register that allows manual definition of trace links between variability and requirements. Our register instance can be used to define new trace links, or perform maintenance operations on existing ones. As shown in Figure 3.5, the extension point used is net.ample.tracing.framework.core.traceRegister. Figure 3.9 depicts this instantiation.

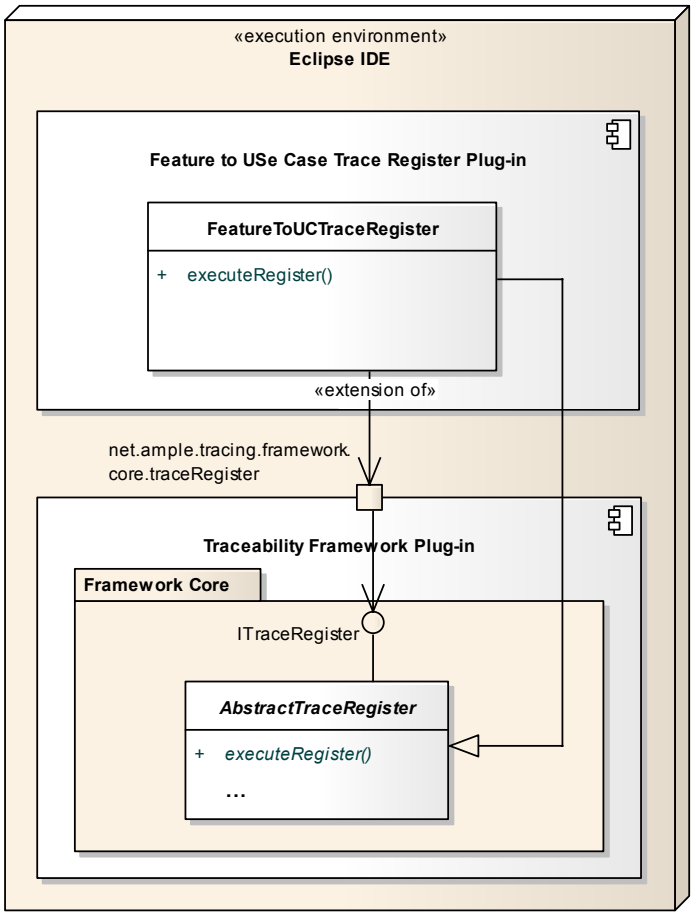


Figure 3.9 – Feature to Use Case trace register instance

The Traceability Framework provides an abstract implementation of a register (*AbstractTraceRegister*), which implements the common methods to all register instances. The only method missing is the `executeRegister()` that will be invoked when an extractor instance is selected. The instance that was implemented provides the GUI shown in Figure 3.10. This instance displays a tree of features and use case elements (e.g., use cases, use case steps, actors and packages) that were previously stored in the repository. The user can then use the checkboxes to create or remove trace links between the different elements. Once the "Save" button is pressed, the changes performed are committed to the ATF repository.

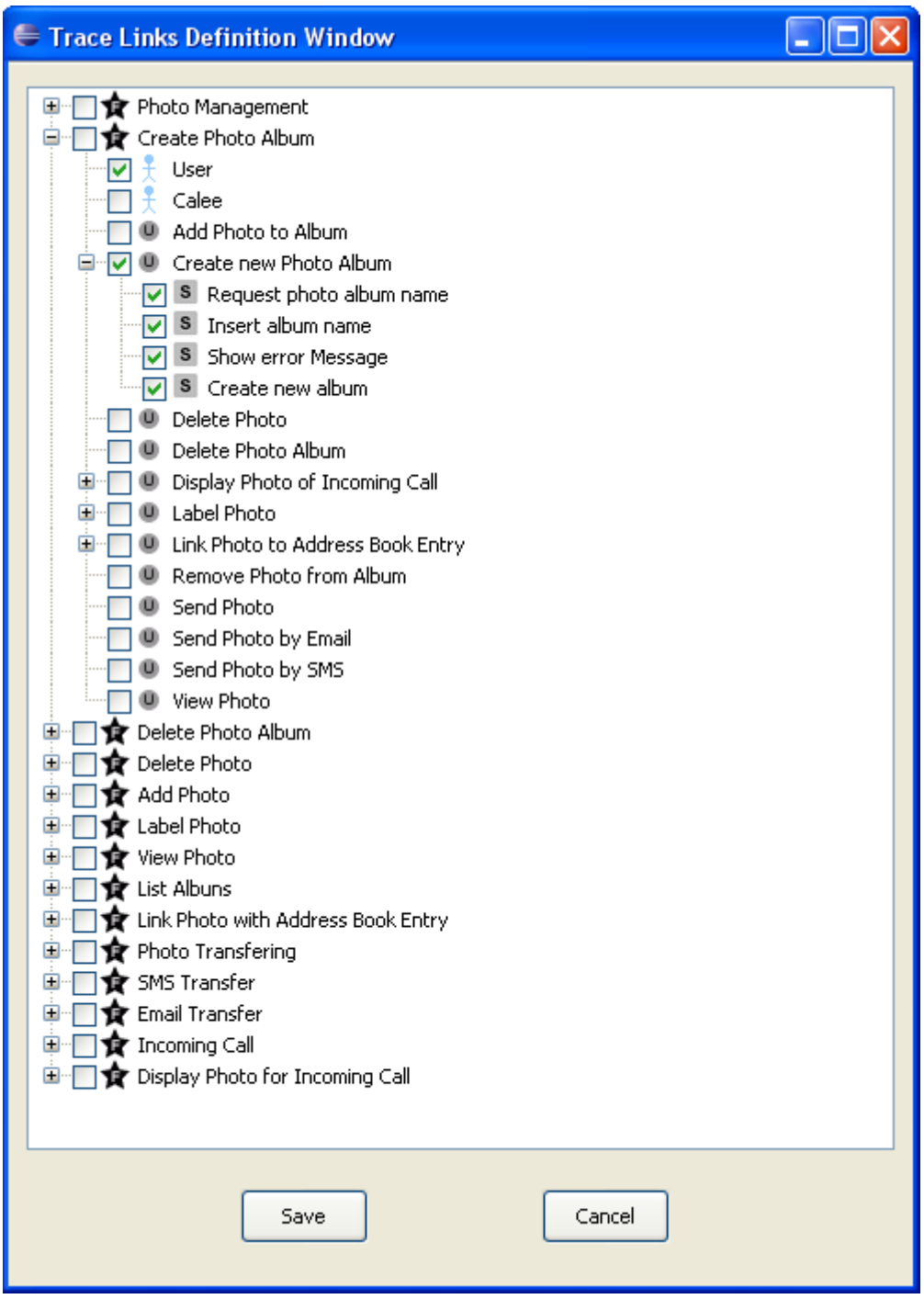


Figure 3.10 – Feature to Use Case trace register GUI

3.3.3 Query Instantiation

The first *TraceQuery* instance that was implemented is used to query which artifacts are associated with a given feature. There are plans to implement other types of queries, which will be discussed in Chapter 4. The instantiation of the trace query for related artifacts is shown in Figure 3.11. The extension point used is the net.ample.tracing.framework.core.traceQuery.

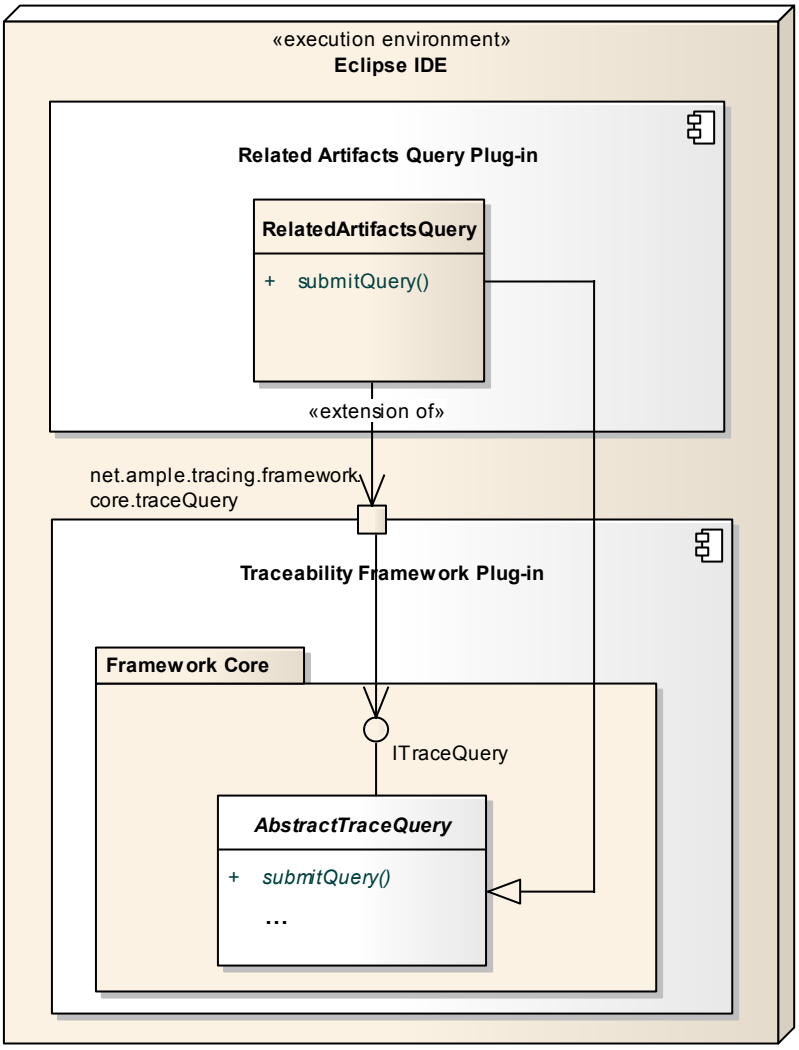


Figure 3.11 – Related artifacts query instance

The *AbstractTraceRegister* provides a base implementation for the Trace Query extension point. Instances of this hotspot can be implemented by extending this abstract class and by providing an implementation of the method `submitQuery()` which will perform the query on the ATF repository and return the set of trace links that are desired. The instance that is implemented, provides the user with an interface for selecting the features (among the features stored in the repository) that he whishes to query. Figure 3.12 shows the *RelatedArtifactsQuery* interface. After the user selects desired features and presses the submit button, the results will be displayed on the chosen view.

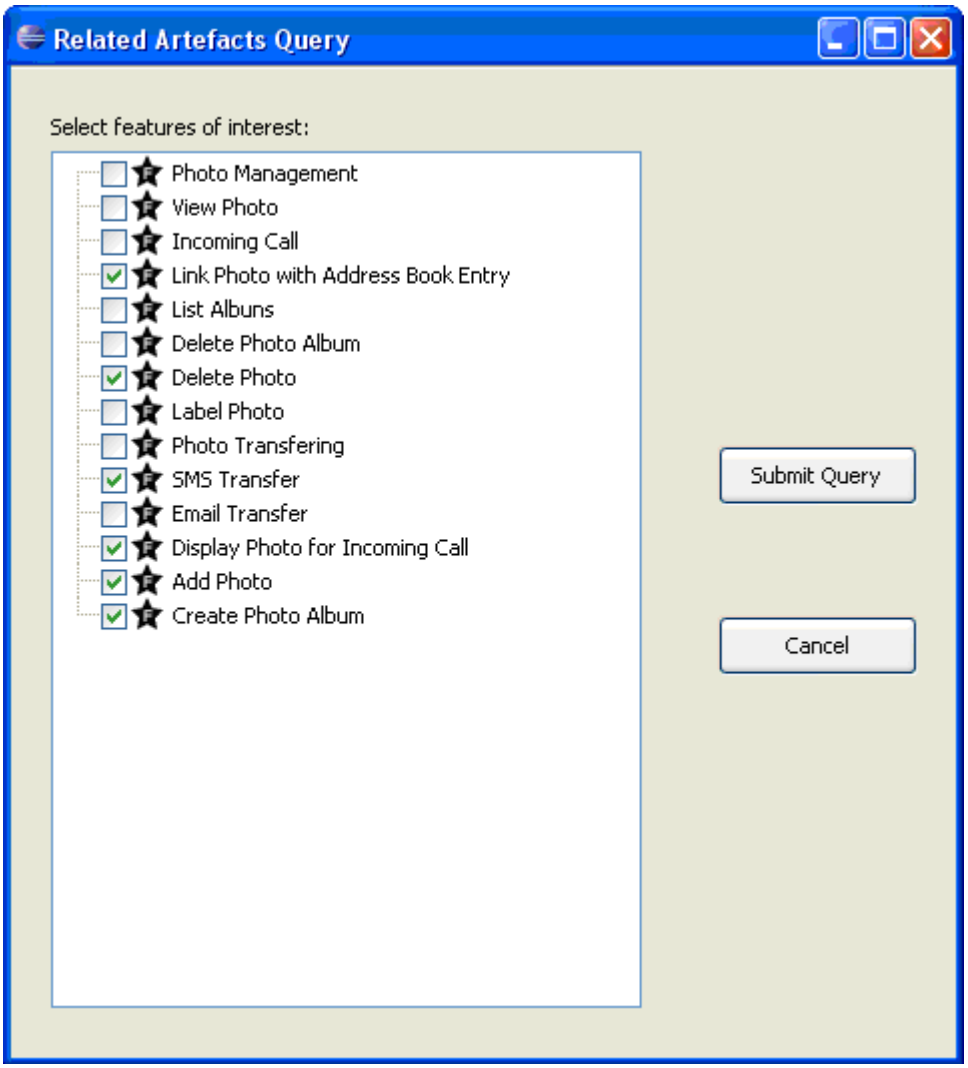


Figure 3.12 – Related artifacts query GUI

3.3.4 View Instantiation

The *TraceView* instances that were implemented display a tree view of the queried artifacts. Other types of views may be implemented in the future, as the need arises. To implement new trace views, it is necessary to extend the hotspot defined in the framework. This hotspot uses the net.ample.tracing.framework.core.traceView extension point. Figure 3.13 depicts the implementation of two trace views, both extending the same extension point and inheriting from the abstract implementation provided (*AbstractTraceView*).

The two views that are currently implemented display the results of a query using a tree. When applying this view to the *RelatedArtifactsQuery* described in the previous section, the result shown is a tree with the features in the first level, and in the second level it displays the use case elements that are linked to a feature. The *DetailedTreeView* shows the results in a more detailed manner, with the features, the use cases and the respective steps of each use case. The *TreeOverview* implementation shows a view with less detail, showing only the features and the use cases, actor and packages. These instances are shown in Figure 3.14 and Figure 3.15, respectively.

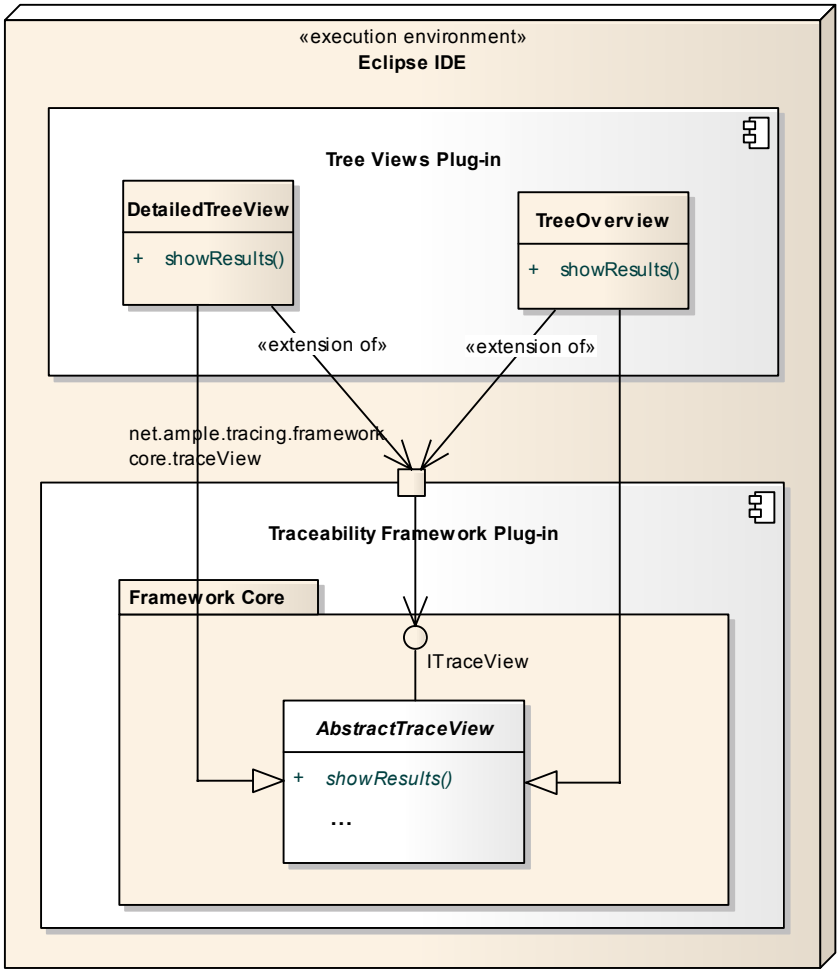


Figure 3.13 – Overview and detailed trace view instances

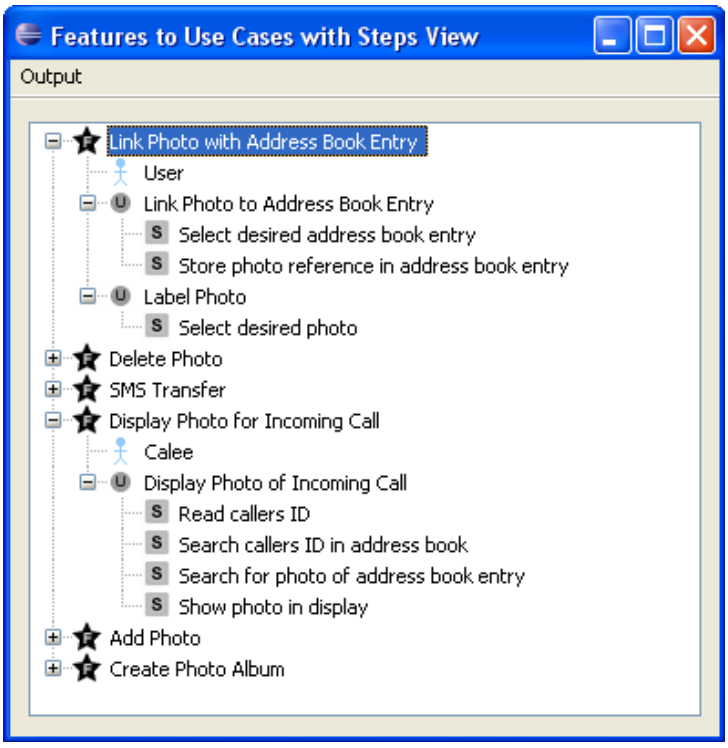


Figure 3.14 – Detailed tree view interface

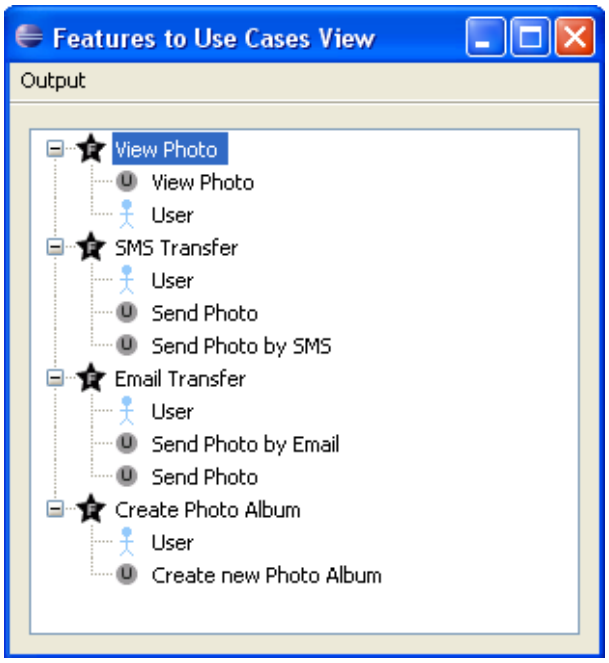


Figure 3.15 – Tree overview interface

3.4 Framework Evolution

The current implementation of our traceability framework allows users to create new framework instances to suit their specific needs. This mechanism makes our framework highly reusable. However, the instantiation mechanism may sometimes be non-trivial, requiring some expertise that fall outside the scope of the actual framework instantiation. (e.g., an instance that provides a window to the user must use the SWT[6] or Swing[7] for handling the graphical components). These mechanisms fall outside of the framework core, because some instances may not require a graphical interface (e.g., a register that creates trace links without manual intervention).

These limitations lead to a "white box" development scenario of framework instances. The developer must not only be aware of the framework instantiation, but also of Eclipse components and controls. Based on the knowledge acquired during the development of the first version, and to facilitate the process of creating new framework instances, we wish to provide generic hotspot instances for rapid and simple framework instantiation. The goal is to provide enough generic instances, so that new instances can be developed in a "black box" manner, where framework developers need only to take a generic instance, tailor it to their specific needs (e.g., choosing what type of icons to use) and abstract from the underlying architecture.

This approach is depicted in Figure 3.16. The framework core is represented by the three hotspots *TraceRegister*, *TraceQuery* and *TraceView*. On top of this sits the "White Box" layer, where developers are concerned with implementing a generic instance for one of the hotspots. For instance, the *GenericTreeRegister* could be implemented to provide a generic checkbox tree for definition of trace links. This generic instance would implement all the widgets and necessary visual controls, while remaining generic enough to be used for a features to use case model scenario or a features to class

[6] http://www.eclipse.org/swt/
[7] http://java.sun.com/javase/6/docs/technotes/guides/swing/

diagram scenario. Finally on top is the “Black Box” layer. Developers only need to choose a generic instance from the “White Box” layer and perform some minor customization (choosing the type of artifacts to trace, or the icons to be displayed, etc.), to provide a framework instantiation to a specific scenario with minimal effort. The *FeaturesUCTreeRegister* is an example of this, where the type of artifacts has been chosen to be features and use case models. The end result could be something like the window shown in Figure 3.10.

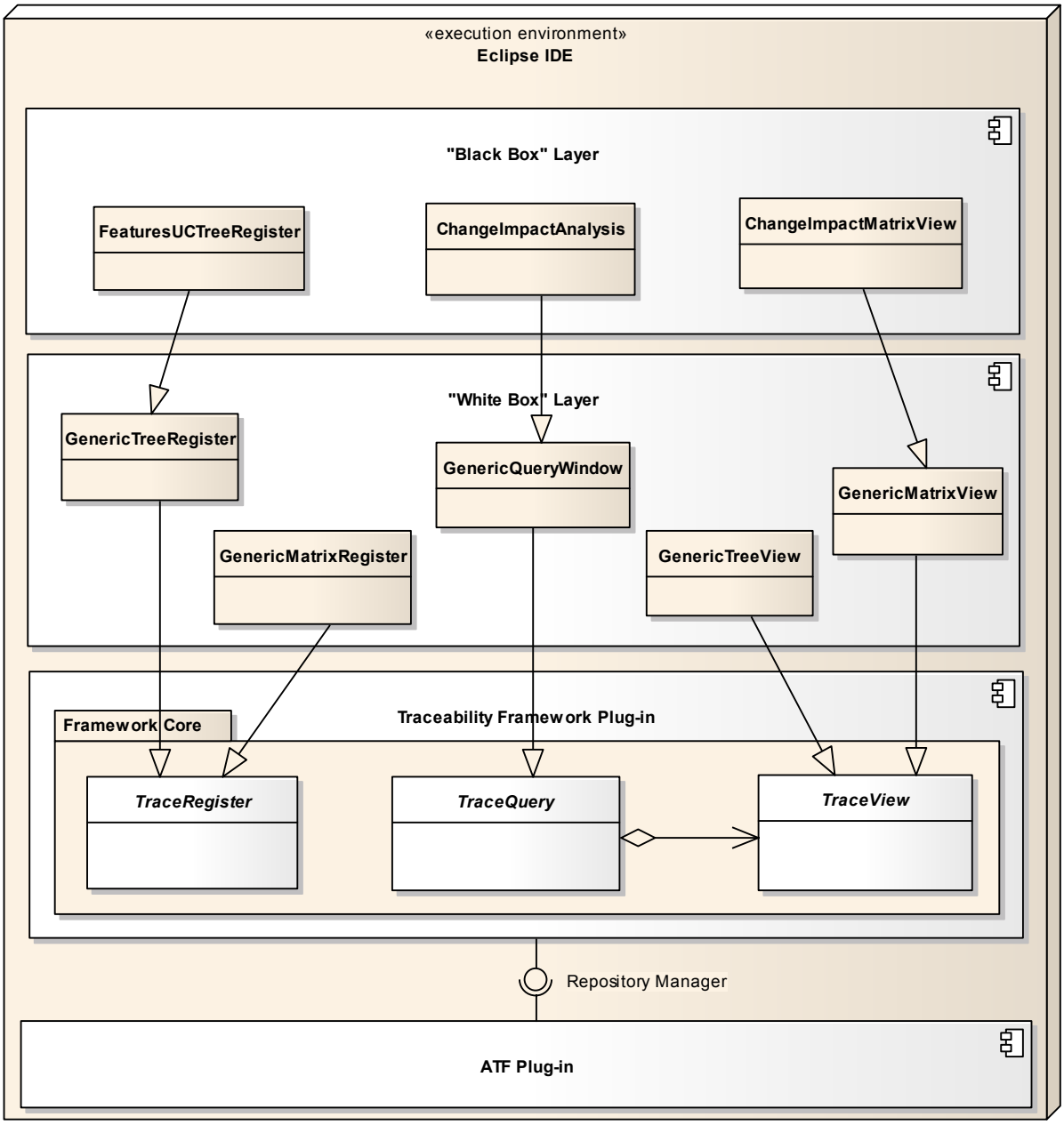


Figure 3.16 – "Black Box" framework instantiation scenario

3.5 Summary

This chapter introduced our proposal for a Model-Driven Traceability Framework. This framework addresses traceability in the context of SPL development. The solution that

we presented addresses the major shortcomings found in the approaches and tools discussed in Chapter 2: the lack of proper tool support and the need for an extensible solution that could be tailored for many scenarios. Our solution decouples the trace information from the remaining models used in software development. This keeps the models clean, goes towards the separation of concerns principle and makes the framework easier to evolve and maintain. Any kind of variability model, requirement model, architectural model, and other software elements can be used in our approach. Since the desired elements are stored in our repository as traceable artifacts we can create trace links between any two elements that we desire. The ability to define trace link types and traceable artifact types also gives a good mechanism for restricting the kind of trace information that one whishes to collect and facilitates a good filtering strategy for trace queries. The variation mechanisms built in our traceability framework provide a powerful mechanism for extensibility and evolution of our proposal. Basic users can benefit from the default implementation to establish trace links in the context of SPL. More advanced users can implement extensions to provide new types of trace registers, queries and views to suit their needs. Even though the initial goal was to design and specify a framework for product lines traceability, the achieved solution is generic enough to be usable in product lines, or instantiated to other software development scenarios (e.g., Single-Systems).

Some problems and difficulties were encountered during the design of our framework. One of the major problems found was during the implementation stages. The lack of good documentation regarding Eclipse plug-ins development was the source of many problems. Even though some tutorials and documentation exist in these topics, they are usually treated in a very simplistic manner and leave out several important aspects that proved to be necessary to achieve the current status of our traceability framework. Another problem was in defining the workflow for executing trace queries and viewing the results in a trace view. As mentioned previously, this thesis work was developed in the context of the AMPLE project, and several partners were involved in the framework specification by providing feedback for our proposal. One of the points of discussion is the current workflow for trace queries and views that was described in the previous sections. The current proposal is quite static, allowing the user to execute the chosen query, and viewing the results in the desired view. This workflow is under revision, as it seems to be more interesting to provide a more dynamic solution, allowing the user to execute a query, view the results, and submit a new query from the viewed results. This process seems to be more appealing, but introduces new challenges in the round-trip that will exist between a trace query and a trace view.

Another problem that is present in our approach is related with the need to keep the artifacts updated. As the software system evolves, the models that represent it may also evolve and suffer changes. These changes may have repercussions in the traceable artifacts and trace links that are stored in the ATF repository. It is therefore necessary to develop some strategy to handle the problems caused by changes. We have not developed such a strategy yet, but we plan to do so as future work. The major problem with the update mechanism is related with the fact that we use a separate metamodel to represent variability. In our approach the artifacts are imported into the trace repository and trace links are defined between the imported artifacts. This poses some challenges for the automatic detection of changes in the original models. It may be necessary to provide some manual mechanism to the user, so that an update event is launched.

In this chapter we have also discussed the instantiation mechanisms for each hotspot of our traceability framework. We demonstrated how to implement an instance (in the form of an Eclipse plug-in). The current version of our proposal already

facilitates these variation mechanisms. However, we do recognize that hotspot instantiation can be a complex task that comes with a great overhead attached, due to the need of implementing all the necessary controls and widgets using the graphical libraries available for the Eclipse platform. To address this issue, we plan to evolve the framework to a “black box” development scenario. In this scenario, a collection of generic instances are provided to the developers of new framework instances, so that they can abstract from the underlying Eclipse platform implementation and concentrate solely on the traceability scenario that is being implemented. We believe that it will provide a much easier way for implementing framework instances.

In the following chapter we will describe some techniques to address problems that may occur during the SPL development, and how traceability can aid developers by providing valuable information.

Chapter 4. Addressing Software Product Lines Development with Traceability

This chapter will discuss how we plan to use some techniques and heuristics to support SPL development. By performing specific analysis on existing trace artifacts and trace links we wish to detect problems that may arise during the SPL lifecycle. We will present the problem of change impact analysis and covering analysis. Even though these problems are also present in Single-System development, we discuss the particular differences that are present in SPL domain and our proposal for addressing these issues. The third problem that is discussed is the detection of feature interactions, which is more applicable to a product line environment (although not exclusively). We describe the problem of feature interaction detection and propose a solution for it by processing the trace links that exist between the different SPL artifacts. All this approaches are planed to be implemented as instances of the traceability framework described in the previous chapter.

4.1 Covering Analysis

Covering analysis in traditional Single-System development is usually established by creating trace links between requirements and the software artifacts that describe or implement them. The covering analysis process then consists of querying the available trace information to discover requirements that have not been satisfied, or artifacts that are not linked to a requirement (i.e., superfluous artifacts).

In SPL the requirements of a single product are not derived individually, but are generated from the requirements of the entire product line, by choosing which variabilities to incorporate in a product variant. In our opinion, this makes the variability

model the center of all SPL traceability. This is the reason for the choice of establishing trace links from features to all other software artifacts used in a SPL. This also introduces a small change in the covering analysis problem. Since the trace links between requirements and other artifacts no longer exist, one can no longer use that information to reason about requirements not being satisfied or unused artifacts. In our approach this analysis must be performed by querying the available trace links between features and other artifacts to detect possible features that are not satisfied (i.e., do not have a requirements specification, an implementation, etc.) or artifacts that are superfluous (i.e., artifacts that are not linked to any feature).

Figure 4.1 represents this idea for covering analysis. It shows the variability model (feature model), a requirements model (use case model), some implementation artifacts (files, Java classes, etc.) and some trace links between features and the remaining artifacts. The trace links begin in a feature and end in some artifact. By simply searching for features with no outgoing trace links we can find unsatisfied features (marked by a red circle). On the other hand, searching for artifacts with no incoming trace links yields the superfluous artifacts (marked by a blue square).

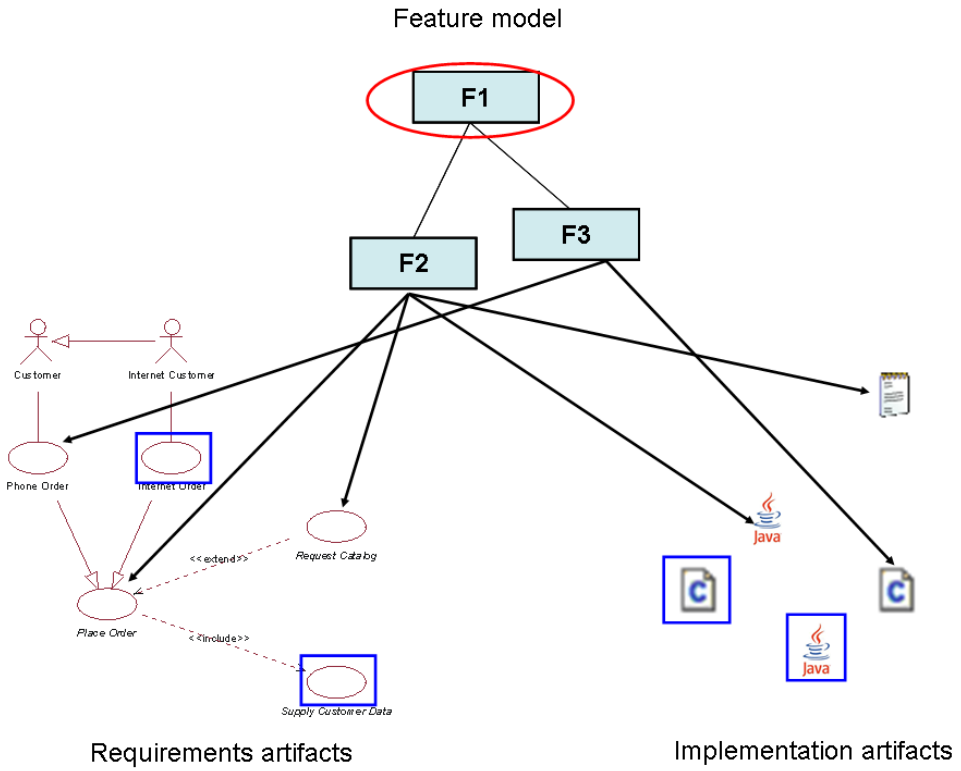


Figure 4.1 – SPL covering analysis

4.2 Change Impact Analysis

As mentioned previously, in SPL development, we propose to use the feature model as the center of all traceability information. All trace links originate from a feature and end in a certain artifact. Starting from this premise, we are proposing to perform change impact analysis in two phases. The first phase is done by querying the traceability information available in the repository. We would then search for trace links that originate in the same feature and reach two different artifacts (red links in Figure 4.2).

We can then reason that a change in one of these artifacts might trigger a change in the other artifacts, because they are linked to the same feature.

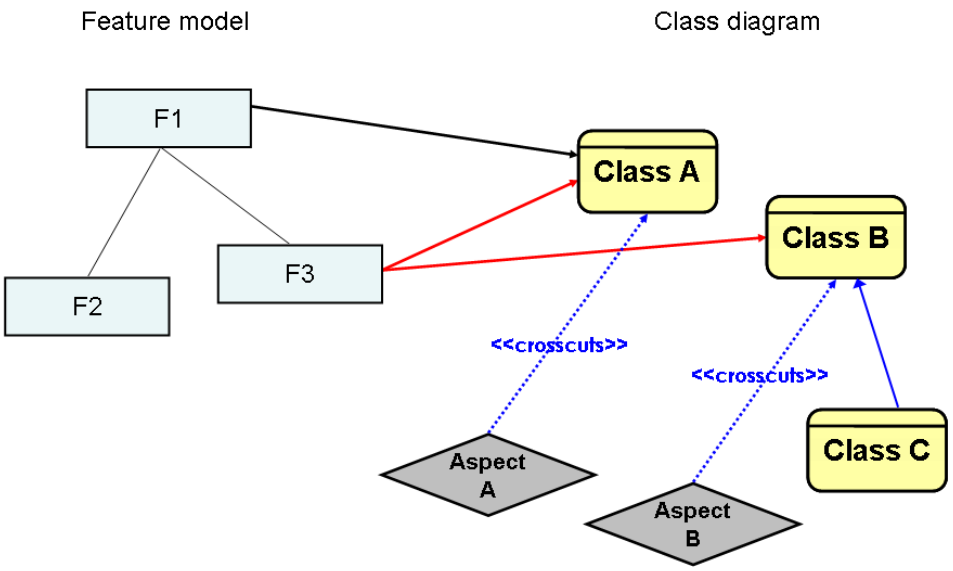


Figure 4.2 – SPL change impact analysis

On the second phase we query the source model to find additional impact caused by the change. Looking at Figure 4.2 we can see that "Class B" is inherited by "Class C" and is crosscut by "Aspect B". "Class A" is also crosscut by "Aspect A". We would then use this links to discover additional artifacts affected by a change (blue links in Figure 4.2).

4.3 Detection of Feature Interaction

Feature interactions have been defined has [72]:

…some way in which a feature or features modify or influence another feature in defining overall system behavior.

A feature interaction can be either good or bad. Good feature interactions occur when the interaction results in a desired system behavior or state. Bad feature interactions occur when the system does not behave as expected as a result of the interaction [72]. Due to the nature of SPL development, where new features can be added or removed to the existing product line, the feature interaction problem is a major concern, because unhandled feature interactions can produce undesired results in the product variants that come out of the SPL. On the other hand, detecting feature interactions is a complex task to perform manually, because feature interaction is implicit in feature composition and therefore difficult to understand. It becomes clear that feature interactions must be detected in order to be properly addressed by software developers.

The solution that we propose is to use trace links to discover feature interaction candidates. Our proposal it to detect feature interactions by discovering artifacts (a use

case, a class, etc.) that are linked to two, or more, features. This idea is illustrated in Figure 4.3. In this example, a feature model and a class diagram have been defined during some SPL development stage. Some trace links between the elements of both models have also been derived during SPL development. The search for features that are connected to the same element returns two trace links (red arrows in Figure 4.3), a candidate for the point where the feature interaction occurs (Class A) and a possible feature interaction between features "F1" and "F3".

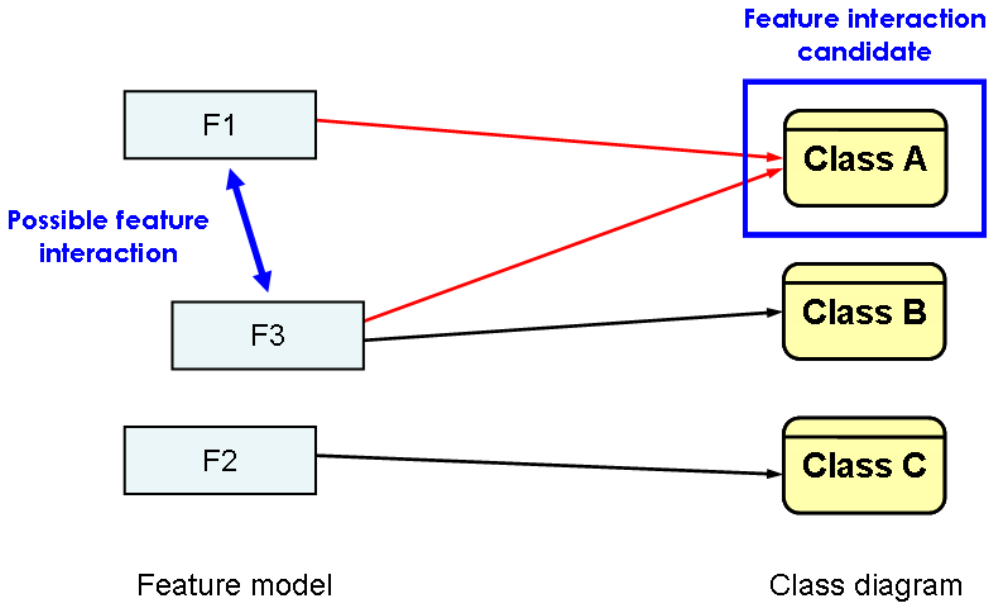


Figure 4.3 – Feature interaction detection

With the results from this analysis, the developers could then take special attention to the way that "Class A" is modeled. Also, some constraints may need to be defined between features "F1" and "F3" as the result of this interaction (e.g., feature "F1" *requires* feature "F3").

It should be pointed out that we are only detecting possible feature interactions. Detecting feature interactions with an absolute degree of certainty can be very difficult, as sometimes the information available is limited. For instance, the example shown in Figure 4.3 might not even constitute a feature interaction. Even though two features *collide* in "Class A" they may be linked to completely separate blocks of code inside the class, and thus not having any interaction whatsoever.

4.4 Summary

With the increasing complexity of software systems the need to facilitate mechanisms that reason about the quality of a system seems to become a necessity for software developers [57]. A system that fails to meet its requirements will probably be discarded by its users. One of the metrics that can be used to assess the quality of a software system can therefore be the degree to which it fulfills the requirements that were specified. Another important quality may be the absence of undesired functionalities. We demonstrated that by performing covering analysis we can determine if there are requirements that are not satisfied, or detect artifacts that are not linked to a requirement

and may therefore clobber the system without any reason for it. We have also proposed an approach to address this problem in the context of Software Product Lines. In our opinion, the techniques used in traditional (non SPL) systems cannot be applied in this domain due to the variability dimension that is not present in Single-Systems [3]. The second problem is related to the detection of how a change in an artifact reverberates over the rest of the system. As software artifacts (architectural models, code, requirements, etc.) are linked to each other, any change introduced in an artifact should be evaluated to see if it produces undesired results in the related artifacts. For instance, if the requirements change, the architecture of the system may need to be adapted to reflect that change. Discovering these relationships manually, is an error-prone and resource consuming task. As with covering analysis, this problem is not exclusive to SPL, but the variability inherent to product families requires a new kind of approach to achieve a solution. We presented a proposal that provides the visualization of the impact of changes. We believe that our proposals for covering and change impact analysis have the ability to address these two concerns in SPL development.

The third problem discussed was the detection of feature interactions. From our experience, not much literature exists on this topic and the largest source of research is the telecommunications industry, where this problem is quite common. This problem can also appear in SPL, due to the use of features to model the variabilities and commonalities of a product line. To effectively handle feature interactions, the first step that must be taken is to detect them. That is the goal of the approach that we have presented in this chapter. We believe that it could provide detection of feature interactions based on the trace links that are defined between artifacts during domain analysis. In our opinion it may be easier for domain engineers to define the trace links between the different artifacts and let tools perform the detection phase, than to detect feature interactions from scratch.

To our knowledge, the approaches presented in this chapter constitute new contributions to the existing traceability research, as no one else has tackled these problems. In addition to the theoretical proposals, we plan to implement these analysis mechanisms in our framework, by implementing the appropriate instances (trace queries and views) to provide tool support for the discussed approaches. This implementation will also serve to validate our proposals.

In the next chapter we present a case study to validate our traceability framework.

Chapter 5. Case Study

This chapter will present a case study for a home automation product line. This case study will be used to validate the framework that we propose. We plan to demonstrate the traceability framework usage and how it could be instantiated to perform detection of feature interaction in the SPL domain. The case study consists of a feature model describing the product line variability and a use cased model modeling the requirements.

5.1 Home Automation Product Line

The home automation product line[8] that is used in this section is meant to provide management functionalities to the home owner. The feature model shown in Figure 5.1 shows the several variants that can be generated from the product line core assets.

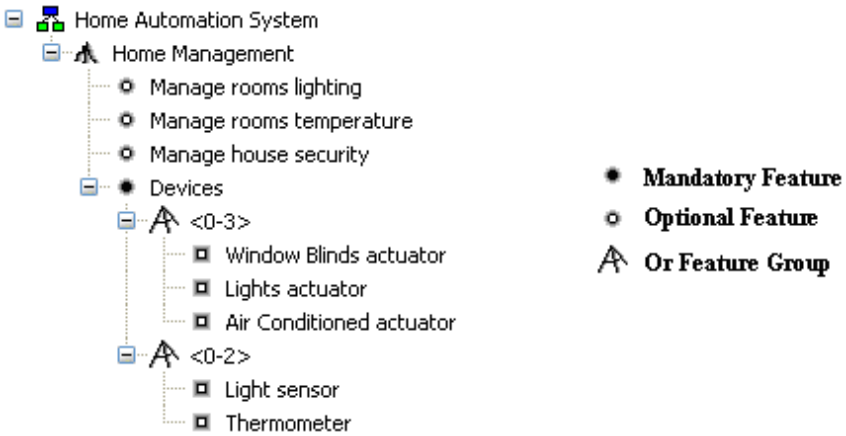


Figure 5.1 – Feature model for a home automation system

[8] Based on the "Smart Home" case study provided by Siemens in the context of the AMPLE project.

A Home Management product is composed of several optional features: “Manage rooms lighting”, “Manage rooms temperature” and “Manage house security (the names are self explanatory). It is also composed of several devices that are used to automate the required functionalities. Two types of sensors exist, one for light detection and another to determine the room temperature. Finally, a selection of actuators is also available, each responsible for providing a certain functionality.

The use case model shown in Figure 5.2 displays the requirements of this home automation system. This use case is modeling the entire product line, and not an individual product. It contains an actor “Home Owner” to represent the person in charge of managing configurations of the system. The remaining actors represent the devices that are available for selection.

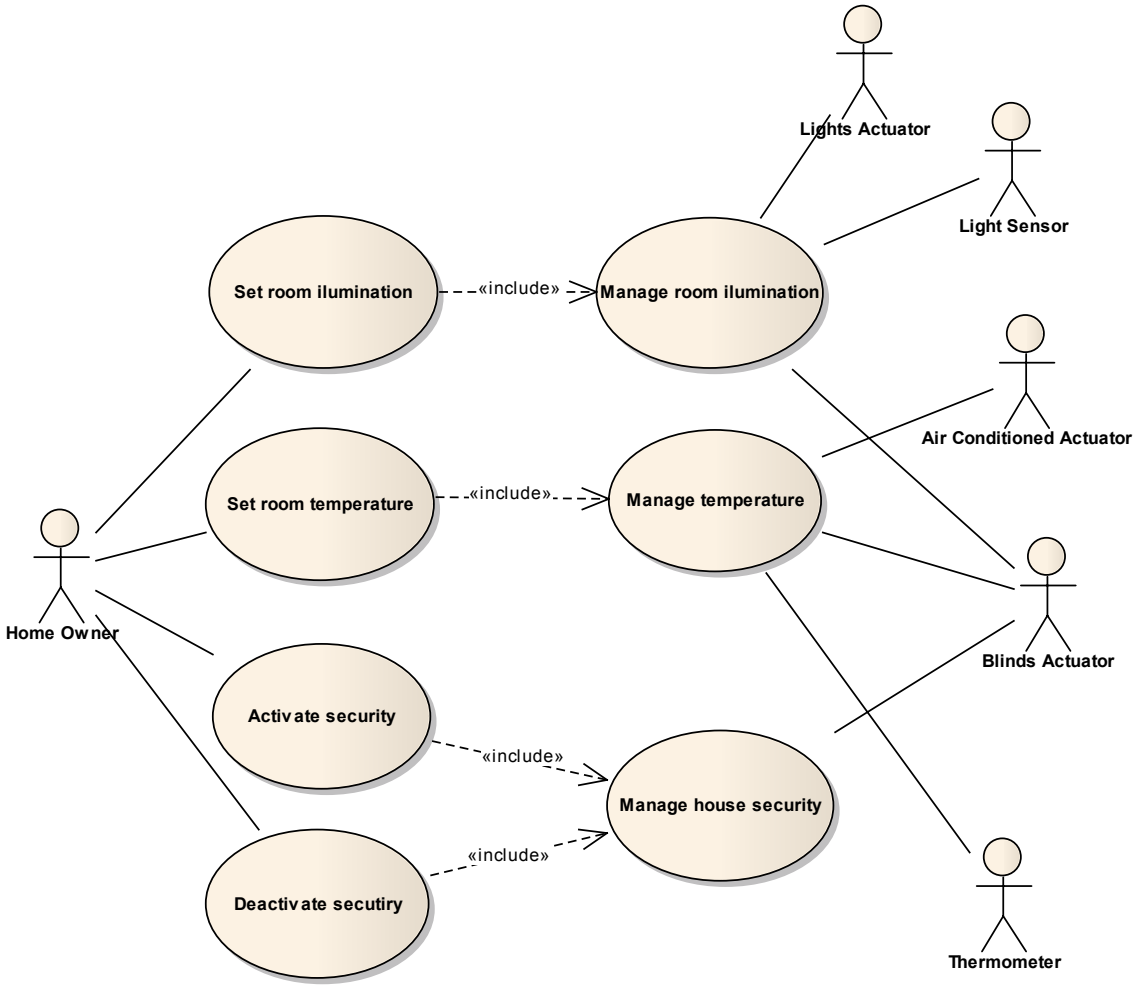


Figure 5.2 – Use case model of a home automation system

The system allows a home owner to define the temperature to be maintained in a room. A thermometer is used to measure the temperature and the air conditioned and blinds actuators are used to regulate the temperature (e.g., closing the blinds to prevent the sun from heating the room). The light of the room can also be defined by the user which will them be automatically maintained by the system. A light sensor is used to measure the amount of light and the blinds and light actuators can be used to regulate the light to the appropriate level (e.g., opening the blinds to allow more sun light to get

in). The security management on the other hand, is defined to close the blinds in certain conditions (e.g., close the blinds after 8 PM).

5.2 Framework Usage

To provide traceability support for this home automation product line, we could use the traceability framework that we proposed. The framework would enable the creation of trace links between the elements of the feature model and the elements of the use case model. The artifacts could be imported into the ATF repository by executing the corresponding extractors that were described in Chapter 3. In Section 5.2.1 we describe the trace links that are defined during domain analysis and Section 5.2.2 describes a framework instance that would perform the detection of possible feature interactions, according to the strategy discussed in Chapter 4.

5.2.1 Defining Trace Links

The trace links that have been identified are represented in Table 5.1. The feature model elements are represented in the columns, while the use case model elements are represented in the rows.

Table 5.1 – Trace links for the home automation system

UC Model \ Feature Model	Manage Rooms Lighting	Manage Rooms Temperature	Manage House Security	Windows Blinds Actuator	Lights Actuator	Air Conditioned Actuator	Light Sensor	Thermometer
Home Owner	X	X	X					
Set Room Ilumination	X							
Set Room Temperature		X						
Activate Security			X					
Deactivate Security			X					
Manage Room Ilumination	X							
Manage Temperature		X						
Manage House Security			X					
Light Sensor	X						X	
Thermometer		X						X
Lights Actuator	X				X			
Air Conditioned Actuator		X				X		
Blinds Actuator	X	X	X	X				

A mark in one of the cells means that a trace link between the feature model and use case model elements has been identified. The definition of these relationships is one

of the problems in traceability. These trace links are usually defined by domain engineers which possess a great deal of knowledge in the domain that is being modeled. Some trace links might be derived automatically, as described in some of the approaches discussed in Section 2.2.1, but in a product lines development scenario we whish to link elements from different dimensions and these relationships are not always obvious and easy to automatically deduct. The validation of these trace links is another concern in which traceability researchers dwell. Even the automatic generation of trace links (e.g., links generated during a model transformation) is seen with some skepticism, as no formal validation mechanism is usually provided. To our knowledge, this validation problem remains largely untackled and a reasonable solution does not exist.

The representation shown in Table 5.1 is used in this document to enhance the ratability of this example, and it works because the number of artifacts is relatively small. In practice traceability matrixes do not scale well, and become unusable if the number of artifacts is too big. For instance, based on our experience a SPL system with twenty features and an equal number of requirements would be quite hard to represent using a traceability matrix. In reality the trace register instance that was implemented uses a tree of checkboxes to allow the manual definition of trace links between features and use cases models. We believe that a tree representation is much more scalable, as it allows a user to expand only the desired subtree and concentrate only on a smaller portion of the artifacts. An example of this register execution is shown in Figure 5.3.

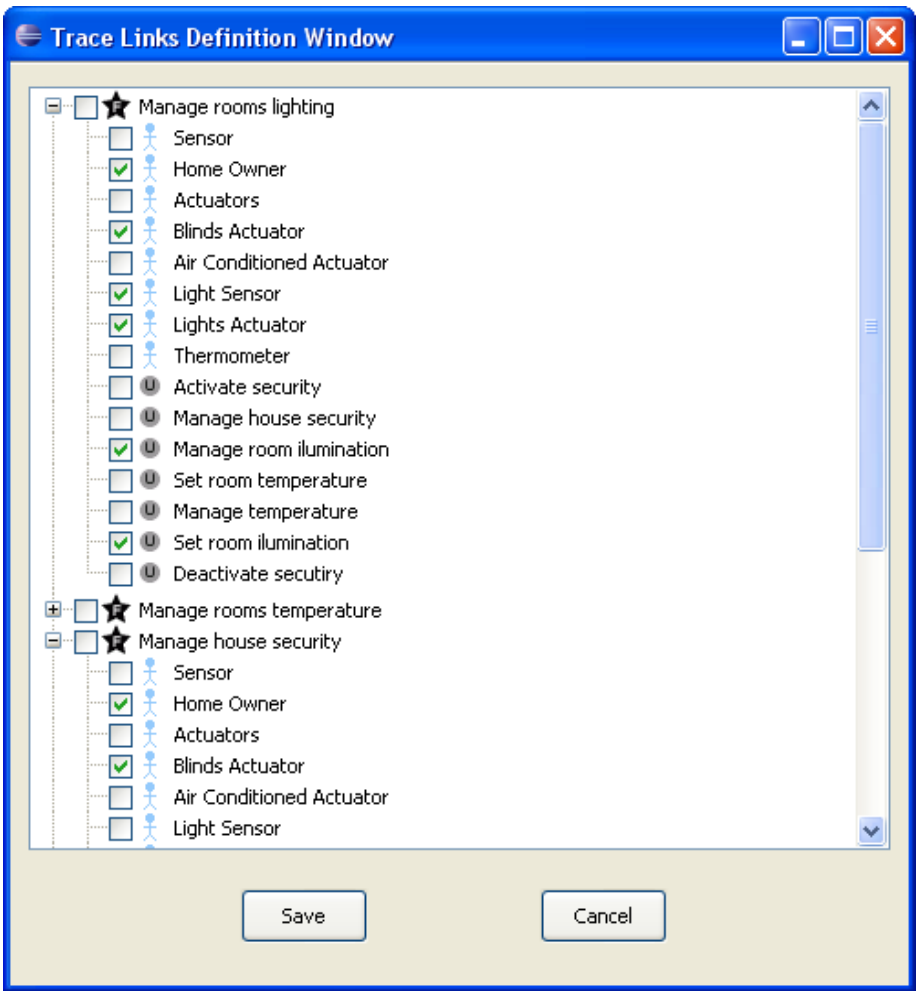


Figure 5.3 – Trace register execution for the home automation system

5.2.2 Detecting Feature Interaction

Once the trace artifacts and respective trace links are stored in the repository, we can perform queries and analysis on that information. Our goal is to implement a trace query instance that performs the feature interaction analysis, returning the list of feature interaction candidates found. An example of how this analysis is performed is shown in Figure 5.4. To enhance the understanding of this example, only a subset of all the trace links is shown.

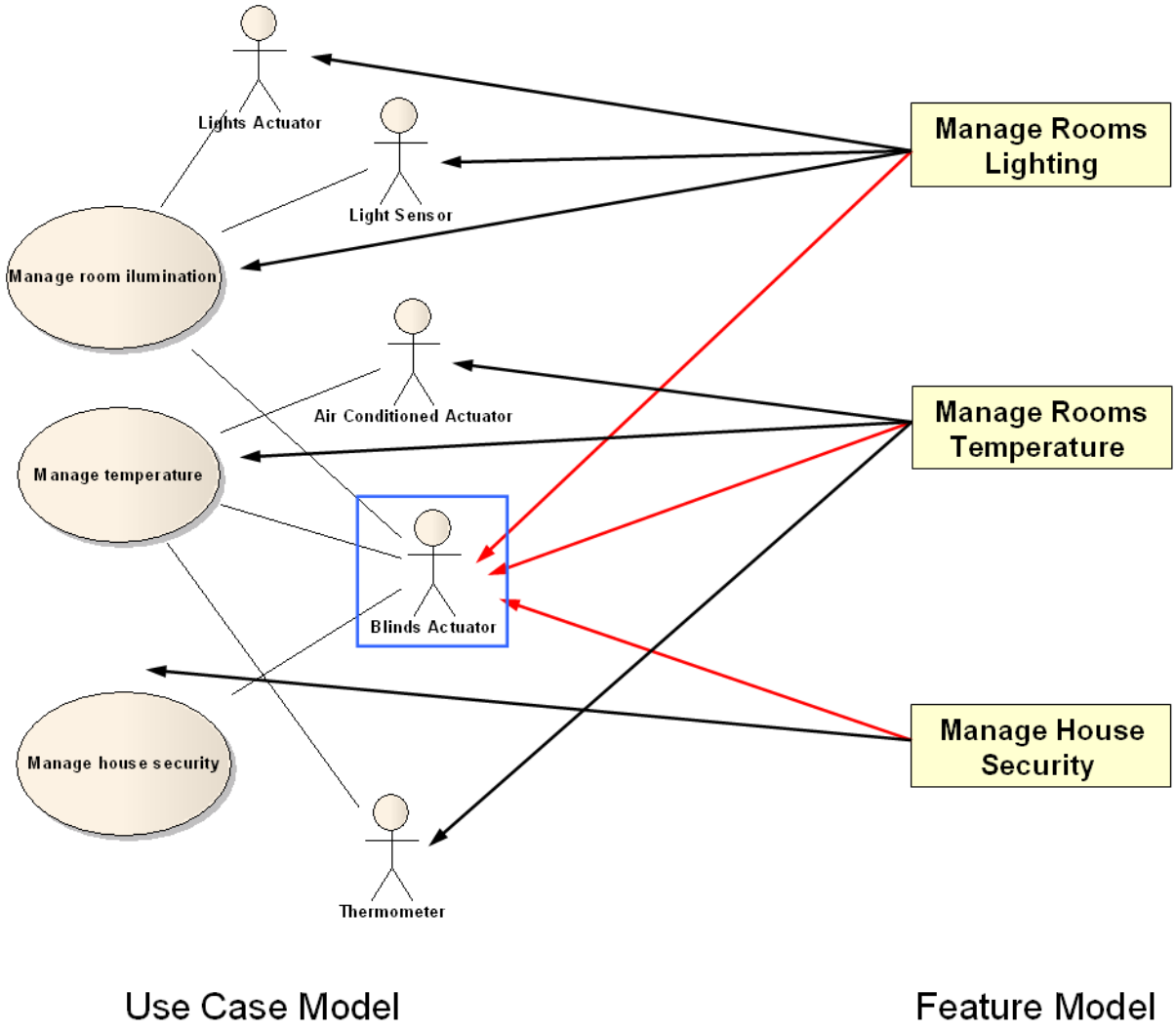


Figure 5.4 – Feature interaction in home automation system

By querying the trace information for links that have different source features and the same destination artifact, we can detect possible feature interactions. This example is demonstrated by the red arrows in Figure 5.4, which originate from three different features (*Manage Rooms Lighting*, *Manage Rooms Temperature* and *Manage House Security*), and arrive at the same actor (*Blinds Actuator*). In fact, this does constitute a feature interaction example. While the *Manage Rooms Lighting* feature might try to open the blinds to let natural light come in, the *Manage Rooms Temperature* feature may want to close the blinds to cool the room down. Some tradeoff must be made to properly handle this conflict. Our contribution is not meant to address the handling of

these conflicts, but rather to provide a means of detecting them and provide this information to the system developer.

This case study constitutes a clear example of a limitation found in existing approaches and traceability tools, which we are planning to address with our traceability framework. This instantiation scenario for detection of feature interactions is not yet implemented. We plan to create the necessary trace query and trace view instances to implement this type of analysis.

5.2.3 Implementing a Feature Interaction Instance

As mentioned in the beginning of the chapter, our idea is to instantiate our traceability framework to provide an implementation of the solution presented in the previous section. This instantiation is shown in Figure 5.5. The *FeatureInteractionQuery* is an extension of the net.ample.tracing.framework.core.traceQuery extension point. This class implements the algorithm described in the previous section and returns the set of relevant trace links to be passed for the corresponding trace view. The *FeatureInteractionTraceView* is used to provide an instance capable of presenting the results of the feature interaction query. For instance, the visualization could be a simple list of the feature interactions that were detected, or a graph view of the query.

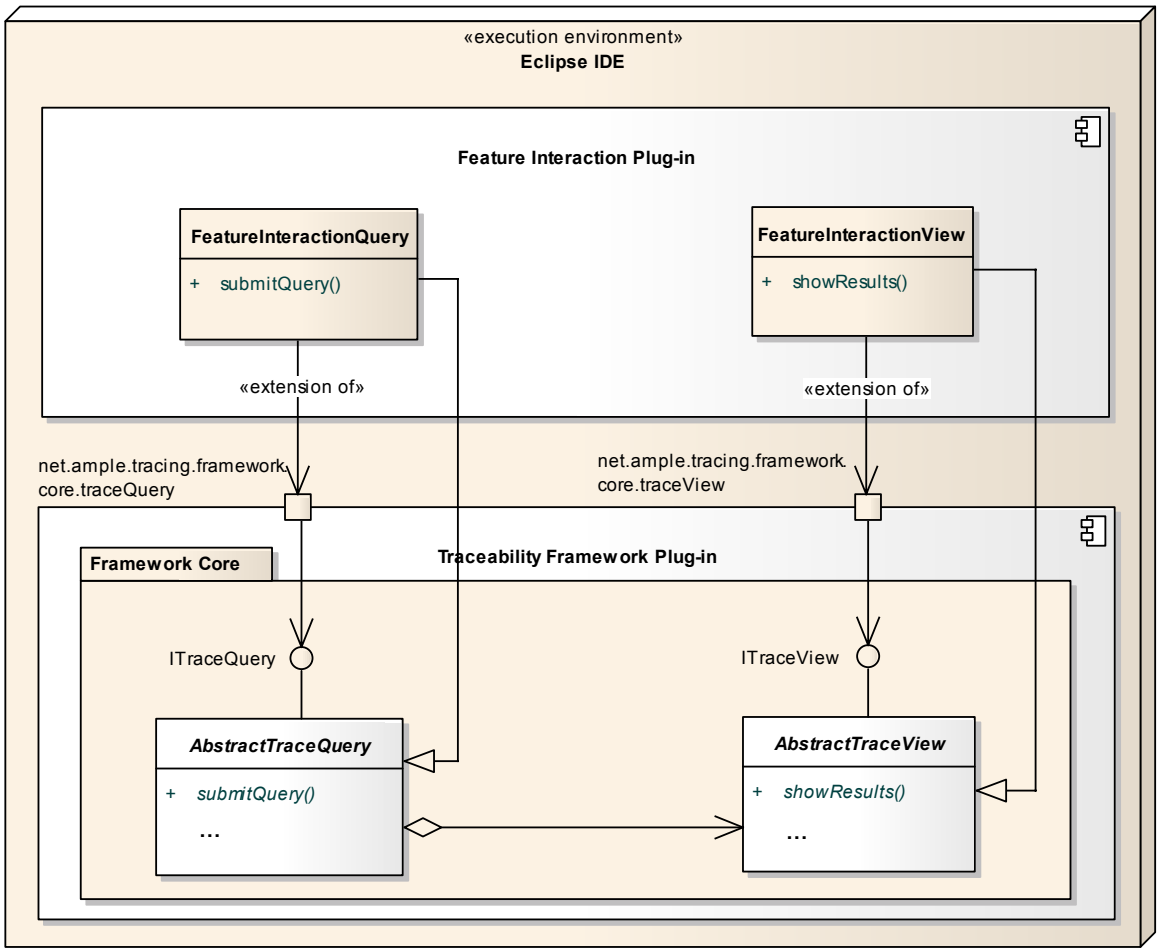


Figure 5.5 – Feature interaction detection instance

5.3 Comparison of Results

In Chapter 2 we presented some of the existing approaches for traceability support in MDE and SPL development and discussed their strengths and weaknesses. We have also presented the traceability tools survey and discussed the benefits and shortcomings of the tools analyzed. We will now compare our approach with the remainder of the SPL approaches. The MDE approaches will not be presented, because even though our traceability framework uses MDE techniques, and could be extended to facilitate traceability support for MDE, our main goal was to provide a platform for definition, maintenance, query and visualization of trace information in the context of SPL. We will also compare our implementation with the traceability tools that were presented in that chapter and discuss the improvements and benefits that we have introduced.

Table 5.2 summarizes the results of the SPL approaches discussed in Section 2.3.1. Our approach has been added to the last line of the table to facilitate the comparison. The criterion used here are the same as the ones used previously.

Table 5.2 – Comparison of SPL approaches with Traceability Framework

Approach	Representation	Mapping and Granularity	Scalability	Change impact analysis	Tool support
Pohl et al. [59]	Directed link	Backward and forward, arbitrary granularity	yes	yes	no
Berg et al. [8]	Directed link	Fine-grained	not discussed	not discussed	no
Ajila and Kaba [2]	Directed link	Coarse-grained	not discussed	yes	ad-hoc tool set
Knauber and Schneider [47]	Directed link	Forward, coarse-grained	not discussed	not discussed	Junit and AspectJ
Mergel et al. [52]	Directed link and meta information	Backward and forward coarse-grained	yes	not discussed	not publicly available
Jirapanthong and Zisman [40]	Directed link	Backward and forward, coarse-grained	not discussed	not discussed	prototype (XtraQue)
Zisman et al. [73]	Directed link and meta information	Backward and forward, fine-grained for requirements	not discussed	not discussed	prototype
Luttikhuizen et al. [51]	Directed link and meta information	Backward and forward Fine-grained for requirements	not discussed	yes	no
Bayer and Widen [7]	Directed link	Backward and forward, arbitrary granularity	not discussed	not discussed	ad-hoc tool set
Moon et al. [53]	Directed link	Backward and forward, arbitrary granularity	yes	not discussed	not discussed
Traceability Framework	Directed link	Backward and forward, arbitrary granularity	yes	yes	yes

As can be seen from the previous table, our framework proposal provides many of the benefits that are found in the best SPL approaches. We have also been able to respond to the tool support criteria. In our opinion, this is one of the major requirements for an efficient traceability solution, a requirement that is missing from many

approaches. In Table 5.3 we present the summary of the traceability tools survey presented in Section 2.4. The implementation of our traceability framework has been added to the last line of the table to facilitate the comparison. The criterion used here are the same as the ones used in that chapter.

Table 5.3 – Comparison of existing traceability tools with Traceability Framework

Tool	Creation of Trace Links	Mapping	Type of Query	Change Impact Analysis	Covering Analysis	Views	Extensible	Support for SPL and MDE
RequisitePro [37]	manual	part of artifacts, forward and backward	filters	yes	no	matrix, tree	no	no
Borland CaliberRM [13]	manual	requirements to artifacts, forward and backward	filters	yes	no	matrix, graph	yes	no
RaQuest [65]	manual, imports requirements from file	requirements to artifacts, forward and backward	n.d.	yes	no	matrix, graph	no	partial support for MDE
Telelogic DOORS [69]	manual, imports requirements from file	requirements to parts of artifacts, forward and backward	custom queries, filters	yes	yes	tree	yes	no
Contour [39]	manual, imports artifacts from file	artifact to artifact, forward and backward	filters	yes	no	matrix, report	yes	no
GatherSpace [30]	manual	requirement to other artifacts, forward	n.d.	no	yes	report	no	no
Trace Analyzer [23]	automatic	models to source code, models to models, forward	n.d.	no	yes	matrix, graph, report	no	no
Traceability Framework	automatic, manual, import artifacts from models	arbitrary granularity, forward and backward	custom queries	yes (planed)	yes (planed)	custom views	yes	yes

The previous table demonstrates the benefits that our framework provides over the remaining tools. We have provided a solution that allows for automatic, manual or semi-automatic definition of trace links, execution of custom queries, visualization of query results in custom views and an arbitrary level of granularity in trace links. We have also presented our ideas for implementing analysis mechanisms to perform covering analysis, change impact analysis and feature interaction detection in the context of SPL development (the implementation of this analysis is not available yet, but it is planed to be provided in the next version of the tool). These types of queries are missing in all of the surveyed tools. Finally we proposed a truly open and extensible solution for traceability, allowing other developers to extend the framework capabilities in terms of registers, queries and views, and adapting it to their specific needs. We believe that these instantiation mechanisms make our proposal highly adaptable and reusable to satisfy SPL and other software development scenarios.

5.4 Summary

This chapter presented a case study based on a product line of home automation systems. The case study was composed of a variability model (feature model) and a requirements model (use case model). We validated our traceability framework proposal with this case study, by demonstrating how to address a problem that exists in SPL development and that, to our knowledge, is not supported by any other tool or approach. The case study process consisted of importing the feature and requirements model into the ATF repository. Once that step was concluded, the domain engineer would define the trace links that were identified between the elements from these two models. This step is the one that requires the greatest amount of work from the developer, as it requires the user to identify the trace links manually. We are not aware of the possibility to perform automatic identification of these links, because this information is very domain specific and will probably require a great deal of knowledge in the domain that is being modeled. The following steps are automatic. The user must simply execute the trace query instance that implements feature interaction detection and the results are reported back.

As mentioned in the previous chapter, we can only detect **possible** feature interactions. Detecting feature interactions with an absolute degree of certainty can be very difficult, if not impossible, as the trace information that is available may be limited. Because our framework allows an arbitrary granularity for the traceable artifacts, even though the same artifact may be linked, to different features, there may not exist a feature interaction between them. For instance, if we link two features to a use case, our approach detects a feature interaction between those two features. However, if we further specify the use case by defining its steps, and instead link the features to the steps of the use case, the previous feature interaction may not exist anymore, if the features are no longer related in the same steps. From this example we can see that fine-grained traceability yields much better results than coarse-grained traceability. Never the less, we can only provide feature interaction candidates, because the level of granularity to be used in our framework is a decision of the developer.

The last section of the chapter also presented a comparison between the work developed in this thesis and the SPL approaches and traceability tools surveyed in Chapter 2. We demonstrated the benefits that our framework achieves, in terms of the evaluation criterion, over the remaining solutions. We have proposed a solution that is highly adaptable and extensible, provides tool support and tackles the concrete challenges introduced by SPL development.

In the next chapter we will present the conclusions of this thesis, along with the contributions and the future work that is planed.

Chapter 6. Conclusion

The focus of this master thesis dissertation is traceability. We discussed how traceability can provide a valuable aid in addressing some of the problems associated with software development. A special focus was given on traceability in the context of SPL, where the approaches and tools for Single-System development do not cope well with the new challenges presented by this paradigm. To address this gap, we presented a proposal for a Model-Driven traceability framework that aims to provide a platform for design and implementation of traceability mechanisms and tools for SPL. This framework uses a traceability metamodel for storage of trace information and three hotspots that allow instantiating it to serve different SPL needs. The first hotspot is used to create different trace registers, used for performing CRUD operations in the trace repository. The second hotspot is used to implement different trace queries to perform analysis in the stored trace information. The last hotspot allows framework developers to implement distinct views for visualization of query results.

To validate our ideas we have implemented a version of this framework. The implementation consists of the framework core with the hotspots. The framework was implemented as an Eclipse plug-in with several extension points defined. Each hotspot instance is implemented by extending the desired extension point. Included in this first implementation, are instances of each hotspot to allow the definition, query and visualization of traceability information between features and requirements artifacts.

Some proposals for addressing SPL development problems using traceability were also discussed. We presented our ideas on how to use traceability to perform covering analysis and change impact analysis in the context of product lines. A proposal for detection of feature interactions, a complex problem that exists in the SPL domain, was also presented. We plan to implement our ideas in the following versions of our framework.

Finally, we have presented a case study for a SPL system based in a home automation product line. This example was composed of a variability model and the requirements model for the entire SPL. We demonstrated how our solution can be used to store the trace links between the elements of these different domains. We have also

shown how the framework can be instantiated to solve problems that are found in SPL development and that are not addressed by other approaches.

6.1 Contributions

The following contributions are direct results of this work:

- **A Model-Driven Traceability Framework** (Chapter 3). Specification of a framework for the definition and implementation of methods and tools for traceability in the context of Software Product Lines.

- **An implementation of the Framework Core**. The core of the framework was implemented in the form of an Eclipse plug-in with a set of extension points that allow instantiating to other scenarios.

- **Framework instances to trace from features to requirements** (Chapter 3). The base implementation that is provided allows users to import requirements (modeled with use case models) and variability models (modeled with feature models) and trace links between the artifacts of these separate domains and perform queries on this information.

- **A proposal for addressing SPL development problems through traceability** (Chapter 4). A proposal for detection of feature interactions was presented. Methods for performing covering analysis and change impact analysis in the context of SPL were also discussed.

6.2 Future Work

Much research is still under development and may be addressed as future work.

- **Implementation of new framework instances**. The detection of feature interaction, covering analysis and change impact analysis are thought to be implemented as instances of the *TraceQuery* and *TraceView* hotspots.

- **Refactoring the framework UI**. The framework interface will be revised and will be subjected to a major transformation. New functionalities to be included are under revision.

- **Evolve to "black-box" instantiation scenario**. Implement generic hotspot instances that allow developers to provide new instances with minimal effort.

- **Framework evolution.** The framework will be used and extended by the partners of the AMPLE project. Suggestions for improvement and necessary adaptations will be analyzed and implemented in future versions.

- **Handle system evolution.** As a software system evolves, its models might change. These changes must trigger updates in the repository. We plan to develop some strategy for performing this task.

References

[1] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni, "Model Traceability", *IBM Systems Journal*, vol. 45, pp. 515-526, 2006.

[2] S. Ajila and B. A. Kaba, "Using Traceability Mechanisms to Support Software Product Line Evolution", in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration (IEEE IRI-2004)*, Las Vegas, Nevada, USA, 8-10 Nov. 2004, pp. 157-162.

[3] N. Anquetil, B. Grammel, I. Galvão, J. Noppen, S. S. Khan, H. Arboleda, A. Rashid, and A. Garcia, "Traceability for Model Driven, Software Product Line Engineering", presented at 4th ECMDA Traceability Workshop, Berlin, Germany, 2008.

[4] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature Modeling Plug-In for Eclipse", in *Proceedings of the Workshop on Eclipse Technology eXchange (OOPSLA 2004)*, Vancouver, BC, Canada, ACM, October 24-28, 2004, pp. 67-72.

[5] AOSD Steering Committee, "Aspect-Oriented Software Development Community & Conference", http://aosd.net/.

[6] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud, "PuLSE: A Methodology to Develop Software Product Lines", in *Proceedings of the 1999 symposium on Software reusability*, Los Angeles, California, USA, ACM Press, 1999, pp. 122-131.

[7] J. Bayer and T. Widen, "Introducing Traceability to Product Lines", presented at the 4th International Workshop on Software Product-Family Engineering, Bilbao, Spain, 2002.

[8] K. Berg, J. Bishop, and D. Muthig, "Tracing Software Product Line Variability – From Problem to Solution Space", in *Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT '05)*, White River, South Africa, South African Institute for Computer Scientists and Information Technologists, 2005, pp. 182-191.

[9] D. Beuche and M. Dalgarno, "Software Product Line Engineering with Feature Models", *Methods & Tools*, vol. 14, pp. 42, 2006.

[10] J. Bézivin, "On the Unification Power of Models", *Software and Systems Modeling*, vol. 4, pp. 171-188, 2005.

[11] BigLever Software Inc., "Software Product Lines - BigLever Software", http://www.biglever.com/.

[12] A. Bolour, "Notes on the Eclipse Plug-in Architecture", http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.

[13] Borland, "Borland® CaliberRM™", http://www.borland.com/us/products/caliber/rm.html.

[14] A. Bragança and R. J. Machado, "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines", in *Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, IEEE Computer Society, 10-14 Sept. 2007, pp. 3-12.

[15] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*: Pearson Education, 2003.
[16] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-Based Traceability for Managing Evolutionary Change", *IEEE Transactions on Software Engineering*, vol. 29, pp. 796 - 810, 2003.
[17] J. Cleland-Huang and D. Schmelzer, "Dynamically Tracing Non-Functional Requirements through Design Pattern Invariants", presented at 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'03), Montreal, Canada, 2003.
[18] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina, "Goal-Centric Traceability Managing Non-Functional Requirements", in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, USA, ACM, 15-21 May, 2005, pp. 362-371.
[19] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*, 1st ed., Boston, MA, USA: Addison-Wesley, 2002.
[20] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants", in *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, Tallinn, Estonia, Springer, September, 2005, pp. 422-437.
[21] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Using Feature Models", in *Proceedings of the 3rd International Software Product Line Conference (SPLC 2004)*, Boston, MA, USA, Springer, August 30-September 2, 2004, pp. 266-283.
[22] A. M. Davis, *Software Requirements: Objects, Functions and States*, Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.
[23] A. Egyed, "Trace Analyzer WebPage", http://www.alexander-egyed.com/tools/trace_analyzer_tool.html.
[24] A. Egyed, "A Scenario-Driven Approach to Trace Dependency Analysis", *IEEE Transactions on Software Engineering*, vol. 29, pp. 116-132, 2003.
[25] A. Egyed, "Resolving Uncertainties during Trace Analysis", in *Proceedings of the 12th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, Newport Beach, California, USA, ACM, November 2004, pp. 3-12.
[26] M. Eriksson, J. Börstler, and K. Borg, "The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations", in *Software Product Lines, 9th International Conference*, Rennes, France, Springer, 2005, pp. 33-44.
[27] J.-R. Falleri, M. Huchard, and C. Nebut, "Towards a Traceability Framework for Model Transformations in Kermeta", presented at the 2nd ECMDA Traceability Workshop (ECMDA-TW), Bilbao, Spain, 2006.
[28] I. Galvão and A. Goknil, "Survey of Traceability Approaches in Model-Driven Engineering", in *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, Annapolis, Maryland, USA, IEEE Computer Society, 15-19 Oct. 2007, pp. 313.
[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
[30] GatherSpace.com, "GatherSpace", http://www.gatherspace.com/.
[31] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*: Addison-Wesley, 2004.

[32] O. C. Z. Gotel and A. C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem", in *Proceedings of the First International Conference on Requirements Engineering*, Colorado Springs, CO, USA, IEEE Computer Society Press, 18-22 Apr. 1994, pp. 94-101.
[33] Graphviz, "Graph Visualization Software", http://www.graphviz.org/.
[34] J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Indianapolis, IN, USA: Wiley, 2004.
[35] M. L. Griss, J. Favaro, and M. d' Alessandro, "Integrating Feature Modeling with the RSEB", in *Proceedings of the 5th International Conference on Software Reuse*, Victoria, British Columbia, Canada, IEEE Computer Society, June 2-5, 1998, pp. 76-85.
[36] IBM, "Rational Rose Modeler", http://www-306.ibm.com/software/awdtools/developer/rose/modeler/.
[37] IBM, "Rational® RequisitePro®", http://www-306.ibm.com/software/awdtools/reqpro/.
[38] Institute of Electrical and Electronics Engineers Inc., *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York, NY, USA: IEEE, 1991.
[39] Jama Software, "Jama Contour", http://www.jamasoftware.com/contour.htm.
[40] W. Jirapanthong and A. Zisman, "Supporting Product Line Development through Traceability", in *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, Taiwan, IEEE Computer Society, 15-17 Dec. 2005, pp. 506-514.
[41] W. Jirapanthong and A. Zisman, "XTraQue: Traceability for Product Line Systems", *Software and Systems Modeling*, 2007.
[42] F. Jouault, "Loosely Coupled Traceability for ATL", in *Proceedings of the ECMDA Traceability Workshop (ECMDA-TW)*, Nuremberg, Germany, November 2005, pp. 29-37.
[43] F. Jouault and I. Kurtev, "Transforming Models with ATL", in *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, Springer, 2005, pp. 128-138.
[44] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Software Engineering Institute, Technical report, CMU/SEI-90-TR-021, 1990.
[45] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", *Annals of Software Engineering*, vol. 5, pp. 143-168, 1998.
[46] S. S. Khan, A. Rashid, A. Goknil, I. Galvão, I. Groher, C. Schwanninger, J.-C. Royer, K. Garces, and C. Pohl, "State-of-the-art for traceability in software product line development, with specific focus on aspect traceability in the software development process. Evaluate the potential benefits of aspectoriented programming and model-driven engineering." Aspect-Oriented, Model-Driven, Product Line Engineering (AMPLE), Survey, M4.1, 2007.
[47] P. Knauber and J. Schneider, "Tracing Variability from Implementation to Test Using Aspect-Oriented Programming", in *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004)*, Boston, Massachusetts, USA, August 2004, pp. 36-44.
[48] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Merging Models with the Epsilon Merging Language (EML)", in *Procedings of the ACM/IEEE 9th*

International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006), Genova, Italy, Springer, October 2006, pp. 215-229.

[49] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "On-Demand Merging of Traceability Links with Models", presented at the 2nd ECMDA Traceability Workshop (ECMDA-TW), Bilbao, Spain, 2006.

[50] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*, 1st ed., New York, NY, USA: John Wiley & Sons, Inc., 1998.

[51] P. Luttikhuizen, "Requirements Modelling and Traceability", ESAPS project, Consortiumwide deliverable, 2001.

[52] M. Mergel, S. Thiel, and S. Ferber, "Product Line Asset Classification and Dependency Specification", ESAPS project, Consortiumwide deliverable, 2000.

[53] M. Moon, H. S. Chae, and K. Yeom, "A Metamodel Approach to Architecture Variability in a Product Line", in *Proceedings of the 9th International Conference on Software Reuse*, Torino, Italy, Springer Berlin, 11-15 June 2006, pp. 115-126.

[54] L. Naslavsky, T. A. Alspaugh, D. J. Richardson, and H. Ziv, "Using Scenarios to Support Traceability", in *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '05)*, Long Beach, California, USA, ACM, 8 Nov. 2005, pp. 25 - 30.

[55] National Institute of Standards and Technology, "Dictionary of Algorithms and Data Structures", http://www.nist.gov/dads/.

[56] Object Management Group (OMG), "Meta Object Facility (MOF) Specification", http://www.omg.org/docs/formal/02-04-03.pdf, 2002, 03.04.2002.

[57] J. D. Palmer, "Traceability", in *Software Requirements Engineering*, R. Thayer and M. Dorfman, Eds., 2nd ed., Los Alamitos, California: IEEE Computer Society Press, 2000, pp. 412-422.

[58] K. Pohl, "VARMOD-EDITOR", http://www.sse.uni-due.de/wms/de/index.php?go=256.

[59] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Berlin, Germany: Springer, 2005.

[60] Project AMPLE, "Aspect-Oriented, Model-Driven Product Line Engineering", http://ample.holos.pt/.

[61] pure-systems GmbH, "*pure::variants*", http://www.pure-systems.com/Variant_Management.49.0.html.

[62] B. Ramesh and M. Jarke, "Toward Reference Models for Requirements Traceability", *IEEE Transactions on Software Engineering*, vol. 27, pp. 58-93, 2001.

[63] I. Sommerville, *Software Engineering*, 8th ed.: Addison-Wesley, 2007.

[64] A. Sousa, U. Kulesza, A. Rummler, N. Anquetil, R. Mitschke, A. Moreira, V. Amaral, and J. Araújo, "A Model-Driven Traceability Framework to Software Product Line Development", presented at 4th ECMDA Traceability Workshop, Berlin, Germany, 2008.

[65] Sparx Systems Japan, "RaQuest", http://www.raquest.com/.

[66] T. Stahl and M. Volter, *Model-Driven Software Development*, 1st ed., Glasgow, UK: Wiley, 2006.

[67] Sun Microsystems, "Java™ Platform, Standard Edition 6 API Specification ", http://java.sun.com/javase/6/docs/api/.

[68] M. S. Tabares and A. Moreira, "Towards a Meta Aspect for Traceability", presented at Early Aspects: Traceability of Aspects in the Early Life Cycle Workshop (AOSD'06), Bonn, Germany, 2006.
[69] Telelogic, "Telelogic DOORS", http://www.telelogic.com/products/doors/index.cfm.
[70] Triskel Project (IRISA), "The Metamodeling Language Kermeta", http://www.kermeta.org/.
[71] UNL/FCT, "Traceability Requirements", Aspect-Oriented, Model-Driven, Product Line Engineering (AMPLE), AMPLE Internal Documentation, 2007.
[72] P. Zave, "FAQ Sheet on Feature Interaction", http://www.research.att.com/~pamela/faq.html.
[73] A. Zisman, G. Spanoudakis, E. Pérez-Miñana, and P. Krause, "Towards a Traceability Approach for Product Families Requirements", presented at 3rd International Workshop on Software Product Lines: Economics, Architectures, and Implications, Orlando, Florida, USA, 2002.

Glossary of Abbreviations

AMPLE – Aspect-Oriented Model-Driven Product Line Engineering

AOSD – Aspect-Oriented Software Development

ATL – ATLAS Transformation Language

CRS – Commercial Requirements Specification

CRUD – Create, Read, Update and Delete

EBT – Event Based Traceability

EBT_{DP} – Event Based Traceability with Design Patterns

EMF – Eclipse Modeling Framework

EML – Epsilon Merging Language

FCT/UNL – Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

FRS – Functional Requirements Specification

GCT – Goal Centric Traceability

GUI – Graphical User Interface

MDE – Model-Driven Engineering

MOF – Meta-Object Facility

NFR – Non-Functional Requirement

OMG – Object Management Group

OVM – Orthogonal Variability Model

ROM – Requirements Object Model

RTD – Research and Technological Development

SIG – Softgoal Interdependency Graph

SPL – Software Product Lines

UML – Unified Modeling Language

XML – eXtensible Markup Language

Annexes

Annex 1 - Trace Extractor Extension Point

Identifier: net.ample.tracing.core.traceExtractor

Since: 0.1.0

Description: Extension point for trace extractors.

Configuration Markup:

```
<!ELEMENT extension (extractor)+>
<!ATTLIST extension
    point       CDATA #REQUIRED
    id          CDATA #IMPLIED
    name        CDATA #IMPLIED>

<!ELEMENT register (parameter)>
<!ATTLIST register
    Id          CDATA #REQUIRED
    class       CDATA #REQUIRED
    description CDATA #REQUIRED>
```

- **id** - The unique identifier of this extractor.
- **class** - The implementing class.
- **description** - A short description of what the extractor actually does.

```
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
    name        CDATA #REQUIRED
    type        (string|boolean|int|float|resource)
    required    (true | false)
    description CDATA #IMPLIED
    default     CDATA #IMPLIED>
```

Each extractor is configured via several parameters, which can be defined with this tag.

- **name** - Name of the parameter.
- **type** - Type of the parameter. There are five types of possible parameters: string, boolean, int, float and resource.
- **required** - Flag to indicate that this parameter is required for the trace extractor to work properly.
- **description** - A short description of the parameter, that might be displayed in a user interface.
- **default -** The default value of the parameter.

Examples: The following is an example of the extension point usage:

```
<extension point="net.ample.tracing.core.traceExtractor">
   <extractor
      id="net.ample.tracing.sample_extractor"
      class="net.ample.tracing.SampleExtractor”
      description="some description">
        <parameter
```

```
            name="Sample Parameter"
            type="boolean"
            required="true"
            description="some description"
            default="false">
         </parameter>
   </extractor>
</extension>
```

API Information: Plug-ins that want to extend this extension point must implement `net.ample.tracing.core.TraceExtractor` interface.

Annex 2 - Trace Register Extension Point

Identifier: net.ample.tracing.framework.core.traceRegister

Since: 0.1.0

Description: This extension point is used to plug in additional trace registers for establishing trace links between SPL artifacts.

Configuration Markup:

```
<!ELEMENT extension (register)+>
<!ATTLIST extension
    point       CDATA #REQUIRED
    id          CDATA #IMPLIED
    name        CDATA #IMPLIED>

<!ELEMENT register (description)>
<!ATTLIST register
    id          CDATA #REQUIRED
    name        CDATA #REQUIRED
    class       CDATA #REQUIRED>
```

- **id** - a unique name that will be used to reference this trace register.
- **name** - a translatable name that will be used for presenting this trace register in the UI.
- **class** - Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.traceregister.ITraceRegister` interface.

```
<!ELEMENT description (#PCDATA)>
```

Examples: The following is an example of the extension point usage:

```
<extension point="net.ample.tracing.framework.core.traceRegister">
   <register
      id="net.ample.tracing.sample_register"
      name="Sample Trace Register"
      class="net.ample.tracing.SampleRegister">
        <description>some description.</description>
   </register>
</extension>
```

API Information: Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.traceregister.ITraceRegister` interface.

Supplied Implementation: Traceability Framework Plug-in provides a default implementation of a trace register.

Annex 3 - Trace Query Extension Point

Identifier: net.ample.tracing.framework.core.traceQuery

Since: 0.1.0

Description: This extension point is used to plug in additional trace queries.

Configuration Markup:

```
<!ELEMENT extension (query)+>
<!ATTLIST extension
    point       CDATA #REQUIRED
    id          CDATA #IMPLIED
    name        CDATA #IMPLIED>

<!ELEMENT query (description)>
<!ATTLIST query
    id          CDATA #REQUIRED
    name        CDATA #REQUIRED
    class       CDATA #REQUIRED>
```

- **id** - a unique name that will be used to reference this trace query.
- **name** - a translatable name that will be used for presenting this trace query in the UI.
- **class** - Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.tracequery.ITraceQuery` interface.

```
<!ELEMENT description (#PCDATA)>
```

Examples: **The following is an example of the extension point usage:**

```
<extension point="net.ample.tracing.framework.core.traceQuery">
   <query
      id="net.ample.tracing.sample_query"
      name="Sample Trace Query"
      class="net.ample.tracing.SampleQuery">
        <description>some description.</description>
   </query>
</extension>
```

API Information: Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.tracequery.ITraceQuery` interface.

Supplied Implementation: Traceability Framework Plug-in provides a default implementation of a trace query.

Annex 4 - Trace View Extension Point

Identifier: net.ample.tracing.framework.core.traceView

Since: 0.1.0

Description: This extension point is used to plug in additional trace views.

Configuration Markup:

```
<!ELEMENT extension (view)+>
<!ATTLIST extension
    point       CDATA #REQUIRED
    id          CDATA #IMPLIED
    name        CDATA #IMPLIED>

<!ELEMENT view (description)>
<!ATTLIST view
    id          CDATA #REQUIRED
    name        CDATA #REQUIRED
    class       CDATA #REQUIRED
```

- **id** - a unique name that will be used to reference this trace view.
- **name** - a translatable name that will be used for presenting this trace view in the UI.
- **class** - Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.traceview.ITraceView` interface.

```
<!ELEMENT description (#PCDATA)>
```

Examples: The following is an example of the extension point usage:

```
<extension point="net.ample.tracing.framework.core.traceView">
   <view
      id="net.ample.tracing.sample_view"
      name="Sample Trace View"
      class="net.ample.tracing.SampleView">
        <description>some description.</description>
   </view>
</extension>
```

API Information: Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.traceview.ITraceView` interface.

Supplied Implementation: Traceability Framework Plug-in provides a default implementation of a trace view

Annex 5 - Traceability Framework User Guide

Traceability Framework to SPL Development

User Guide
&
Instantiation Manual

(V 0.3)

André Sousa

DI – FCT/UNL

TABLE OF CONTENTS

Introduction

This document describes a Framework that provides an open and flexible platform to implement trace links between artifacts from SPL development. In order to address this aim, the framework was designed and implemented with several hotspots that allow developers to extend its capabilities as needed.

The document is divided into three main parts: framework description, tutorial and framework instantiation. The first part describes the concepts behind this trace framework and gives an overview of the same. The second chapter contains tutorials to guide the user in his first steps. The third section shows how to instantiate the hotspots of this framework to extend the functionalities provided by default.

System Requirements

This framework was developed as an Eclipse plug-in and was designed to work with the following set of requirements.

- JRE 5.0 (the framework core runs on JRE 5.0, however the framework extensions provided by default require JRE 6.0 to run).
- Eclipse SDK 3.3.2
- ATF 0.1.7 (All ATF requirements must also be satisfied. Check ATF documentation)

Installation

To install this framework, simply copy the contents of the file *spl_traceability_framework_core_X.X.X.zip* (core) and *spl_traceability_framework_extensions_Y.Y.Y.zip* (framework instances) to your Eclipse installation directory and launch Eclipse.

Framework Description

This section describes the fundamental concepts of the trace repository

Traceability Framework Structure

Our traceability framework aims to provide an open and flexible platform to implement trace links between different artifacts from SPL development. For now the variability model (feature model) is used in this approach as the main reference to trace the SPL artifacts. However, the design of the framework is generic, so it may be applied outside SPL development.

The following main functionalities are provided by our framework to support the tracing of SPL artifacts:

- creation and maintenance of trace links between a variability model and other existing artifacts (UML models, architecture models, source code);
- persistent storage of trace links using a repository (ATF);
- searching of specific trace links between artifacts using pre-defined or customized trace queries. Trace queries can be executed over the trace links in order to select interesting traceability information to help the SPL development or evolution;
- flexible visualization of the results of trace queries using different types of trace views, such as, tree views, graphs, tables, etc.

The architecture of this traceability framework is shown in Figure 1 using a UML class diagram. The classes that represent a hotspot (interfaces and abstract classes) must be instantiated in to provide the required functionalities.

Our traceability framework is structured as an object-oriented framework that defines an infrastructure to provide basic services to search and store trace links and it also offers a set of extension points to create specific SPL traceability functionalities (trace queries and views).

The *ITraceRegister*, *ITraceQuery* and *ITraceView* interfaces, along with the respective abstract classes (*AbstractTraceRegister* and *AbstractTraceQuery*) represent the extension points of the framework's main components.

Each of them must be instantiated and customized to address specific traceability scenarios in SPL development.

The *AbstractTraceRegister* class must be specialized to create specific ways to create and store trace links between artifacts. The `executeRegister()` abstract method must be implemented for this purpose. The trace links are stored using the services provided by an ATF repository. The framework does not specify the concrete ways that the trace links must be obtained. This functionality can be provided, for example, by specifying a strategy to automatically identify possible trace links between artifacts or by providing a graphical interface to allow the SPL developers to manually define the desired trace links.

The *AbstractTraceQuery* class establishes the general structure to implement traceability queries. The method `submitQuery()` allows each instance to implement a specific type of trace query. It uses the query services provided by ATF component to search trace links of interest in the repository. After that, it delegates the resulted trace links from its query to an associated trace view by calling the `showResults()` method.

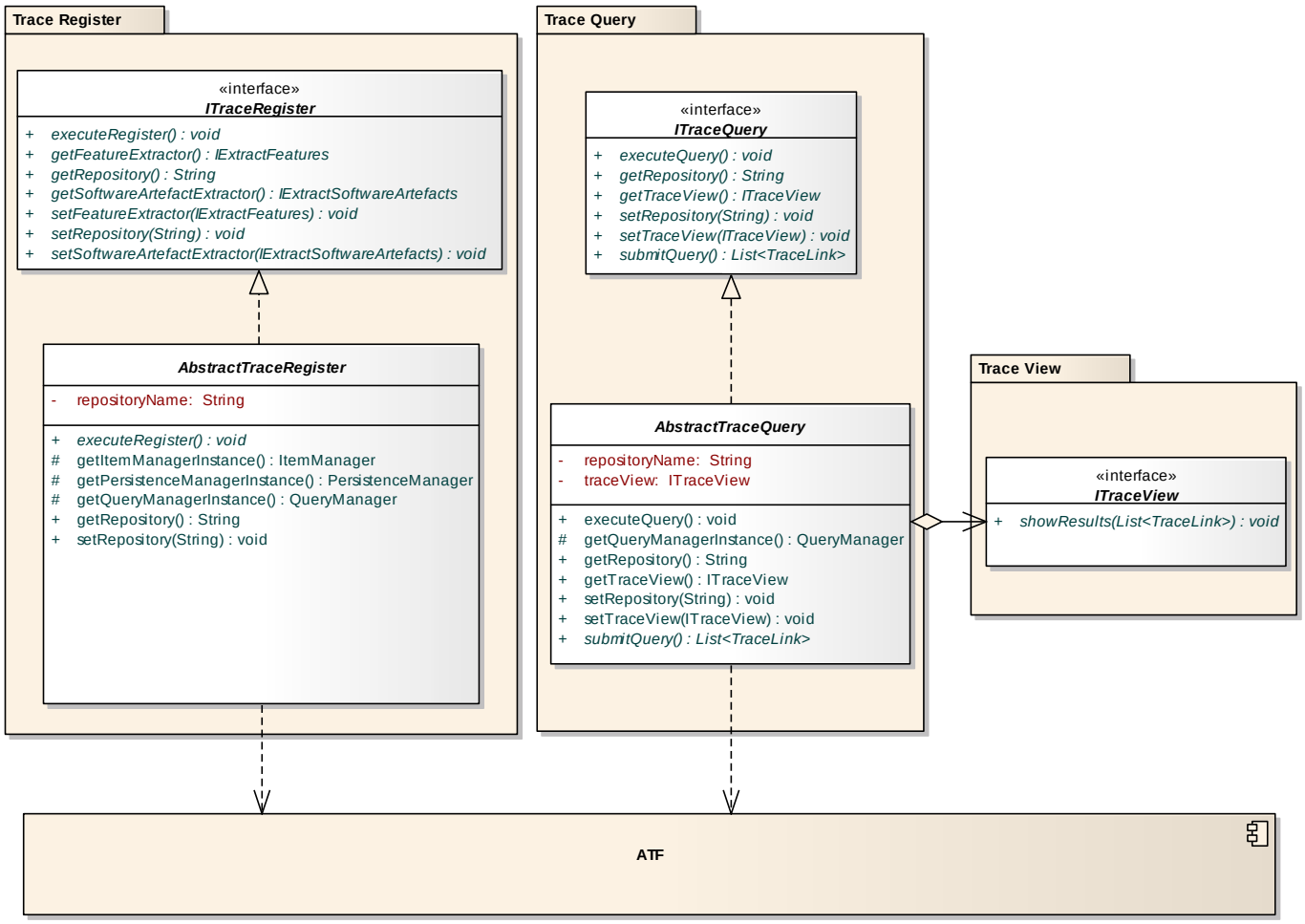


Figure 1 - Traceability Framework Architecture

The trace views are implemented as classes implementing the *ITraceView* interface. The ATF component provides basic traceability services to retrieve and query basic trace links between specific artifacts. Our framework aims to create more advanced traceability queries (such as, requirements/feature coverage, change impact analysis, product variants tracing) built on top of these basic ones.

Framework Implementation

The framework described above, has been implemented as an Eclipse plug-ins called *net.ample.tracing.framework.core*. This plug-in defined several extension points that are used to create instances of each of the framework's hotspots. Figure 2 shows the five extension points defined in *net.ample.tracing.framework.core*. Each of these extension points as an attached Schema that explains how this extension can be used by extending plug-ins. Check the Extension Points Reference in Appendix III for this info.

Figure 2 - net.ample.tracing.framework.core extension points

To automate the detection and usage of new extension added to the base framework, another Eclipse plug-in (*net.ample.traceabilityProject*) has been implemented, which provides a front-end that allows users to define a traceability project and perform the actions desired on the framework instances. When new extensions are defined, this front-end makes them automatically available.

API Reference

The Framework API and Extension Points Reference can be checked at http://ample.di.fct.unl.pt/TraceFramework/

Default Instantiation

The Traceability Framework that is provided has some default instantiations for each of the hotspots.

Trace Register

The trace register instance provides a GUI that allows developers to define trace links between the features and requirement artifacts extracted using any of the provided extractors. This trace register provides a GUI for manual definition of trace links. The tutorial section contains more details on how to perform this task.

Trace Query

Two trace query instances are provided by default. One is capable of finding the artifacts that are related with a set of chosen features. The second performs a change impact analysis query. For now these are the only queries available, but there are plans to develop queries capable of detecting feature interactions, query by product variant and others. The tutorial section contains more details on how to perform this task.

Trace View

The results returned by a query can be seen in one of the trace view instances available. There are currently four views implemented, each designed to show the results of a type of query. Two instances provide a tree view of the artifacts related with each feature, a more detailed one (which includes the steps of each use case) and a general overview (excluding use case steps). These two views should be used to browse the results returned by a related artifacts query. The two remaining views are used for presenting the results of a change impact analysis query. As with the related artefacts view, a more detailed and a general overview are available. If the wrong view is used, (for instance, using a related artifacts query with a change

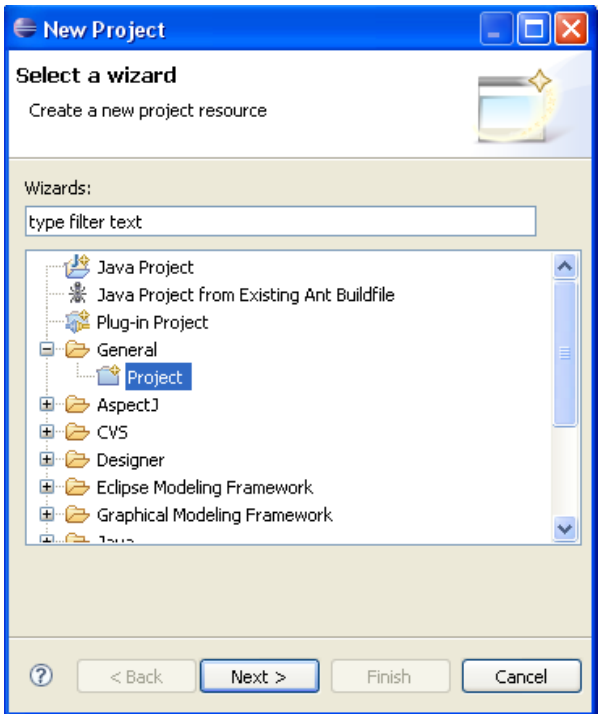
impact analysis view), it may result in presenting trace information that does not correspond to the expected results. This is due to the fact that each view is prepared to process the list of links returned by the appropriate query. The tutorial section contains more details on how to perform this task.

Tutorials

This section of the documentation contains some tutorials for common tasks to be carried out when working with the traceability framework. These tutorials will be based on the default framework instantiation explained previously and the SPL case study found in Appendix I and Appendix II. Please refer to them for info on the case study.

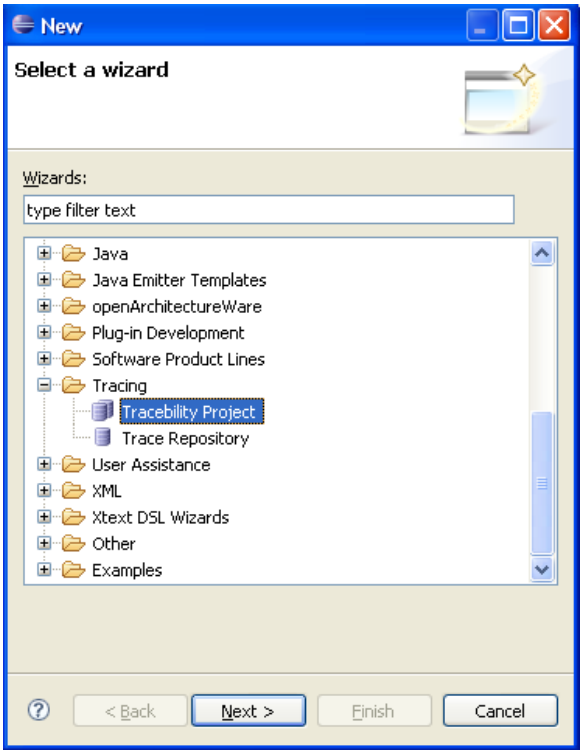
How to Create a Traceability Project

Traceability Projects can be created via the user interface. The first step is to create a new project in Eclipse (you can also use an existing one if you wish). Inside that project you can either create folders to contain your traceability project files, or just dump them into the newly created project. Select *File* > *New* > *Project*.



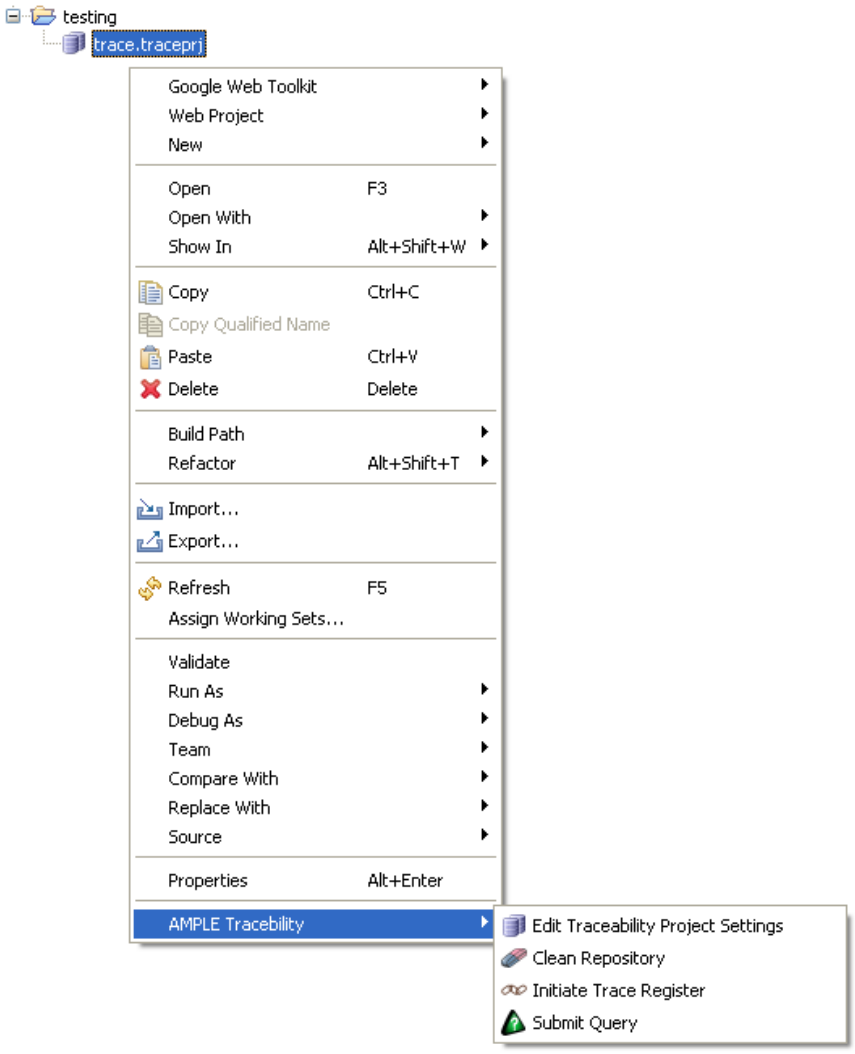
Once the project is created select *File* > *New* > *Other…* and choose the new **Traceability Project** option. A wizard will open, guiding you through the process of a

new traceability project creation. You 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 traceable artifacts. The new types of traceable artifacts are *Feature*, *Use Case*, *Step*, *Actor* and *Package*, which can be use to store the appropriate traceable artifacts. The new types of trace links are *Relationship* and *Hierarchy*. Relationship is used to create a link between a feature (from the variability model) and an Actor/Use case/Step/Package (from the requirements model). A Hierarchy link is used to define hierarchical information, such as the fact that a step belongs to its parent use case.

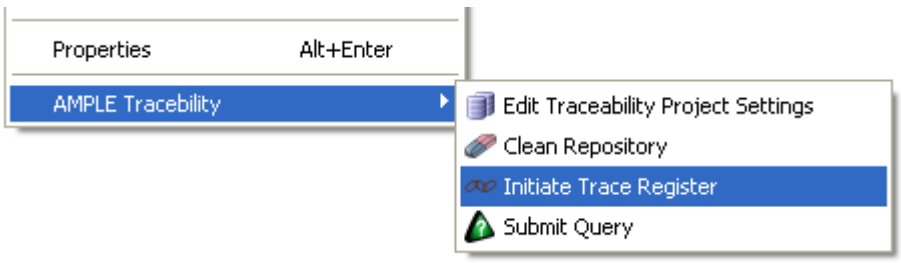
Once the project is created a popup menu will be available for the traceability project file. To access it, just right click on the project file and choose the desired action.

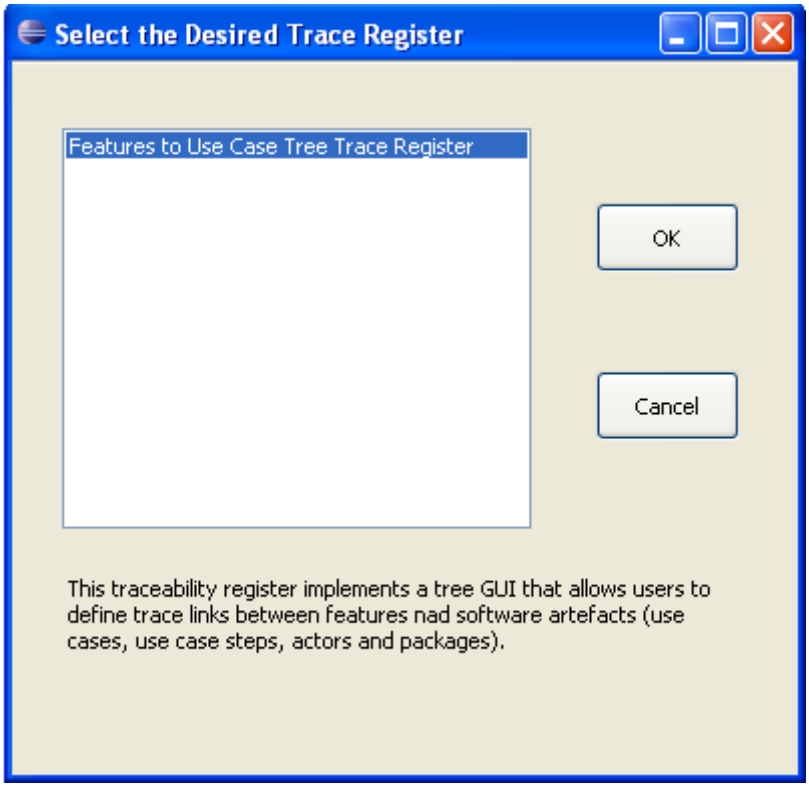


Defining new Trace Links

To define new trace links, we must initialize a trace register instance.

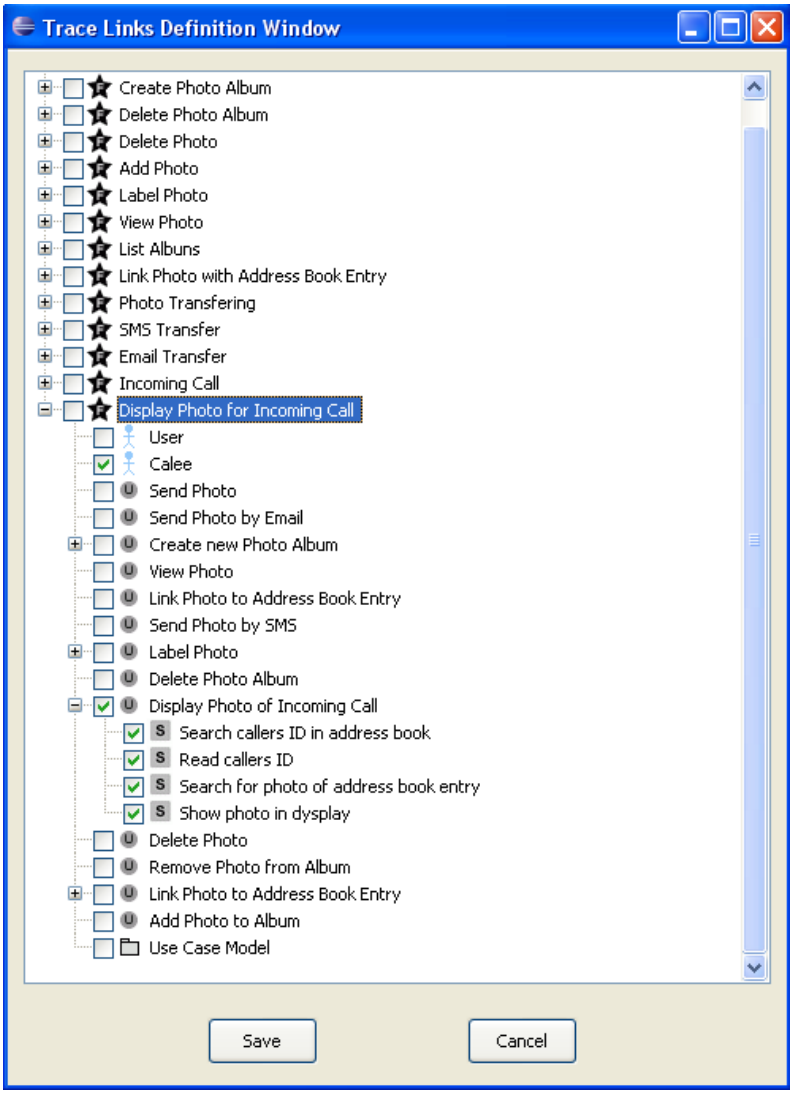
Choose **Initiate Trace Register** from the popup menu and then choose the desired trace register instance from the list of available trace registers.





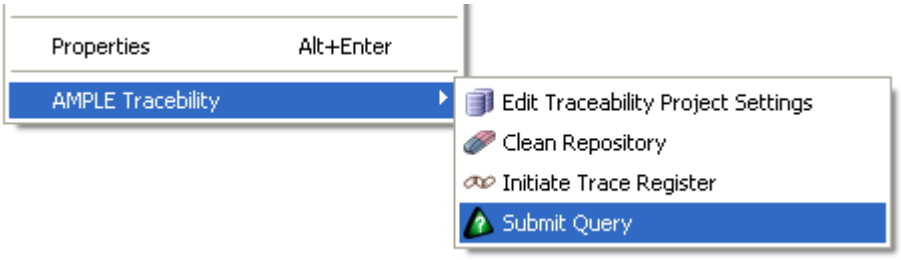
Now define the desired trace links between features and requirement artifacts by checking the corresponding boxes. For instance, the box labeled “Read callers ID” under the “Display Photo for incoming Call” feature is checked, so a link between that feature and that use case step will be created in the repository.

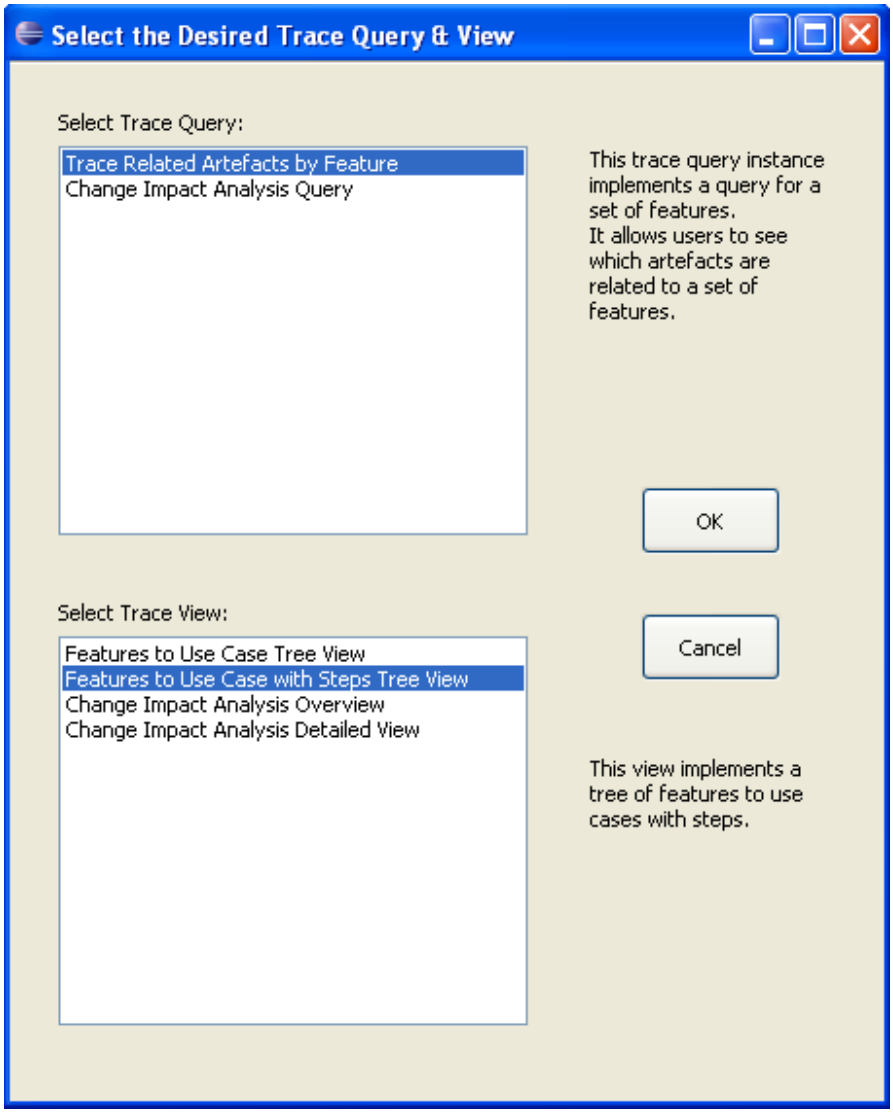
When the *Save* button is pressed the changes performed in the trace register window are committed to the ATF repository.



Submitting Queries and Viewing Results

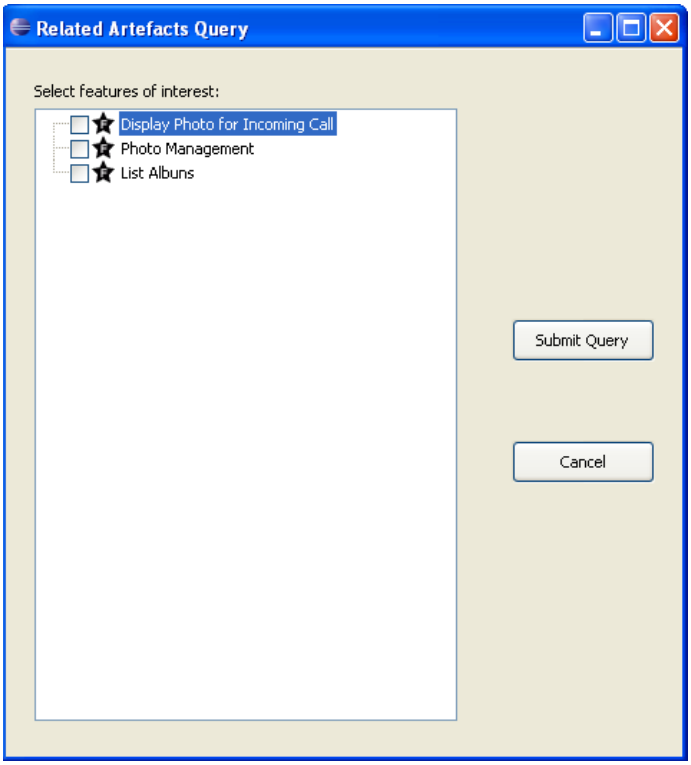
To submit a query to the ATF repository, choose Submit Query from the popup menu and then choose the desired trace query instance and trace view from the list of available extensions.

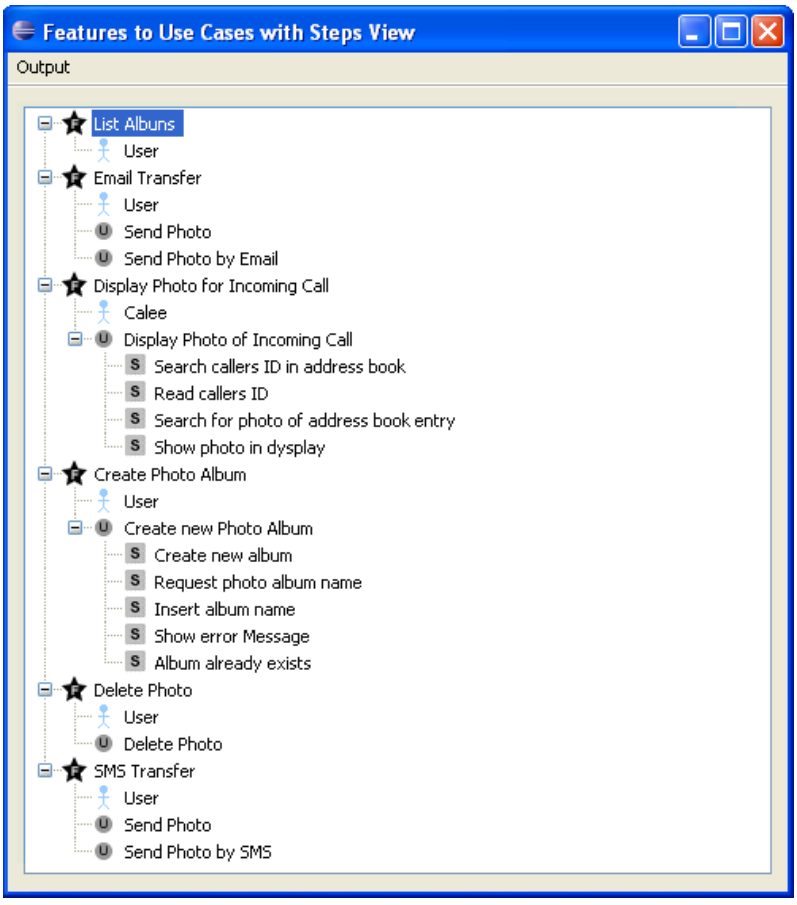




On the next window just choose your FMP features model file, and once the feature model tree is displayed you should choose which features are to be queried. As explained previously, the **Trace Related Artefacts by Feature** query, finds the artifacts that are linked to a set of chosen features.

Once the choice has been made, click the Submit Query button, and the results will be displayed in the chosen view.





The menu **Output** can be used to write the results of shown in the selected view to an output file chosen by the user.

The “Change Impact Analysis Query” usage follows the same guidelines.

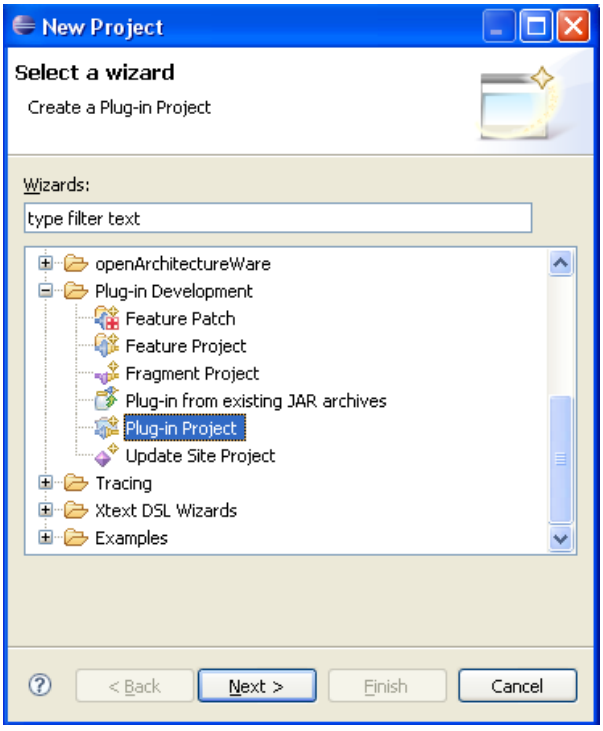
Framework Instantiation

This section of the documentation describes how the framework hotspots can be instantiated to allow developers to add new functionality to the base implementation provided by default. To help demonstrating these steps, the tutorial shown here are based on some fictitious examples (the file formats, algorithms and heuristics described in this section are meant to be used only for demonstrating how to create framework instances. They are in no way meant to be used in a real software development environment).

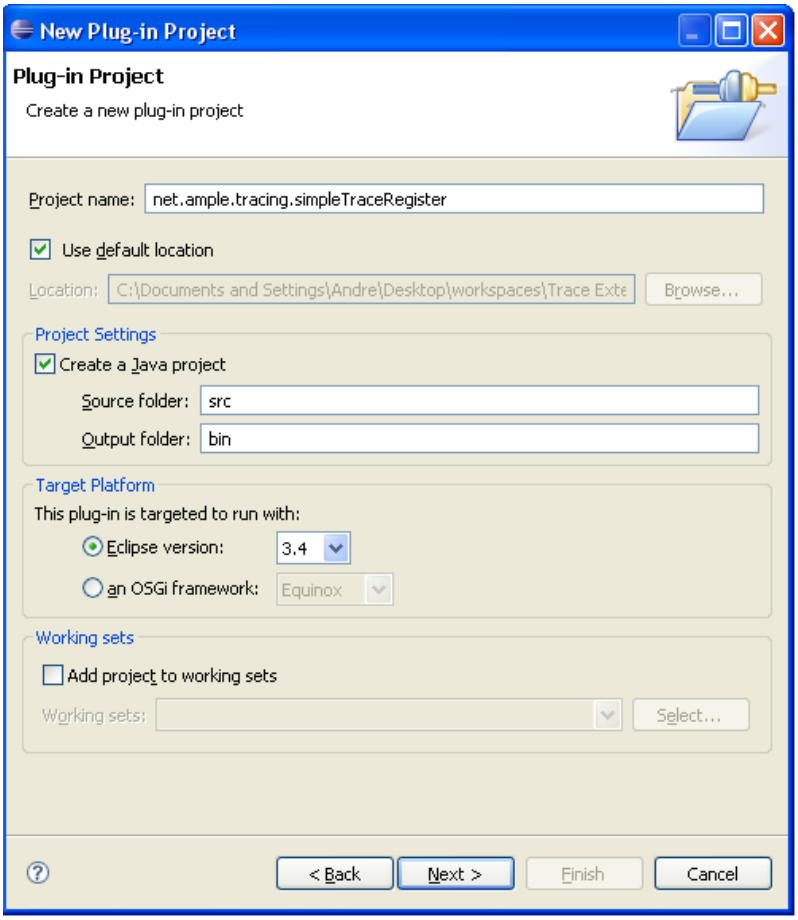
The complete mechanism behind Eclipse plug-in development will not be explained here, as it falls out of the scope of this document. The user is expected to have a basic understanding on this subject.

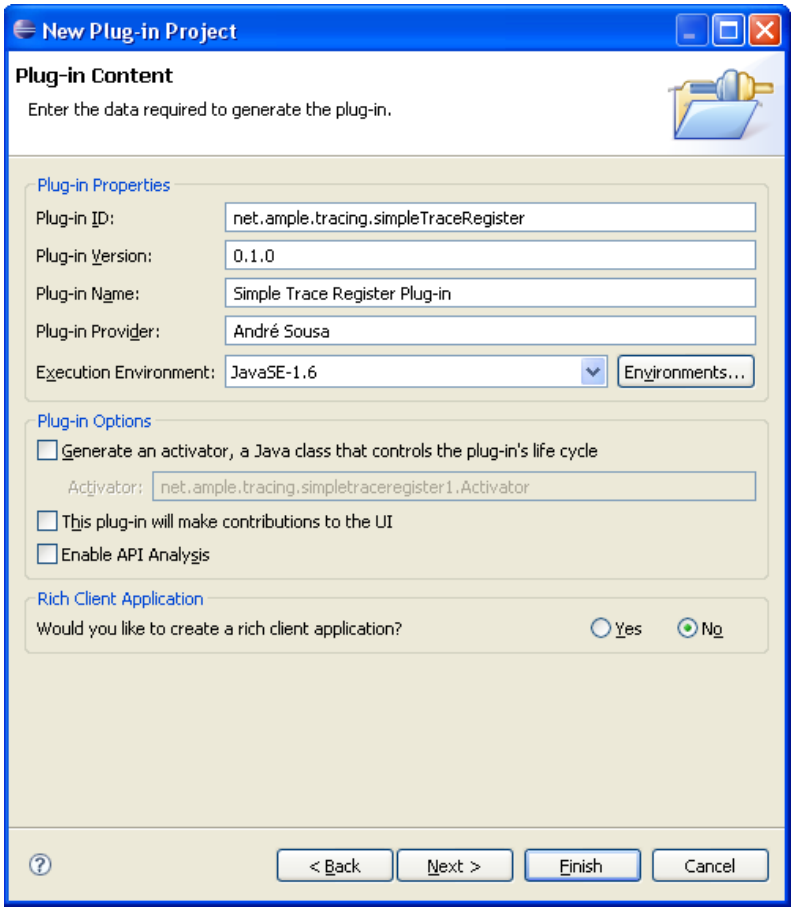
Trace Register Instance

To create a new trace register instance we begin by creating a new plug-in. *File* > *New* > *Project* and choose Plug-in Project.

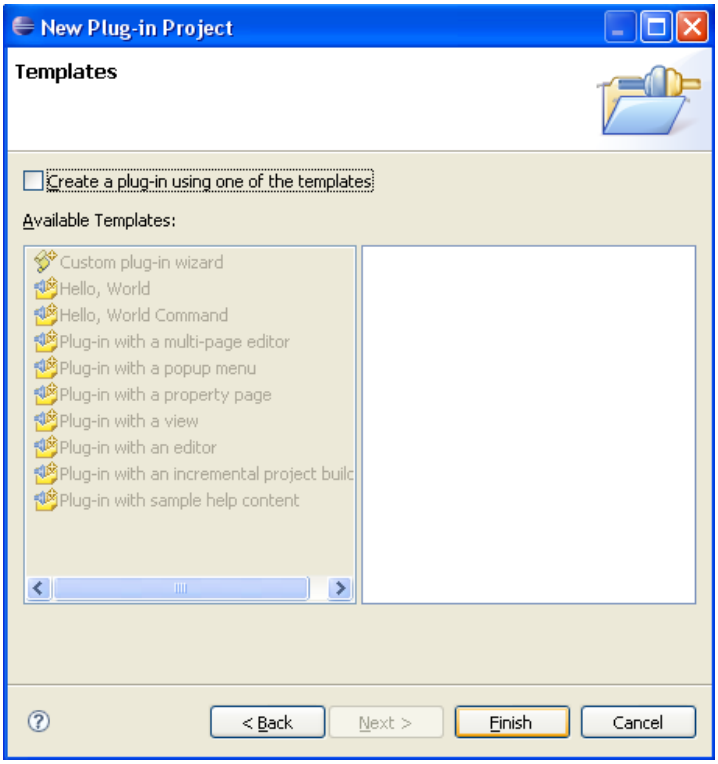


Then we will define this plug-in settings. For the project name we will choose *net.ample.tracing.simpleFeaturesExtractor*. Leave the rest of the settings as follows.

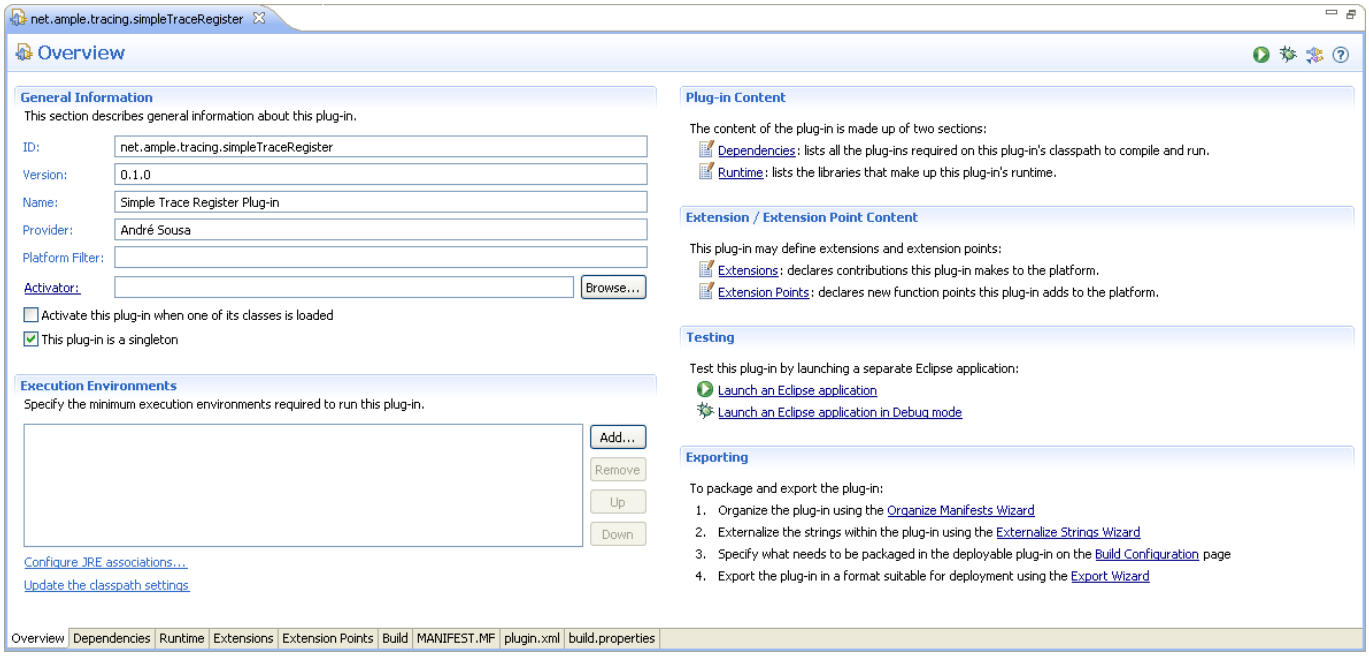

New Plug-in Project
Plug-in Project
Create a new plug-in project
Project name: net.ample.tracing.simpleTraceRegister
Use default location
Location: C:\Documents and Settings\Andre\Desktop\workspaces\Trace Exte
Browse...
Project Settings
Create a Java project
Source folder: src
Output folder: bin
Target Platform
This plug-in is targeted to run with:
Eclipse version: 3.4
an OSGi framework: Equinox
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel


New Plug-in Project
Plug-in Content
Enter the data required to generate the plug-in.
Plug-in Properties
Plug-in ID: net.ample.tracing.simpleTraceRegister
Plug-in Version: 0.1.0
Plug-in Name: Simple Trace Register Plug-in
Plug-in Provider: André Sousa
Execution Environment: JavaSE-1.6
Environments...
Plug-in Options
Generate an activator, a Java class that controls the plug-in's life cycle
Activator: net.ample.tracing.simpletraceregister1.Activator
This plug-in will make contributions to the UI
Enable API Analysis
Rich Client Application
Would you like to create a rich client application?
Yes
No
< Back
Next >
Finish
Cancel

On the last window we will not use one of the templates, so we will uncheck that box.



Press the *Finish* button and after Eclipse finishes creating all the files, we are place in the Plug-in Development view for our newly created plug-in.

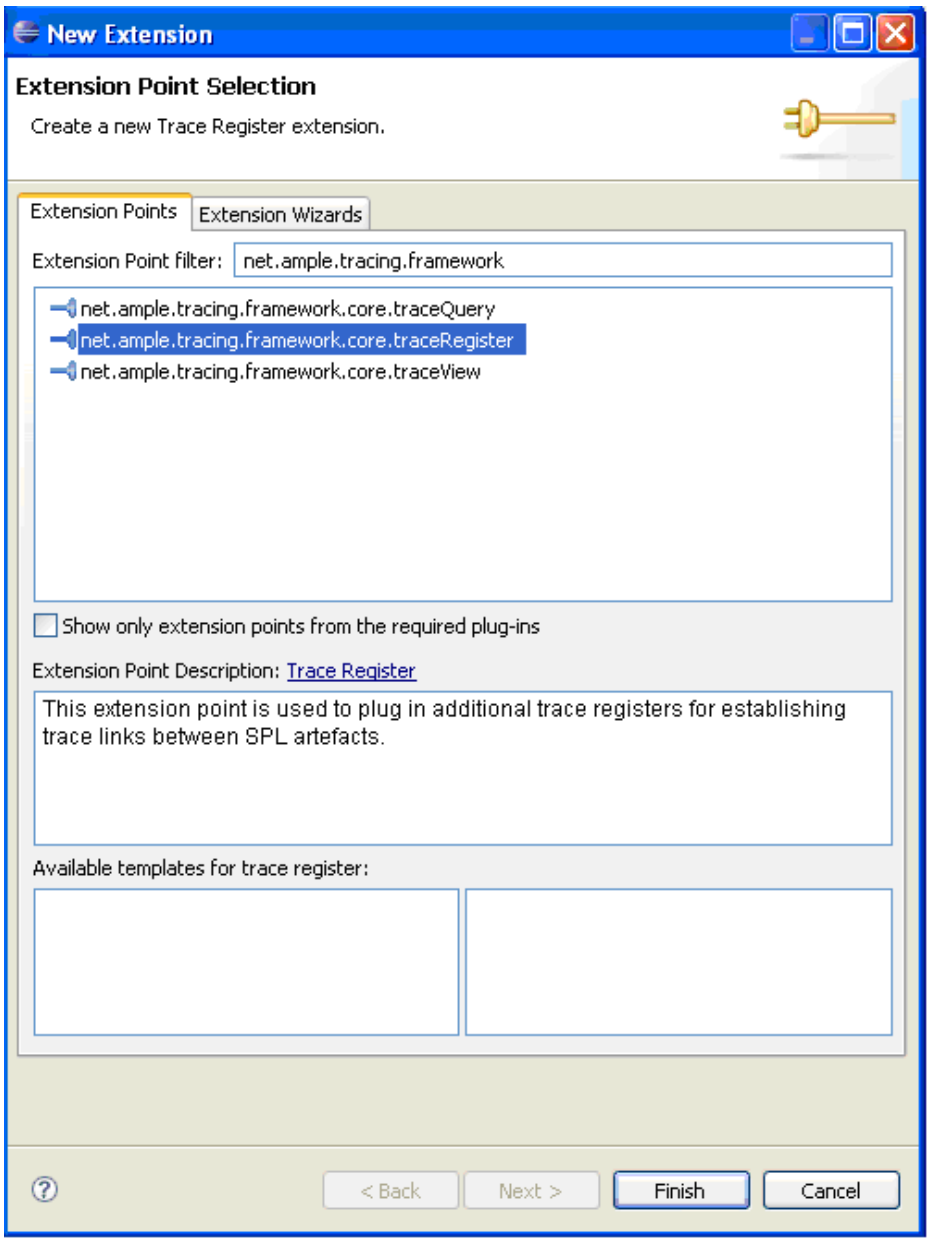


No we must define our new extension point. To do this, we will go to the *extensions* tab and add a new extension. Click *Add…* and in the new window put **net.ample.tracing.framework** in the *Extension Point Filter*, to remove all the other extension points from the list.

If you recall the definition of the *ITraceRegister* in the Framework description, there is no information on how the links are to be created. That is a decision that must be made by the person that implements a trace register. You can devise a method to automatically create trace links between elements (for instance, based on some heuristic), or you can just simply ask the user to manually define (using a GUI) the trace links that he wishes to store.

We are going to create a very simple trace register that is not meant to be used in real SPL development. It is used solely for the purpose of demonstrating how to create a trace register instance. With that in mind, we are going to implement a trace register that creates a new trace link between all the extracted features and use case artifacts, i.e. every feature will be linked to every use case.

After creating the new Plug-in Project, go to the Extensions tab and add a new extension. Choose the *net.ample.tracing.framework.core.traceRegister* extension and press *Finish*.



Now we must complete the configuration of our new extension in order for it to behave as expected. Expand the extension point tree, and go to the *extractor* element. Each field in this element has the following meaning:

Property	Description
id	A unique name that will be used to reference this trace register.
name	A translatable name that will be used for presenting this trace register in the UI.
class	Plug-ins that want to extend this extension point must implement *net.ample.tracing.framework.core.traceregister.ITraceRegister* interface.

Fill in each of these fields with the following values (without the quotes):

id = “net.ample.tracing.simpleTraceRegister.register”

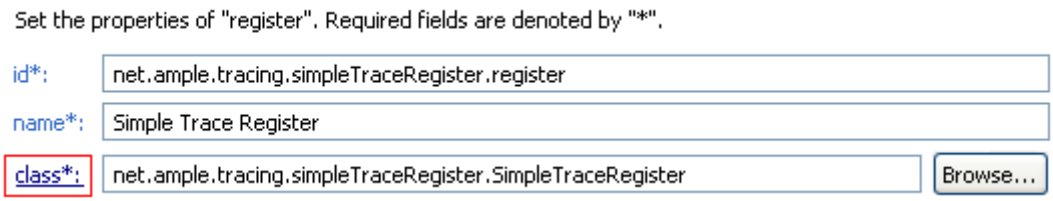
name = “Simple Trace Register”

class = “net.ample.tracing.simpleTraceRegister.SimpleTraceRegister”

If you want, you can also add a description for this extension point in the corresponding element.

Once we have defined all the attributes for our extension point, we are now going to implement a Java class that implements the required interface, i.e. *ITraceRegister* (check Figure 1).

Because there is an abstract class that implements some of the standard methods, and to make the process simpler, we can get Eclipse to do some of the work for us. Click on *class**.



And in the new window, just press *Finish*.

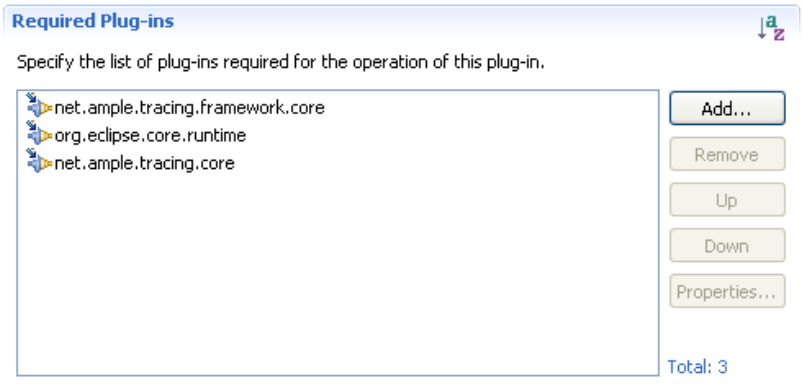
Now we must implement our *SimpleTraceRegister* class, and the method `executeRegister()` to create the trace links. To implement this class just copy the code shown below.

```
package net.ample.tracing.simpleTraceRegister;

import java.io.IOException;
```

```
import java.util.Hashtable;
import java.util.List;
import net.ample.tracing.core.ItemManager;
import net.ample.tracing.core.PersistenceManager;
import net.ample.tracing.core.QueryManager;
import net.ample.tracing.core.TraceLink;
import net.ample.tracing.core.TraceLinkType;
import net.ample.tracing.core.TraceableArtefact;
import net.ample.tracing.core.TraceableArtefactType;
import net.ample.tracing.core.query.Constraints;
import net.ample.tracing.core.query.Query;
import net.ample.tracing.framework.core.TRACEABILITY_FRAMEWORK;
import net.ample.tracing.framework.core.exceptions.ExtractionException;
import net.ample.tracing.framework.core.exceptions.MissingRepositoryException;
import net.ample.tracing.framework.core.extraction.Artefact;
import net.ample.tracing.framework.core.traceregister.AbstractTraceRegister;
import net.ample.tracing.framework.core.traceregister.ITraceRegister;
import org.eclipse.core.runtime.CoreException;

public class SimpleTraceRegister extends AbstractTraceRegister implements
ITraceRegister {

  private ItemManager itemManager;
  private PersistenceManager persistenceManager;
  private QueryManager queryManager;

  public SimpleTraceRegister() {
  }

  @Override
  public void executeRegister() throws MissingRepositoryException,
CoreException, IOException, ExtractionException {
    Hashtable<Artefact,TraceableArtefact> createdArtefacts = new
Hashtable<Artefact,TraceableArtefact>();
    itemManager = getItemManagerInstance();
    persistenceManager = getPersistenceManagerInstance();
    queryManager = getQueryManagerInstance();
    List<Artefact> features = getFeatureExtractor().getFeatures();
    List<Artefact> artefacts =
getSoftwareArtefactExtractor().getSoftwareArtefacts();
    for(int i=0; i<features.size(); i++) {
      for(int j=0; j<artefacts.size(); j++) {
        if(artefacts.get(j).getArtefactType().equals(
TRACEABILITY_FRAMEWORK.USE_CASE_ARTEFACT)) {
          TraceableArtefact feature = createArtefact(features.get(i),
createdArtefacts);
          TraceableArtefact artefact = createArtefact(artefacts.get(j),
createdArtefacts);
          TraceLinkType linkType =
getTraceLinkType(TRACEABILITY_FRAMEWORK.RELATIONSHIP_LINK);
          TraceLink link = itemManager.createTraceLink(feature, artefact,
linkType);
          persistenceManager.begin();
          persistenceManager.add(feature);
          persistenceManager.add(artefact);
          persistenceManager.add(link);
          persistenceManager.commit();
        }
      }
    }
  }
```

```
  private TraceableArtefact createArtefact(Artefact artefact,
Hashtable<Artefact,TraceableArtefact> createdArtefacts) {
    TraceableArtefactType artType = getTraceableArtefactType(
artefact.getArtefactType());
    //verify if it as already created this artifact.
    if(createdArtefacts.containsKey(artefact)) {
      return createdArtefacts.get(artefact);
    }
    //verify if this artifact is already stored in the repository.
    TraceableArtefact target =
getTraceableArtifact(artefact.getArtefactName(),artType);
    if(target != null) {
      return target;
    }
    //if it does not exist, then create a new one.
    else {
      target = itemManager.createTraceableArtefact(artType,
artefact.getArtefactName());
      createdArtefacts.put(artefact, target);
      return target;
    }
  }

  private TraceableArtefact getTraceableArtifact(String name,
TraceableArtefactType type) {
    Query<TraceableArtefact> query = queryManager.queryOnArtefacts();
    query.add(Constraints.and(Constraints.name(name),Constraints.type(type)));
    return query.executeUnique();
  }

  private TraceableArtefactType getTraceableArtefactType(String name) {
    Query<TraceableArtefactType> query = queryManager.queryOnArtefactTypes();
    query.add(Constraints.name(name));
    return query.executeUnique();
  }

  private TraceLinkType getTraceLinkType(String name) {
    Query<TraceLinkType> query = queryManager.queryOnLinkTypes();
    query.add(Constraints.name(name));
    return query.executeUnique();
  }
}
```

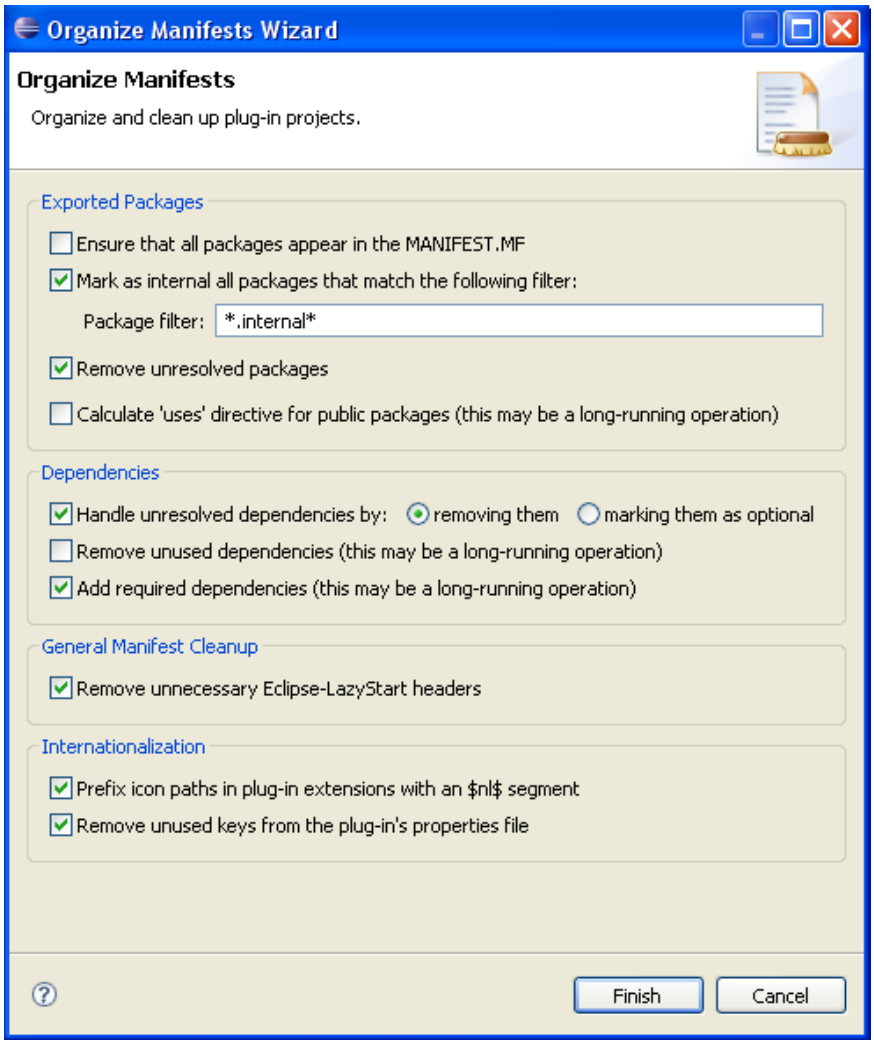
If you copy this code, you will get some errors regarding some missing dependencies. To solve this just go to the **Dependencies** tab and add the following dependencies to your plug-in



We must take some considerations into account when creating extension points. All extension points classes must provide a constructor with no arguments. This is necessary due to the reflection mechanisms included in the framework that allow new extensions to be detected and launched automatically.

Now that we have written our code to parse the input file, we can finish up our plug-in and finally pack it in a JAR file, ready to be used.

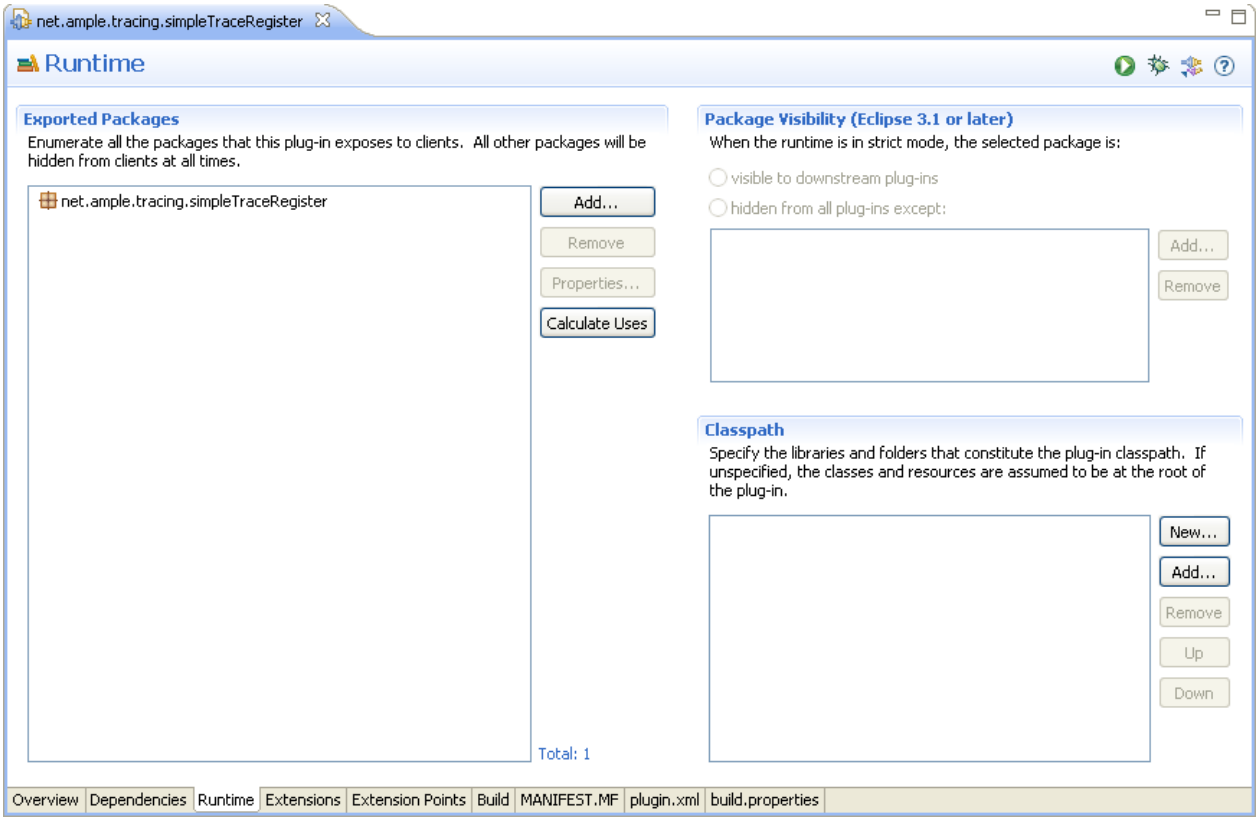
The first thing to do is going to Overview tab and right clicking on it. You will see a menu with an option **Externalize Strings**. Choose this option and in this menu just press **Select All** and then *Finish*. Still in the Overview tab go to **Organize Manifests Wizard** and put in your preferences as follows.



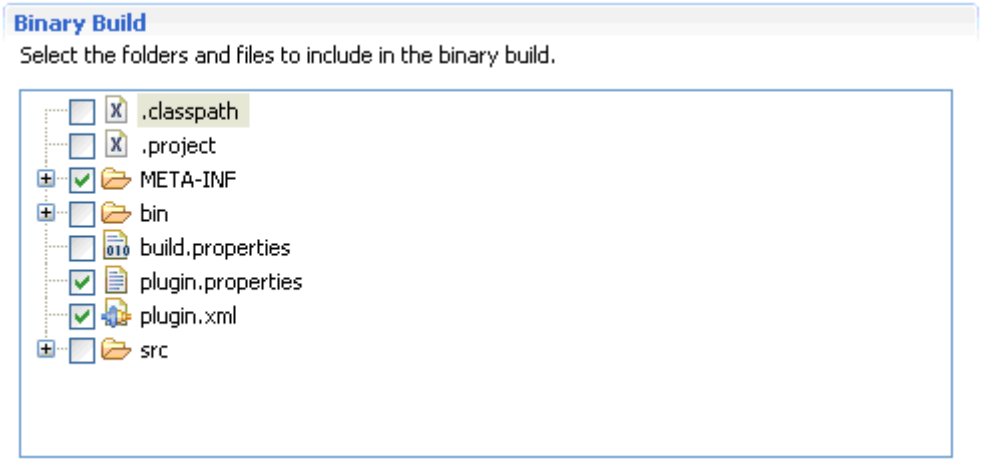
Now we must edit the list of exported packages. This is necessary, because other plug-ins will only see the packages available in this list. If you forget to add the package containing the *SimpleTraceRegister* class, it will not be accessible by anyone.

There are two ways of doing this, you can manually edit you MANIFEST.MF (for experts only) or you can use the Eclipse Plug-in Development Environment to help you out.

Just go to the Runtime tab and add *net.ample.tracing.simpleTraceRegister* to your list of exported packages.



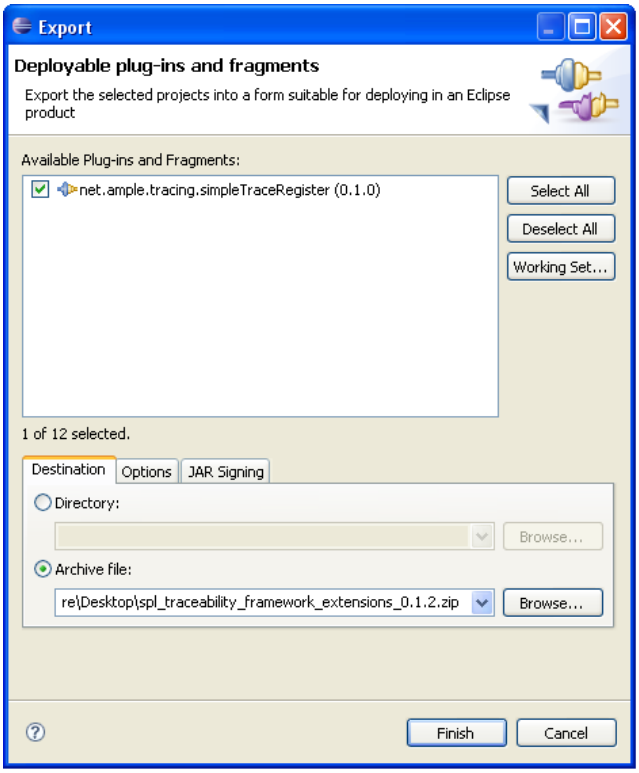
Finally go to the Build tab and choose the following configurations.

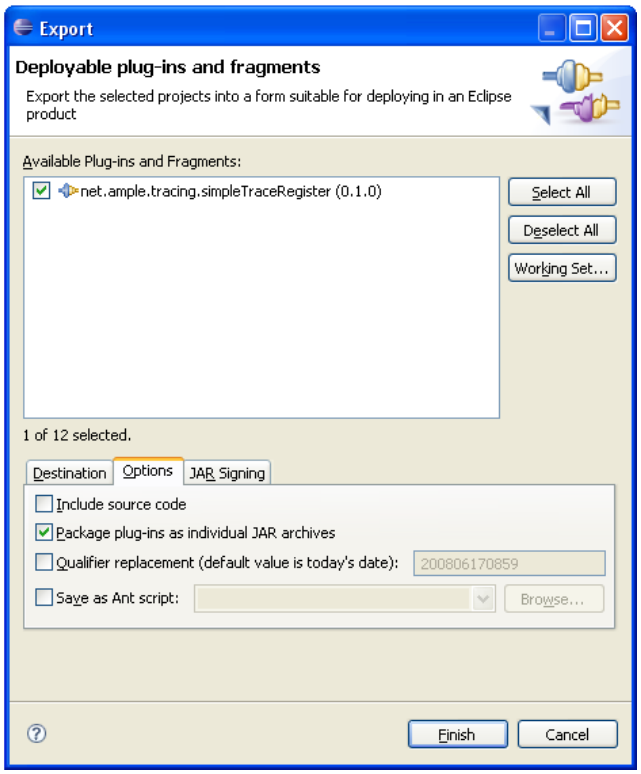


This will ensure that your binary build will include all the necessary files.

Once you are done with the configurations, all there is left to do is exporting the plug-in as a JAR file. This is also an easy task to perform, since Eclipse provides a wizard to do all the hard work.

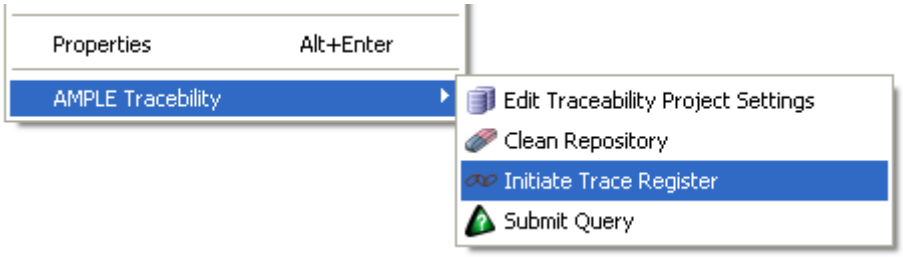
So, just go back to the Overview tab and click on **Export Wizard**. Now select an output directory or an archive file (whichever you prefer) and make sure that you select **Package plug-ins as individual JAR archives**. Press *Finish* and wait until Eclipse finishes the exporting process.



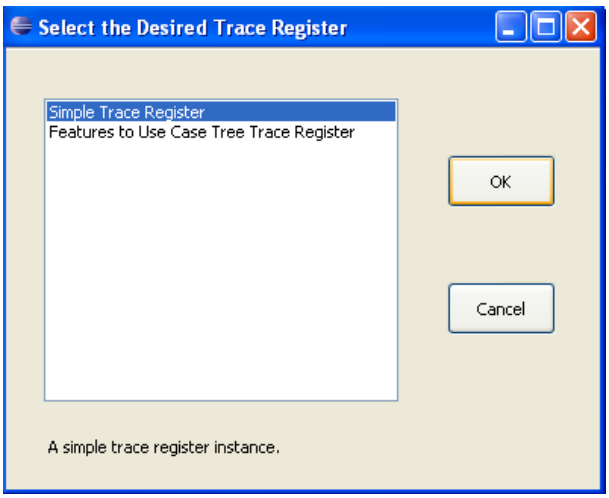


After the plug-in as been exported, you should copy the JAR file into your “plugins” directory inside the Eclipse installation directory.

Restart Eclipse, and right click your traceability project file and choose “Initiate Trace Register”.



The new trace register extension is automatically displayed and ready to be used.



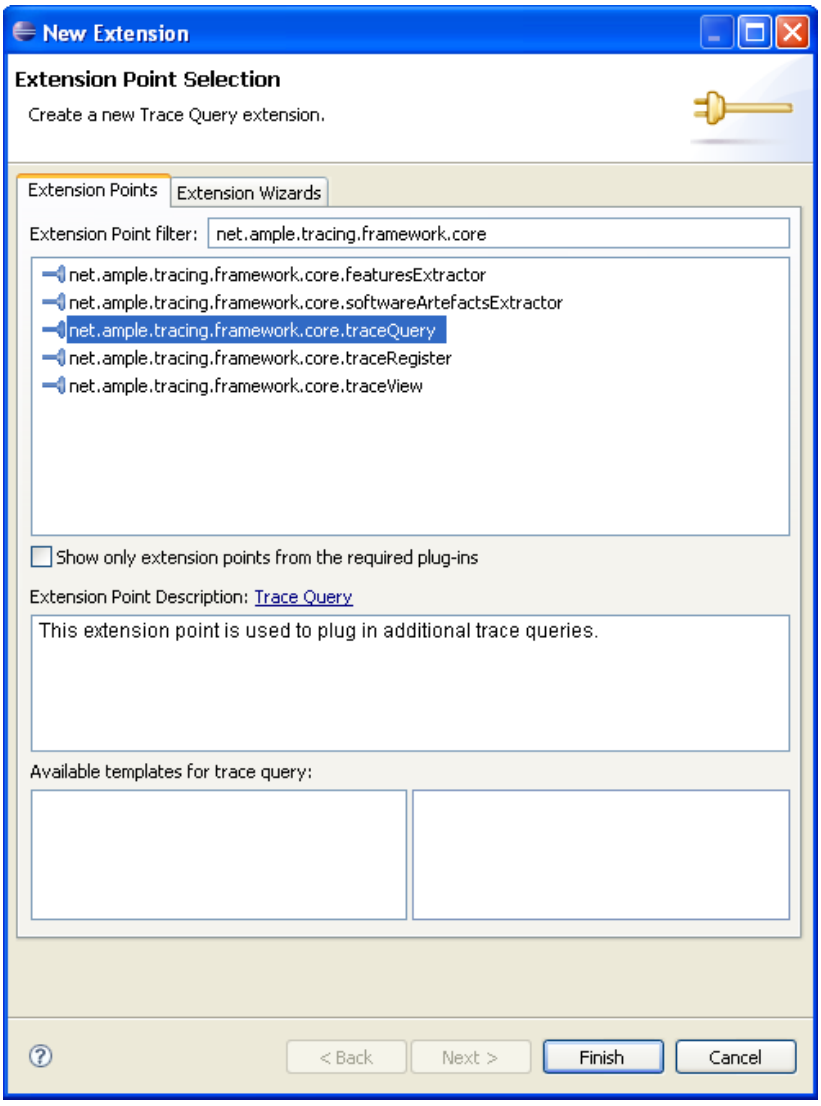
Trace Query Instance

The definition of the *ITraceQuery* in the framework description, showed that there was an abstract method `submitQuery()` which must be implemented. This is the method that will perform the desired query, returning a list of trace links form the ATF repository. Different types of queries can be implemented by instantiating this hotspot with the desired query.

For this trace query example we are going to create a query that extracts all the trace links, between features and use case artifacts, stored in a repository.

Start by creating a new Plug-in Project named *net.ample.tracing.simpleTraceQuery*. Put the same configurations as for the trace register instance.

After creating the new Plug-in Project, go to the Extensions tab and add a new extension. Choose the *net.ample.tracing.framework.core.traceQuery* extension and press *Finish*.



Now we must complete the configuration of our new extension in order for it to behave as expected. Expand the extension point tree, and go to the *extractor* element. Each field in this element has the following meaning:

Property	Description
id	A unique name that will be used to reference this trace query.
name	A translatable name that will be used for presenting this trace query in the UI.
class	Plug-ins that want to extend this extension point must implement *net.ample.tracing.framework.core.tracequery.ITraceQuery* interface.

Fill in each of these fields with the following values (without the quotes):

id = “net.ample.tracing.simpleTraceQuery.query”

name = “Simple Trace Query”

class = “net.ample.tracing.simpleTraceQuery.SimpleTraceQuery”

If you want, you can also add a description for this extension point in the corresponding element. Once we have defined all the attributes for our extension point, we are now going to create a Java class that implements the required interface, i.e. *ITraceQuery* (check Figure 1).

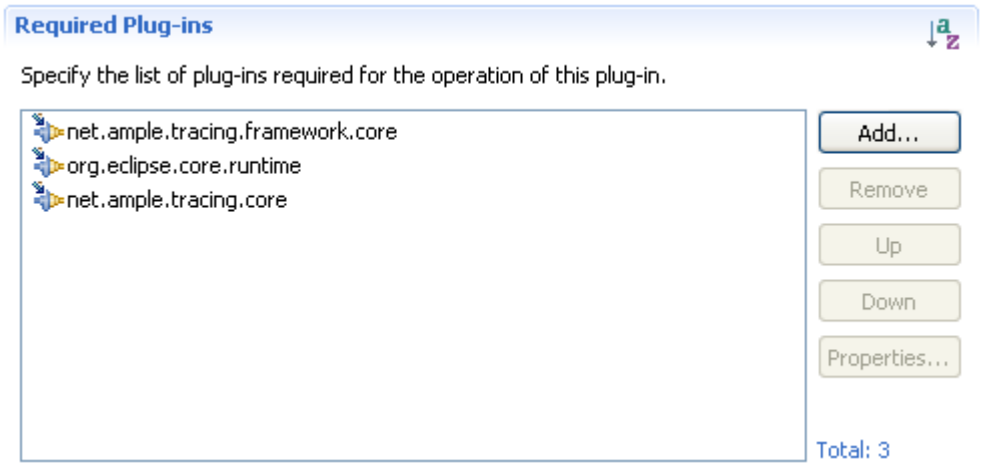
Follow the steps described in the features extractor instance, and create the new class SimpleTraceQuery.

To implement our *SimpleTraceQuery* class, and the method `submitQuery()`just copy the code shown below.

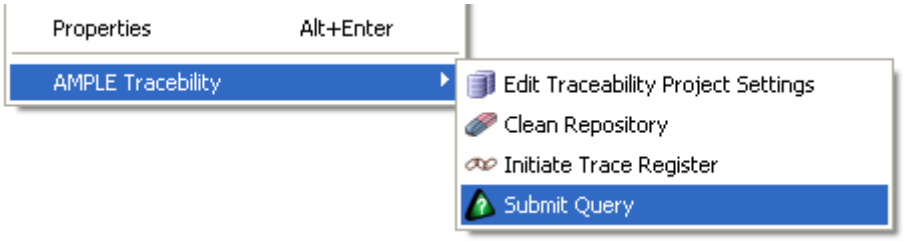
```
package net.ample.tracing.simpleTraceQuery;

import java.util.ArrayList;
import java.util.List;
import net.ample.tracing.core.QueryManager;
import net.ample.tracing.core.TraceLink;
import net.ample.tracing.core.TraceableArtefact;
import net.ample.tracing.core.TraceableArtefactType;
import net.ample.tracing.core.query.Constraints;
import net.ample.tracing.core.query.Query;
import net.ample.tracing.framework.core.tracequery.AbstractTraceQuery;
import net.ample.tracing.framework.core.tracequery.ITraceQuery;
import net.ample.tracing.framework.core.utils.MissingRepositoryException;
import org.eclipse.core.runtime.CoreException;

public class SimpleTraceQuery extends AbstractTraceQuery implements
ITraceQuery {

  private QueryManager queryManager;
```

```
  public SimpleTraceQuery() {
  }

  @Override
  public List<TraceLink> submitQuery() throws MissingRepositoryException,
CoreException {
    queryManager = getQueryManagerInstance();
    List<TraceLink> results = new ArrayList<TraceLink>();
    Query<TraceableArtefact> query = queryManager.queryOnArtefacts();
    query.add(Constraints.type(getArtefactTypeByName("Feature")));
    List<TraceableArtefact> queryResult = query.execute();
    for (int i=0; i<queryResult.size(); i++) {
      TraceableArtefact feature = queryResult.get(i);
      List<TraceLink> outgoingLinks = feature.getOutgoingLinks();
      results.addAll(outgoingLinks);
    }
    return results;
  }

  private TraceableArtefactType getArtefactTypeByName(String name) {
    Query<TraceableArtefactType> query =
queryManager.queryOnArtefactTypes();
    query.add(Constraints.name(name));
    return query.executeUnique();
  }
}
```

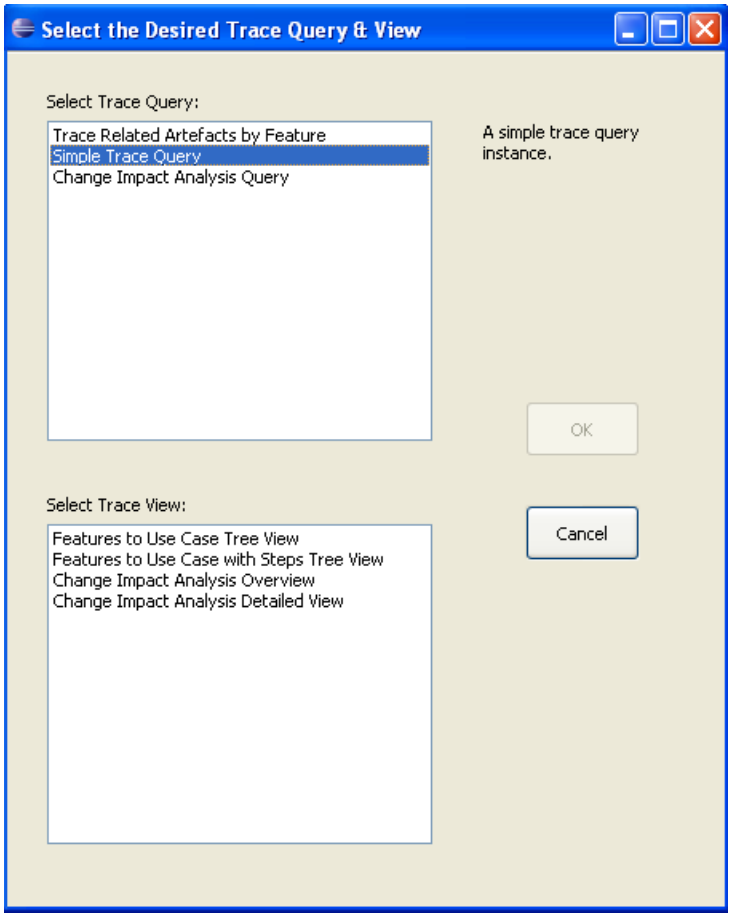
For this code to work properly, we need to add some missing dependencies. The ones that are needed are shown in following figure.



Finally, follow the steps mentioned previously to define your exported packages, organize your manifest files, etc. when you are done export the new trace query into a JAR file (same as for the features extractor instance), and copy it to your “plugins” directory inside your Eclipse installation directory. Restart Eclipse, and right click your traceability project file and choose the menu “Submit Query”.



The new trace query extension is automatically displayed and ready to be used.



Trace View Instance

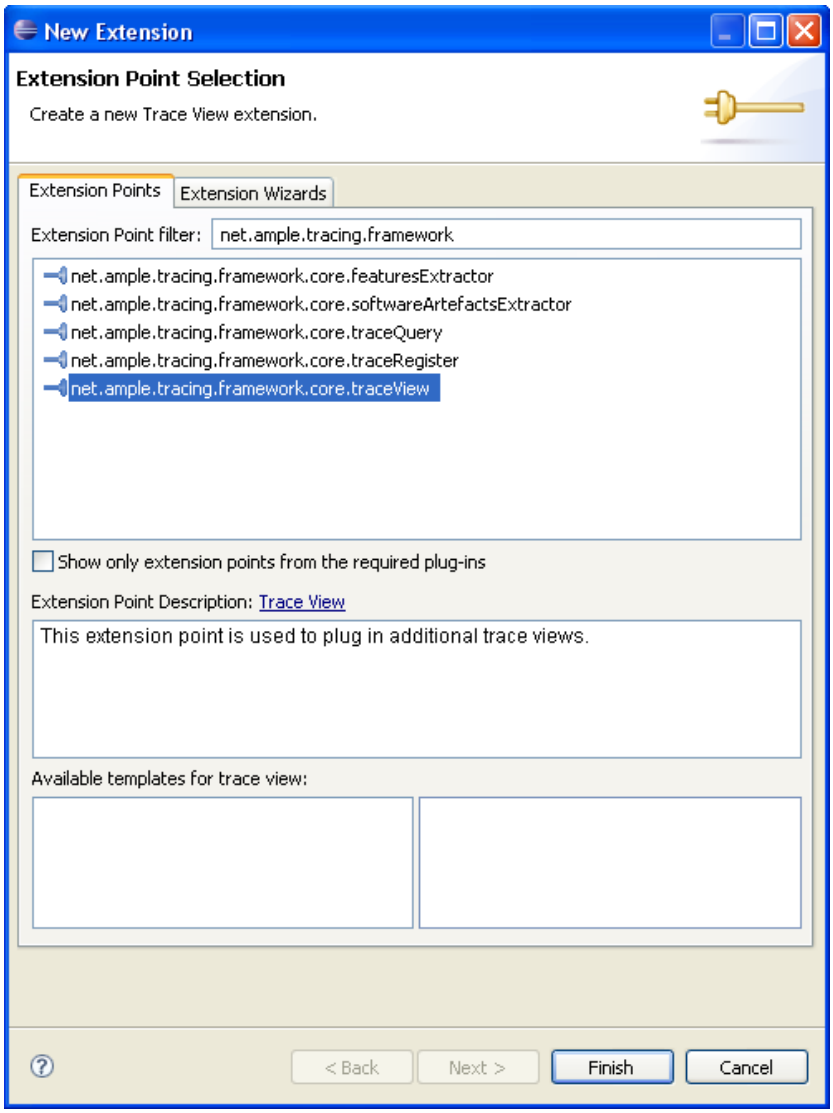
The definition of the *ITraceView* in the framework description, showed that there was an abstract method `showResults(List<TraceLink> results)` which must be implemented. This is the method that will present, to the user, the list of trace links returned by a query execution. Different types of trace views can be implemented by instantiating this hotspot with the desired view.

As with the previous examples, we are going to create a very simple trace view that is not meant to be used in real SPL development. We will be creating a view that opens a window which displays the list of trace links passed as argument to the method `showResults`. For each trace link in the list, we will create a string with the

source and target artifacts of the trace link and add each string to a list of strings (each string represents a link). Finally, that list will be shown inside a window.

Start by creating the new Plug-in Project as mentioned previously and name it *net.ample.tracing.simpleTraceView*. The rest of the options, are the same as the ones used for the trace register instance.

After creating the new Plug-in Project, go to the Extensions tab and add a new extension. Choose the *net.ample.tracing.framework.core.traceQuery* extension and press *Finish*.



Now we must complete the configuration of our new extension in order for it to behave as expected. Expand the extension point tree, and go to the *extractor* element. Each field in this element has the following meaning:

Property	Description
id	A unique name that will be used to reference this trace view.
name	A translatable name that will be used for presenting this trace view in the UI.
class	Plug-ins that want to extend this extension point must implement *net.ample.tracing.framework.core.traceview.ITraceView* interface.

Fill in each of these fields with the following values (without the quotes):

id = “net.ample.tracing.simpleTraceView.view”

name = “Simple Trace View”

class = “net.ample.tracing.simpleTraceView.SimpleTraceView”

If you want, you can also add a description for this extension point in the corresponding element. Once we have defined all the attributes for our extension point, we are now going to create a Java class that implements the required interface, i.e. *ITraceView* (check Figure 1).

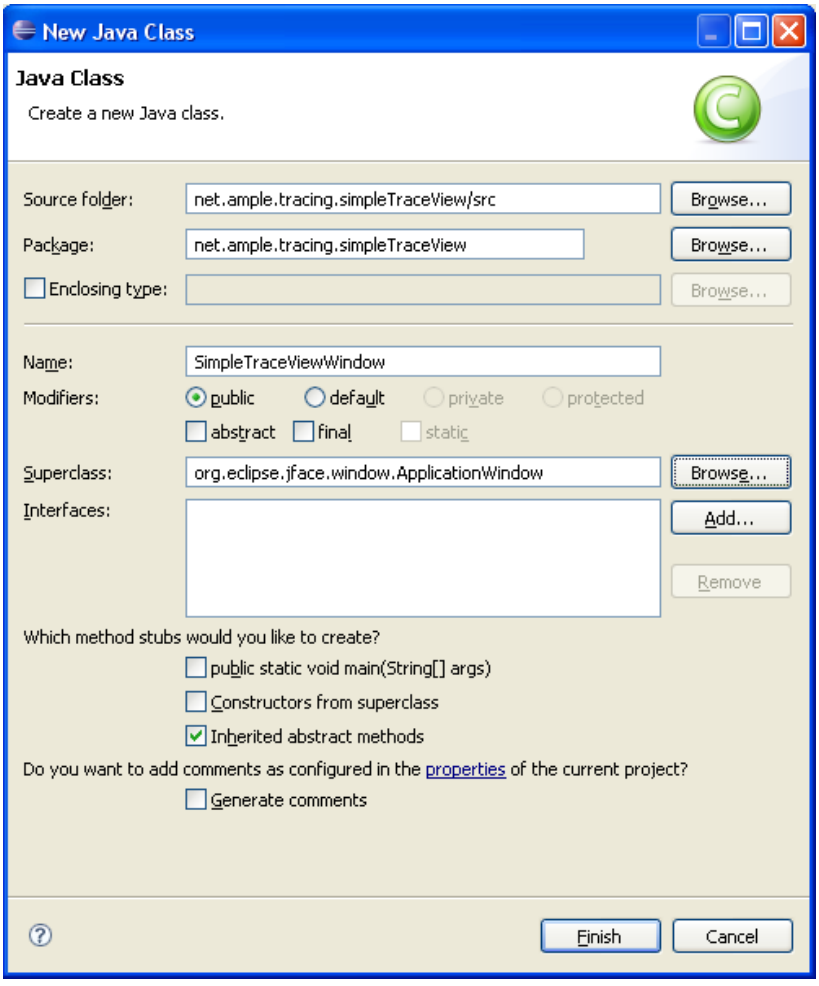
Follow the steps described in the features extractor instance, and create the new class SimpleTraceView. Use The code shown below to implement the *SimpleTraceView* class which includes the method `showResults(List<TraceLink> results)` responsible for presenting the results to the user.

```
package net.ample.tracing.simpleTraceView;

import java.util.ArrayList;
import java.util.List;
import net.ample.tracing.core.TraceLink;
import net.ample.tracing.core.TraceableArtefact;
import net.ample.tracing.framework.core.traceview.ITraceView;
import org.eclipse.jface.window.Window;

public class SimpleTraceView implements ITraceView {

  public SimpleTraceView() {
  }

  @Override
  public void showResults(List<TraceLink> results) {
    List<String> resultsList = new ArrayList<String>();
    for(int i=0; i<results.size(); i++) {
      List<TraceableArtefact> sources = results.get(i).getSources();
      List<TraceableArtefact> targets = results.get(i).getTargets();
      for(int j=0; j<sources.size(); j++) {
        for(int k=0; k<targets.size(); k++) {
          String link = "'" + sources.get(j).getName() + "' links to '" +
targets.get(k).getName() + "'";
          resultsList.add(link);
        }
```

```
      }
    }
    SimpleTraceViewWindow window = new SimpleTraceViewWindow(resultsList);
    openWindow(window);
  }

  private static void openWindow(Window window) {
    window.setBlockOnOpen(true);
    window.open();
  }
}
```

The code in *SimpleTraceView* class processes the list of results to create a list of strings containing all the sources and targets of each link. However, to show them in a window, we will create another class called *SimpleTraceViewWindow* which will launch a window with the desired contents.

To do this, just choose *File* > *New* > *Class* and fill in the remaining options as follows.

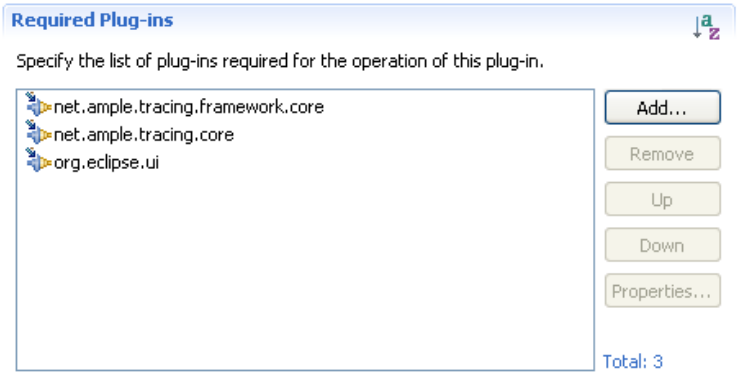


And to implement this window just paste the code shown below in the *SimpleTraceViewWindow* class.

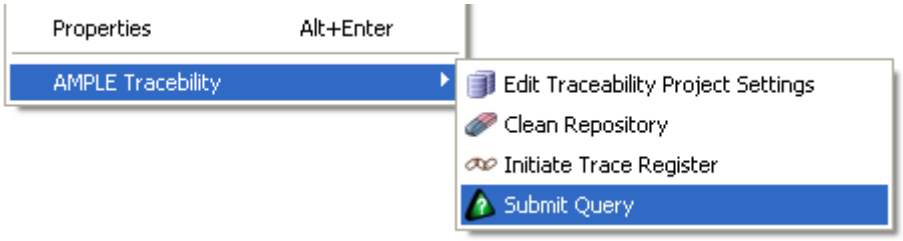
```
package net.ample.tracing.simpleTraceView;

import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
```

```
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.List;
import org.eclipse.swt.widgets.Shell;

public class SimpleTraceViewWindow extends ApplicationWindow {

  private Composite container;
  private java.util.List<String> links;

  public SimpleTraceViewWindow(java.util.List<String> links) {
    super(null);
    this.links = links;
  }

  @Override
  protected Control createContents(Composite parent) {
    this.container = new Composite(parent, SWT.NONE);

    final Label linksReturnedByLabel = new Label(container,
SWT.NONE);
    linksReturnedByLabel.setText("Links returned by query
execution:");
    linksReturnedByLabel.setBounds(30, 15, 222, 13);

    final List list = new List(container, SWT.V_SCROLL | SWT.H_SCROLL
| SWT.BORDER);
    list.setBounds(30, 45, 329, 584);
    for(int i=0; i<links.size(); i++) {
      list.add(links.get(i));
    }
    return container;
  }

  @Override
  protected Point getInitialSize() {
    return new Point(400, 700);
  }

  @Override
  protected void configureShell(Shell newShell) {
    super.configureShell(newShell);
    newShell.setText("Simple Trace View");
  }
}
```

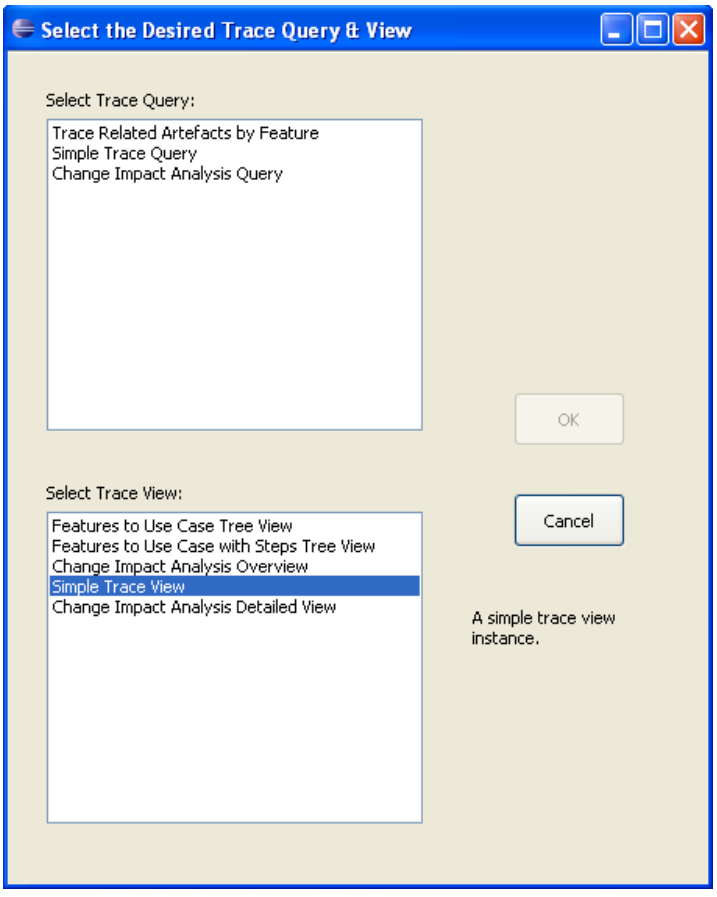
All that is missing for this code to compile work is adding the necessary dependencies. The ones that are needed are shown in following figure.



Finally, follow the steps mentioned previously to define your exported packages, organize your manifest files, etc. When you are done export the new trace query into a JAR file (same as for the features extractor instance), and copy it to your "plugins" directory inside your Eclipse installation directory. Restart Eclipse, and right click your traceability project file and choose the menu "Submit Query".



The new trace view extension is automatically displayed and ready to be used.



APPENDIXES

Appendix I - Mobile Photo Variability Model

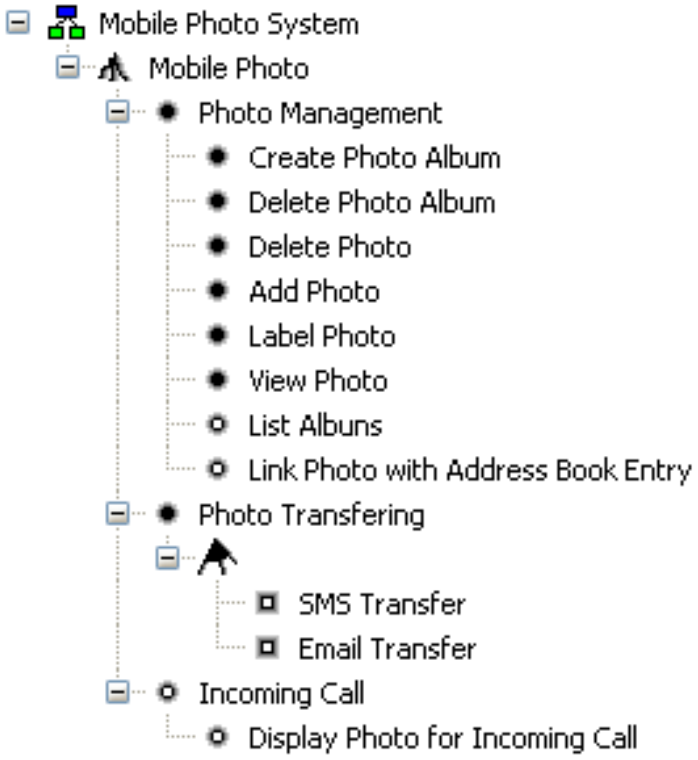


Figure 3 – Mobile Photo Feature Model

Appendix II - Mobile Photo Use Case Model

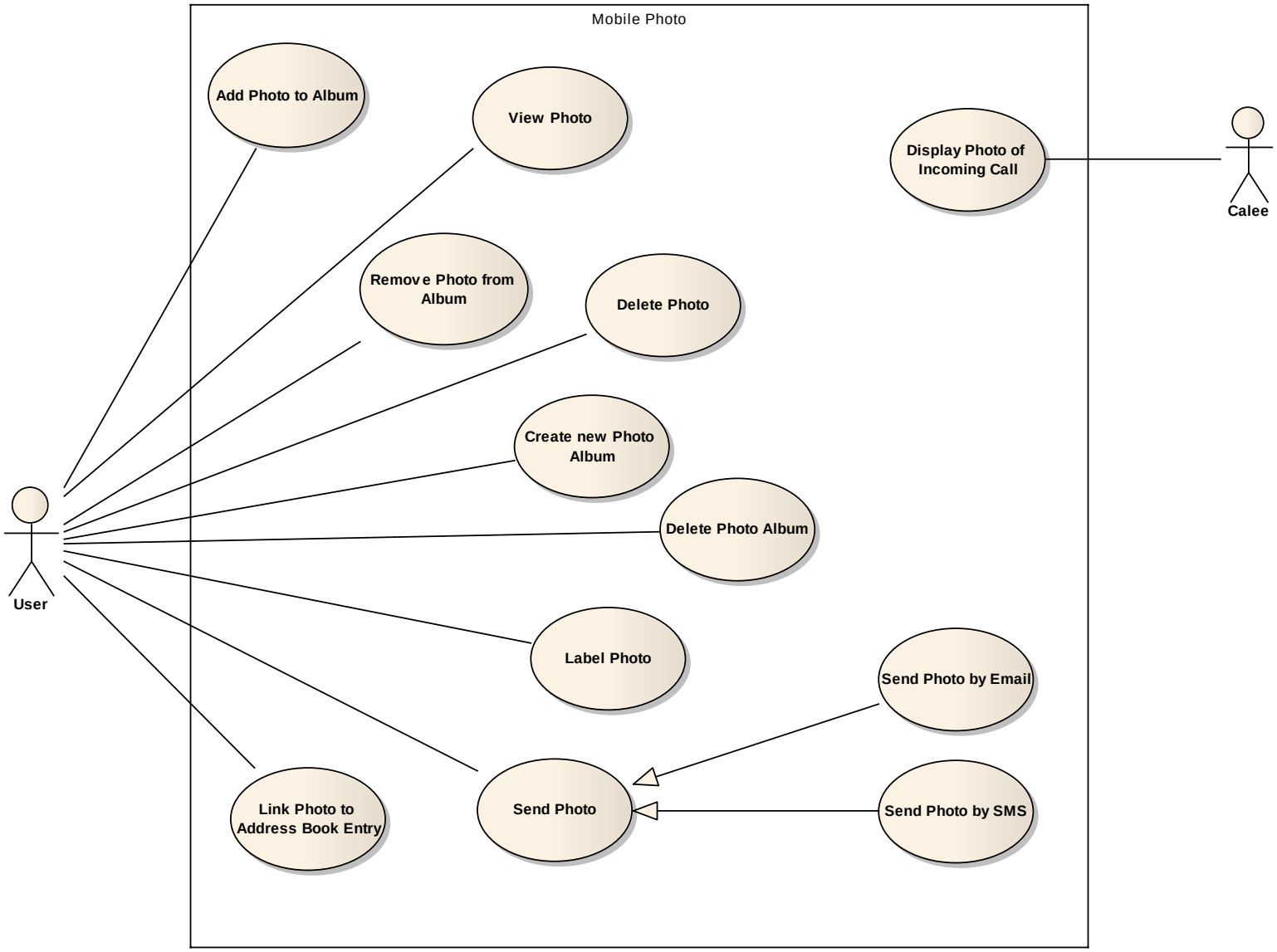


Figure 4 – Mobile Photo Use Case Model

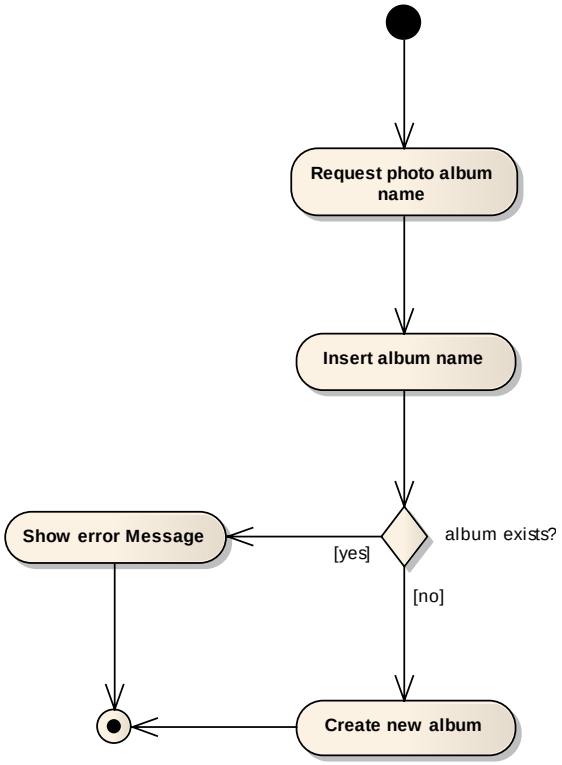


Figure 5 - Create new Photo Album Steps

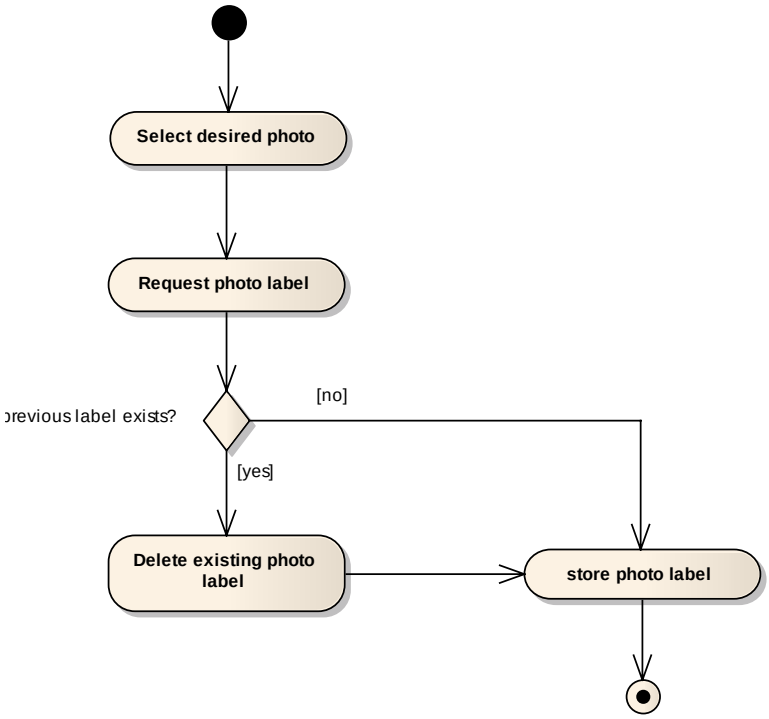


Figure 6 - Label Photo Steps

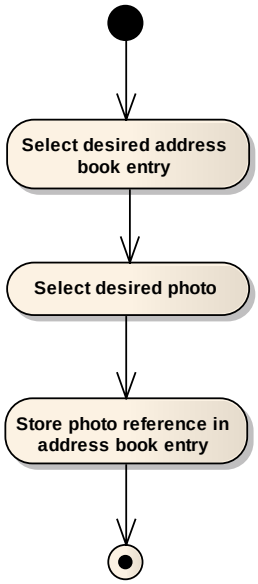


Figure 7 - Link Photo to Address Book Entry Steps

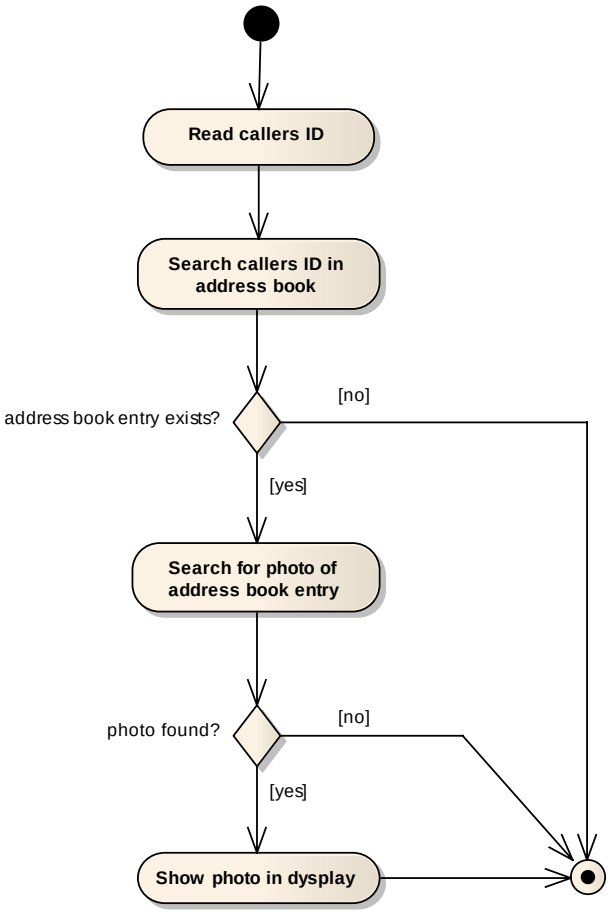


Figure 8 - Display Photo of Incoming Call Steps

Appendix III - Extension Points Reference

Trace Register

Identifier: net.ample.tracing.framework.core.traceRegister

Since: 0.1.0

Description: This extension point is used to plug in additional trace registers for establishing trace links between SPL artefacts.

Configuration Markup:

```
<!ELEMENT extension (register)+>
<!ATTLIST extension
    point       CDATA #REQUIRED
    id          CDATA #IMPLIED
    name        CDATA #IMPLIED>

<!ELEMENT register (description)>
<!ATTLIST register
    id          CDATA #REQUIRED
    name        CDATA #REQUIRED
    class       CDATA #REQUIRED
```

- **id** - a unique name that will be used to reference this trace register.
- **name** - a translatable name that will be used for presenting this trace register in the UI.
- **class** - Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.traceregister.ITraceRegister` interface.

```
<!ELEMENT description (#PCDATA)>
```

Examples: The following is an example of the extension point usage:

```
<extension point="net.ample.tracing.framework.core.traceRegister">
   <register
      id="net.ample.tracing.sample_register"
      name="Sample Trace Register"
      class="net.ample.tracing.SampleRegister">
        <description>some description.</description>
   </register>
</extension>
```

API Information: Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.traceregister.ITraceRegister` interface.
Supplied Implementation: Traceability Framework Plug-in provides a default implementation of a trace register.

Trace Query

Identifier: net.ample.tracing.framework.core.traceQuery

Since: 0.1.0

Description: This extension point is used to plug in additional trace queries.

Configuration Markup:

```
<!ELEMENT extension (query)+>
<!ATTLIST extension
    point       CDATA #REQUIRED
    id          CDATA #IMPLIED
    name        CDATA #IMPLIED>

<!ELEMENT query (description)>
<!ATTLIST query
    id          CDATA #REQUIRED
    name        CDATA #REQUIRED
    class       CDATA #REQUIRED
```

- **id** - a unique name that will be used to reference this trace query.
- **name** - a translatable name that will be used for presenting this trace query in the UI.
- **class** - Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.tracequery.ITraceQuery` interface.

```
<!ELEMENT description (#PCDATA)>
```

Examples: The following is an example of the extension point usage:

```
<extension point="net.ample.tracing.framework.core.traceQuery">
   <query
      id="net.ample.tracing.sample_query"
      name="Sample Trace Query"
      class="net.ample.tracing.SampleQuery">
        <description>some description.</description>
   </query>
</extension>
```

API Information: Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.tracequery.ITraceQuery` interface.

Supplied Implementation: Traceability Framework Plug-in provides a default implementation of a trace query.

Trace View

Identifier: net.ample.tracing.framework.core.traceView

Since: 0.1.0

Description: This extension point is used to plug in additional trace views.

Configuration Markup:

```
<!ELEMENT extension (view)+>
<!ATTLIST extension
    point       CDATA #REQUIRED
    id          CDATA #IMPLIED
    name        CDATA #IMPLIED>

<!ELEMENT view (description)>
<!ATTLIST view
    id          CDATA #REQUIRED
    name        CDATA #REQUIRED
    class       CDATA #REQUIRED
```

- **id** - a unique name that will be used to reference this trace view.
- **name** - a translatable name that will be used for presenting this trace view in the UI.
- **class** - Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.traceview.ITraceView` interface.

```
<!ELEMENT description (#PCDATA)>
```

Examples: The following is an example of the extension point usage:

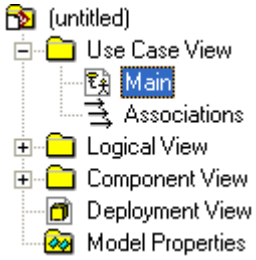
```
<extension point="net.ample.tracing.framework.core.traceView">
   <view
      id="net.ample.tracing.sample_view"
      name="Sample Trace View"
      class="net.ample.tracing.SampleView">
        <description>some description.</description>
   </view>
</extension>
```

API Information: Plug-ins that want to extend this extension point must implement `net.ample.tracing.framework.core.traceview.ITraceView` interface.

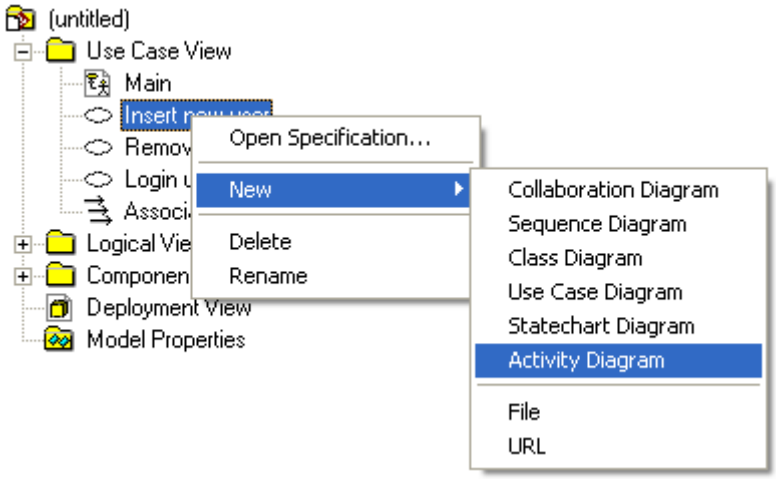
Supplied Implementation: Traceability Framework Plug-in provides a default implementation of a trace view.

Appendix IV – Rational Rose Use Case Modeling

To create use case models that will be correctly imported by the framework, start by creating a new project and then open the Use Case View (Main).



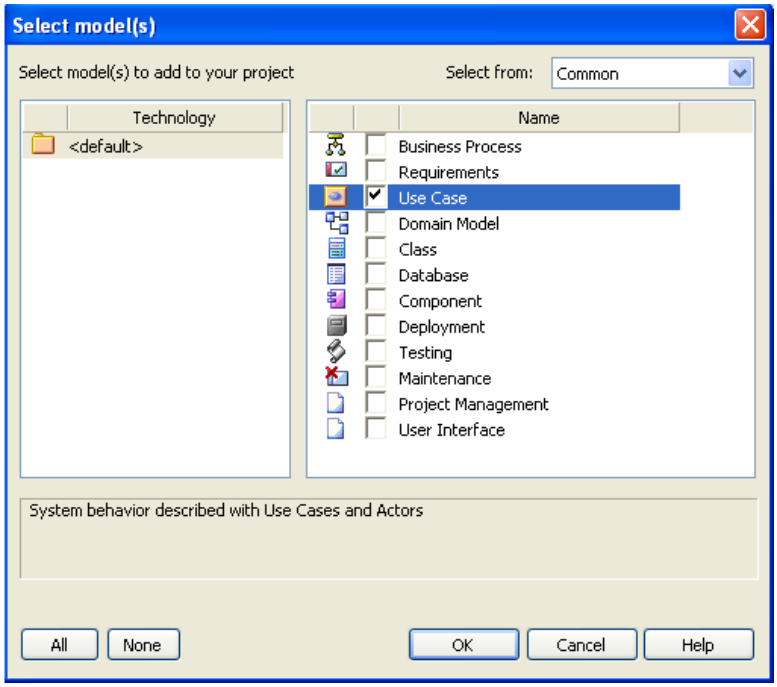
Then create the use case model elements that you desire.
To create use case steps, use activity diagrams inside each use case, and model the steps of the use case using the newly created activity diagram.



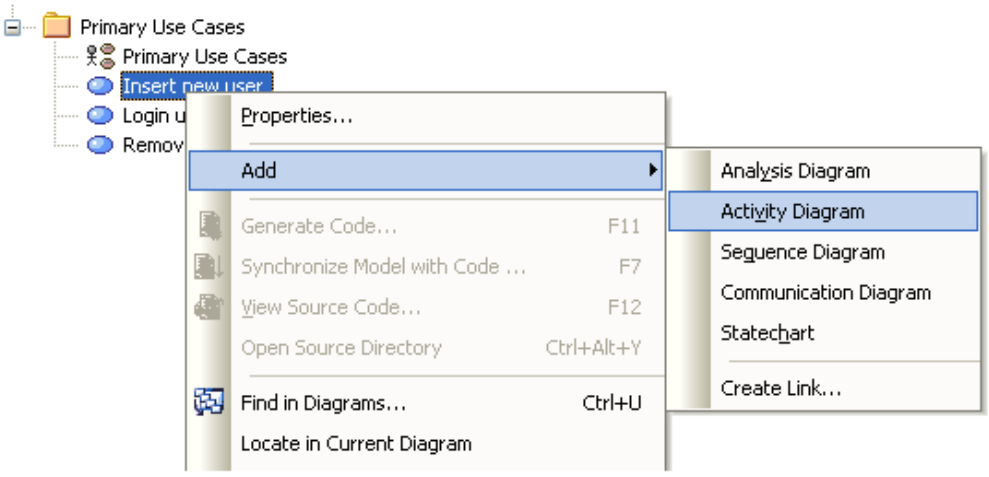
Once all the elements have been modeled, just save you project and it is ready to be imported in the Traceability Framework.

Appendix V – Enterprise Architect Use Case Modeling

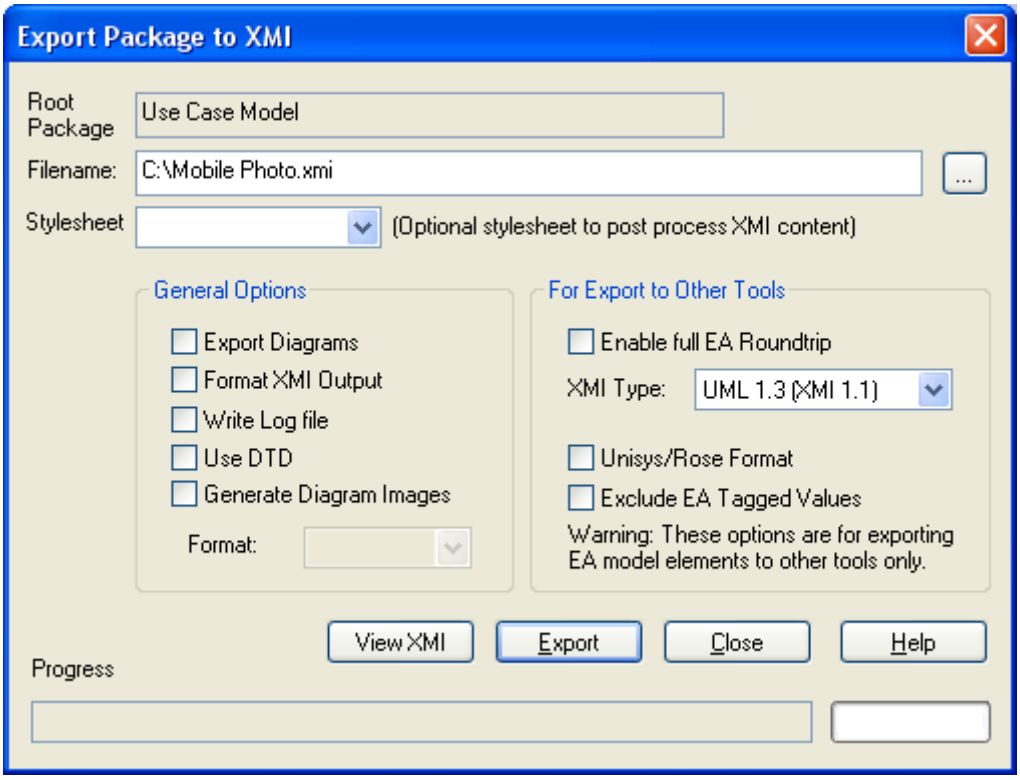
To create use case models that will be correctly imported by the framework, start by creating a new project and select “Use Case”.



Then create the use case model elements that you desire.
To create use case steps, use activity diagrams inside each use case, and model the steps of the use case using the newly created activity diagram.



Now, we must export our model in XMI format. Go to *Project* > *Import/Export* > *Export Package to XMI…* and them insert the name of the file you which to export to (the extension of the file must be XMI), choose **UML 1.3 (XMI 1.1)** as the XMI type and unselect all the other options. Finally press *Export* and we can use the exported file to extract the elements to our framework.


Export Package to XMI
Root Package
Use Case Model
Filename:
C:\Mobile Photo.xmi
...
Stylesheet
(Optional stylesheet to post process XMI content)
General Options
Export Diagrams
Format XMI Output
Write Log file
Use DTD
Generate Diagram Images
Format:
For Export to Other Tools
Enable full EA Roundtrip
XMI Type:
UML 1.3 (XMI 1.1)
Unisys/Rose Format
Exclude EA Tagged Values
Warning: These options are for exporting EA model elements to other tools only.
View XMI
Export
Close
Help
Progress