Fine-Grained Management of Software Artefacts

Fausto Fasano

April 2007

Ph.D Course Chairman
Prof.ssa Margherita Napoli

Supervisor
Prof. Andrea De Lucia
To my women

“Every day I remind myself that my inner and outer life are based on the labors of other men, living and dead, and that I must exert myself in order to give in the same measure as I have received and am still receiving”

Albert Einstein
Abstract

In this thesis I present ADAMS (ADvanced Artefact Management System), a Web-based system that integrates project management features such as resource allocation and process control and artefact management features, such as coordination of cooperative workers and artefact versioning, as well as context-awareness and artefact traceability features.

Rather than defining the control and data flow between activities, like in most Process Support Systems, software processes in ADAMS are modelled through the produced artefacts and the relations between them. Such relations are modelled in terms of traceability links (dependencies). Maintaining traceability links between artefacts supports management of changes during incremental and iterative software development in a flexible way.

ADAMS allows the software engineer to specify traceability links between pairs of artefacts. A traceability link represents a relationship between an independent artefact (or source artefact) and a dependent artefact (or target artefact). Traceability links can be organised in a traceability graph where the artefacts are identified by nodes and the traceability links are represented with edges of the graph. Within a traceability graph we identify traceability paths, i.e. sets of artefacts connected by traceability links. Obviously, an artefact along a traceability path could be impacted by each artefact appearing in the part of the path preceding it as well as it could impact on each artefact appearing in the part of the path following it.

Artefacts in ADAMS can either be an atomic entity (associated to one or more files) or they can be composed of an arbitrary number of atomic and further composed artefacts. In fact, most of the documents produced during software development have a well-defined hierarchical structure. ADAMS provides a fine-grained management of software artefacts, in which each single component of a
document is managed as a different entity and versioning is provided for both the atomic entity changed and the entities it is contained in. Fine-grained management of software artifacts gives the possibility to define more detailed traceability links. In particular, the software engineer is enabled to specify traceability links between entities within the same document, as well as between parts of two different documents, resulting in a more precise impact analysis. Moreover, a fine-grained artifact management reduces the likelihood of conflicts arising when different software engineers apply changes to the same document. Indeed, in case the document is decomposed into sub-artifacts, a conflict occurs only if changes concern the same sub-artifact. Moreover, since an artifact is decomposed into a hierarchy of simpler objects eventually annotated with text, the definition of more precise access policies to specific parts of the artifact is also enabled, thus allowing a finer task allocation and responsibility identification. This is particularly important in a distributed environment, where a work breakdown structure is usually defined to decompose the project components into manageable tasks assigned to distributed development teams.

ADAMS provides a full feature environment supporting the software engineer during all the phases of the software lifecycle paying a special attention to the coordination problems occurring in cooperative development environments. Besides the support provided by typical features of a configuration management system, collaboration is also supported by synchronous and asynchronous tools. Examples of asynchronous collaborative tools are the internal e-mail tool and the rationale management tool. In particular, rationale elements are treated as artifact types and relations among them as traceability links. Rationale elements can also be related to artifacts developed within a project. Examples of synchronous collaboration tools are the internal chat and an UML collaborative editor that allows developers to access and modify the same diagram at the same time, thus allowing a distributed team to discuss and model the system directly within ADAMS, thus maintaining the versioning for each model element as well as for the information shared during the meeting.

Most of the problems experienced by people working concurrently on the same
artefacts, especially in distributed settings, are related to limited communication and coordination that leads to context awareness problems. In order to face up these problems, ADAMS provides an event-based notification mechanism that allows the software engineer working on an artefact to be notified when something relevant occurs on the artefact. This provides a solution to the isolation problem for resources working on the same artefact in different workspaces: it is possible to identify potential conflicts before they occur, since interested resources are notified as soon as an artefact is checked-out and potentially before substantial modifications are applied to it. Events can also be propagated through the traceability layer, so that the developers can be aware of changes applied to artefacts their work depends on. This can contribute to anticipate potential changes thus reducing expensive later modifications. Event propagation also reduces the complexity of subscribing several events for notification, avoids worthless redundancies, and prevents from forgetting indirect but essential subscriptions.

In the last two years, ADAMS has been used as artefact management system in the Software Engineering courses of the Computer Science program at the University of Salerno (Italy). At the end of the projects, the students evaluated ADAMS through a questionnaire. Each question refers to an attribute of four quality characteristics of interest, namely functionality, usability, robustness, and performance. In this thesis a discussion on the collected evaluation questionnaires is also presented.
Abstract
I would like to thank the people who helped me to carry out my work. I want to express my gratitude to Genny Tortora, who gave me the stimulus to start looking outside my office window and desire to explore the open, complex, boundless, and challenging world of research. Above all, I’d like to thank my advisor Andrea De Lucia, who took me in hand tracing the route for my first steps, but also letting me free to lose and find the way back. The findings of these years are fruits caught from his knowledge tree. I want to specially thank Rocco Oliveto, a good friend, an honest colleague, a harsh critic, and a hard benchmark. An excellent combination. I also want to thank all my colleagues and friends for their invaluable support: Giuseppe, Carmine, and Michele (the “Eastern Island”), obrigado! And last but not least, a final word of thanks is due to my family. My parents for their constant and invaluable support; my wife Mady, for the time I took away from us and the way she did not make me feel guilty for that, and Elisa for so many reasons that would require a thesis to make them clear.
Contents

Abstract ................................................................................................................................. v
Acknowledgement ................................................................................................................. ix
Contents ................................................................................................................................. 11
1 Introduction ......................................................................................................................... 15
  1.1 ADAMS Evaluation .................................................................................................. 19
  1.2 Thesis organisation ................................................................................................. 20
2 Related Work ..................................................................................................................... 21
  2.1 Hierarchic Versioning Management Systems ....................................................... 21
      2.1.1 COOP/Orm and CoEd .................................................................................. 21
      2.1.2 CoVer ......................................................................................................... 24
      2.1.3 Stellation ..................................................................................................... 25
      2.1.4 SubCM ........................................................................................................ 27
  2.2 Traceability Management Systems ......................................................................... 29
      2.2.1 Molhado ...................................................................................................... 29
      2.2.2 RDD-100 ..................................................................................................... 33
      2.2.3 TOOR ......................................................................................................... 34
      2.2.4 Ophelia ....................................................................................................... 35
      2.2.5 ArchEvol ..................................................................................................... 36
      2.2.6 OSCAR ....................................................................................................... 38
      2.2.7 MILOS ........................................................................................................ 40
      2.2.8 APER ......................................................................................................... 41
      2.2.9 Event Based Traceability ............................................................................ 42
      2.2.10 PROSYT .................................................................................................... 44
      2.2.11 Sysiphus .................................................................................................... 44
3 ADAMS Overview ............................................................................................................. 47
  3.1 Resource Management Subsystem ......................................................................... 51
  3.2 Project Management Subsystem .............................................................................. 51
  3.3 Quality Management Subsystem ............................................................................. 54
3.4 Artefact Management Subsystem ................................................................. 55
3.5 Cooperative development subsystem .......................................................... 56
4 Fine-grained management of software artefacts .................................................. 59
4.1 Artefact composition ....................................................................................... 61
4.1.1 Rebuild of a composed artefact ................................................................. 65
4.1.2 Visualisation of composed artefacts ............................................................ 68
4.2 Traceability management ................................................................................ 71
4.2.1 Traceability links recovery .......................................................................... 74
4.3 Versioning management in ADAMS ............................................................... 76
4.3.1 Versioning of composed artefacts ............................................................... 80
5 Collaborative development support .................................................................... 83
5.1 Concurrent development management ............................................................ 83
5.2 Synchronous collaborative UML diagrams editing .......................................... 85
5.3 Event notification management ...................................................................... 92
5.3.1 Event notification propagation .................................................................... 96
5.4 Review management ....................................................................................... 98
5.5 Rationale management .................................................................................. 102
6 Evaluation and Experience ............................................................................... 107
6.1 Comparison with other tools ........................................................................... 107
6.1.1 Evaluated characteristics and tools ............................................................... 107
6.1.2 Tool comparison ......................................................................................... 109
6.2 ADAMS Experience ....................................................................................... 112
6.3 Survey Questionnaire .................................................................................... 114
6.4 Discussion ....................................................................................................... 122
7 Conclusions and future work ........................................................................... 127
A. Survey questionnaires ....................................................................................... 131
C1. Functionality .................................................................................................... 132
7.1.1 A1. Usefulness ............................................................................................. 132
7.1.2 A1. Adequacy .............................................................................................. 137
7.1.3 A3. User Considerations ............................................................................ 141
C2. Usability .......................................................................................................... 145
7.1.4 A1. Intuitiveness .......................................................................................... 145
7.1.5 A2. User Interface ....................................................................................... 148
C3. Reliability ........................................................................................................ 152
7.1.6 A1. Robustness ........................................................................................... 152
7.1.7 A2. Error Recovery ..................................................................................... 153
C4. Performances .................................................................................................................. 154
C5. Quantitative Data ............................................................................................................ 157
Figure Index ............................................................................................................................... 163
Glossary ..................................................................................................................................... 166
Publications ............................................................................................................................... 170
Bibliography .............................................................................................................................. 172
Chapter 1

1 Introduction

In the last decade a lot of research effort has been devoted to the development of methodologies and technologies supporting coordination and collaboration of distributed software engineering teams. Examples are Computer Supported Cooperative Work (CSCW) and groupware, workflow management, and configuration management.

Configuration Management (CM) is among the others mostly used in software engineering projects to face with coordination problems. CM tools (see e.g., [11], [22], [27], [63], [73]) help to coordinate the activities of developers, by providing capabilities that either avoid parallel development altogether (e.g., locking) or assist in resolving conflicts (e.g., merging). Independently of the adopted model (Checkout/Checkin, Composition, Long Transaction, ChangeSet), existing CM systems are based on the workspace concept, representing the work environment of each user [68]. The adoption of such separate areas causes a lack of context-
awareness, as a developer is informed of work made by others on the artefacts\(^1\) he/she is working on or on related artefacts, only after these have been checked-in, thereby significantly delaying the discovery of potential problems.

Process Support Systems (PSSs) [6], [24], [55] including Workflow Management Systems (WfMSs) [39], [40], [81] and Process-centered Software Engineering Environments (PSEEs) [2], [7], [44] represent a different research area aiming at supporting the coordination of software development activities through process modelling and enactment. Despite the advances made in the field, most of the solutions proposed have not gained wide acceptance in the industry because the Process Description Languages (PDLs) they propose for the modelling of business processes are too complicated to understand and manipulate for many practitioners. Most PSSs are activity-based and model software processes in a top down manner, focusing on the specification of the control and data flow between activities. Most of them do not support the deviations from the process model when unforeseen situations (frequently) happen and even when such a support is provided, it is difficult to model and manage in advance every possible exceptional situations. Also, the production of an artefact is seen as the result of the execution of an activity and often there is lack of integration with configuration management systems [6].

With respect to CM tools, most recent PSSs provide a greater support to context-awareness, by integrating communication tools and notification mechanisms to make aware developers about events occurring within activities [6], [24], [26], [44], [55].

Software artefact traceability is the ability to describe and follow the life of an artefact (requirements, code, tests, models, reports, plans, etc.) developed during the

\(^1\) A **software artefact** is a deliverable or an outcome of a software process. More generally, any piece of software developed and used during software development and maintenance can be referred to as a software artefact. Examples are requirements specifications, architecture and design models, source and executable code (programs), configuration directives, test data, test scripts, process models, project plans, project documentation. In the following, the terms software artefact and artefact will be used as synonyms anytime the meaning of the work is cleared by the context.
software lifecycle in both forward and backward directions (e.g., from requirements to the modules of the software architecture and the code components implementing them and vice-versa) [42]. Traceability can provide important insights into system development and evolution assisting in both top-down and bottom-up program comprehension, impact analysis, and reuse of existing software, thus giving essential support in understanding the relationships existing within and across software requirements, design, and implementation [62].

Regardless of its importance, the support for traceability in contemporary software engineering environments and tools is not satisfactory. This inadequate traceability is one of the main factors that contribute to project over-runs and failures [32]. Although several research and commercial tools are available that support traceability between artefacts [21], [46], [50], [64], [65], [66], [74] the main drawback of these tools is the lack of automatic or semi-automatic traceability link generation and maintenance. In general, such tools require the user to assign keywords to all the documents prior to tracing and most of them do not provide support for easily retracing new versions of documents. This results in the need for a costly activity of manual detection and maintenance of the traceability links that may have to be done frequently due to the iterative nature of software development [21].

Both PSSs and CM tools generally do not offer an adequate support to artefact traceability and, as a consequence, handling changes is difficult. CM tools mainly enable versioning of artefacts, but traceability information among different artefacts is lacking and when supported, the traceability infrastructure fails during the system evolution [21]. In PSSs dependencies between artefacts can be derived from the data flow links between activities but relationships between the artefacts produced during the software development process is not directly stored and maintained.

Moreover, most of the documents produced during the software development have a well-defined hierarchic structure. Textual documentation structure is usually defined in the quality manual and is essentially composed of chapters, sections, subsections, and paragraphs (we can further decompose paragraphs in sentences composed of words, made up of characters, but the benefit of such a fine
decomposition usually does not pay the overhead of such a complex structure), each of them providing a part of the whole document content. Even diagrams, usually included in such documents must comply with a well-specified meta-model (most of them have a graph-based structure, e.g., UML\textsuperscript{2} use case, class, statechart, activity, component and deployment diagrams). Source code is typically organised in packages, each of them contains classes composed of attributes and methods (composed of instructions).

All these kind of documents can be considered without taking into account this structure and versioning can be provided by keeping track of any change applied to the file they are contained in. However, a finer-grained document management, in which each single element of the document (e.g., a paragraph within a document or a model element in a diagram) is managed as a different entity and versioning is provided for both the atomic entity changed and the entities it is contained in, is more desirable, as it gives the possibility to define more detailed traceability links. In fact, the software engineer is enabled to specify traceability links between entities within the same document, as well as between parts of two different documents, resulting in a more precise impact analysis. Moreover, such a fine-grained artefact management reduces the likelihood of conflicts arising when different software engineers apply changes to the same document. In fact, in case the document is decomposed in sub-artefacts, a conflict occurs only if changes concern the same sub-artefact. It is worth noting that, even if such kind of conflict (i.e., changes occurring in different parts of a document) can be usually addressed by automatic

\footnote{In the field of software engineering, the Unified Modeling Language (UML) is a standardized specification language for object modelling. UML is a general-purpose modelling language that includes a graphical notation used to create an abstract model of a system, referred to as a UML model. UML is officially defined at the Object Management Group (OMG) by the UML metamodel, a Meta-Object Facility metamodel (MOF). Like other MOF-based specifications, the UML metamodel and UML models may be serialized in XMI. UML was designed to specify, visualize, construct, and document software-intensive systems. UML is not restricted to modelling software. UML is also used for business process modelling, systems engineering modelling, and representing organizational structures [80].}
merge algorithms for textual documents, the scenario is much more complicated when dealing with non-textual documents (e.g., UML diagrams), where an automatic conflict resolution is generally impractical even when changes do not affect the same model element. Moreover, since an artefact is decomposed in a hierarchy of simpler objects eventually annotated with text, the definition of more precise access policies to specific parts of the artefact is also enabled, thus allowing a finer task allocation and responsibility identification. This is particularly important in a distributed environment, where a work breakdown structure is usually defined to decompose the project components into manageable tasks assigned to distributed development teams.

In this thesis I present ADAMS (ADvanced Artefact Management System), an extensible system that integrates project management features such as resource allocation and process control and artefact management features, such as coordination of cooperative workers and artefact versioning, as well as context-awareness and traceability features. In particular, I will focus on the fine-grained management of software artefacts.

1.1 ADAMS Evaluation

To assess both the user satisfaction and the effectiveness of ADAMS, some experimental studies have been carried out. ADAMS has been used as process support and artefact management system in the Software Engineering courses of the Computer Science program at the University of Salerno (Italy). Two different experimentations were conducted: a first evaluation has been conducted in 2005 and involved about 140 students allocated on 17 projects. The results have been collected together with several suggestions and consideration to improve the system and, in 2006; a second evaluation has been conducted. The second evaluation involved 28 students allocated on 4 projects.

At the end of the projects, the students evaluated ADAMS through a questionnaire. Each question refers to an attribute of four quality characteristics of interest, namely functionality, usability, robustness, and performance. In this thesis
an analysis of the collected evaluation questionnaires is presented together with a
discussion on some of the results emerged during the evaluation of ADAMS and the
threats to validity that can affect our experience.

ADAMS is currently being experimented in the Software Engineering course of
the Computer Science program at the University of Salerno (Italy).

Moreover, a comparison between the functionalities provided by ADAMS and the
related work is also presented

1.2 Thesis organisation

The rest of the thesis is organised as follows. Chapter 2 presents related work
while Chapter 3 presents an overview of ADAMS. Chapter 4 presents the fine-
grained artefact management of ADAMS, while Chapters 5 illustrates the
cooperative development support of ADAMS. Chapter 6 presents an evaluation of
the system while Chapter 7 concludes and gives some indications for future work.
Chapter 2

2 Related Work

ADAMS is an extensible system for software project management aiming to provide a full feature environment supporting software engineers during software development and evolution. However, in this thesis we will mainly focus on two features of ADAMS, namely hierarchic versioning and traceability management. To this aim, in this Chapter I will present an overview of the tools that provide these functionalities. A comparison between the related work discussed and ADAMS is presented in Chapter 6.

2.1 Hierarchic Versioning Management Systems

Several commercial and research Configuration Management systems providing versioning have been proposed \cite{10}, \cite{11}, \cite{22}, \cite{27}, \cite{63}, \cite{73}. However, most of these tools do not provide an active support to hierarchic documentation versioning. Even if the user can organise composite artefacts using folders and subfolders, such an approach is limited to an enumeration of components, without providing any structural information relating components to the composite.

2.1.1 COOP/Orm and CoEd

Hierarchical versioning has been approached by Asklund et al. \cite{5} who propose a unified extensional versioning model. In this model, a document is a tree structure
consisting of local data, a composite node (which is a collection of nodes), or a link node (which is used to represent arbitrary relations between documents). The model has been implemented in COOP/Orm [51] and CoEd [9]. COOP/Orm [51] is a research tool developed at the University of Lund, in Sweden, supporting the natural hierarchic structure of textual documents, and providing version control and merge facilities within distributed groups. COOP/Orm supports hierarchically structured documents directly since they are very frequently occurring and simple to represent and handle. Such documents can be seen as a kind of internal configurations where each unit as well as the configuration as a whole is version controlled together.

The development history of the document is presented as a graph that can be used to browse the version history of the document, view particular versions of it and compare two versions, either sequential or further apart in time (Figure 1). In order to update a document, the software engineer selects an originating version and creates a new version, applies a sequence of changes to one or several information units, and finally freezes the new version. Following the “change propagation” scheme, all change to an information unit will go into the corresponding delta together with new versions of all composite units they are part of. Since a document has a tree structure, the change propagation ripples up to the top of the tree. As a result, selection of a document version precisely determines the version of all information units of the document. The changes to a document can include changes to information units as well as to the structure of the document (adding/deleting units).

Hierarchical documents are browsed and edited with a specialised editor which allows the user to directly see the differences between versions of a document, both in terms of changes to an information unit as well as to the configuration itself, such as adding or deleting of units in the configuration.

Creating a new version is seen as a comparison with its originating version and changes are highlighted as they are entered. An explicit version graph with a graphical user interface (see Figure 1) allows the user to view and browse the document in terms of its versions.

Users working on the same document are free to create new versions and variants
of the document. The editors offer strong support for merging of variants, suggesting default results and identifying conflicts for the user to solve [5]. During merge changes to the contents of information units as well as changes to the structure can be handled.

Coop/Orm also provides support to the context awareness. Indeed, the version graph is shared by all users editing or viewing the same document. The creation of a new version of the document is thus immediately visible to all the users that access the version window.

![Figure 1. Coop/Orm hierarchical document with version graph](image)

CoEd [9] is a visual environment for textual hierarchic document versioning, developed at the University of Aalborg, in Denmark. CoEd uses the same document management approach adopted by COOP/Orm, focusing on support to LaTeX documents.
2.1.2 CoVer

CoVer [44] (Contextual Version Server) is a hypermedia version server implemented on top of the HyperBase hypermedia engine [70]. Versioned objects are represented by multi-state objects, representing a composite of references to the states of versioned objects.

The hypermedia version server is implemented on top of the Cooperative Hypermedia Server (CHS). The applications in the publishing environment define their application-specific data types (i.e., subclasses of CHS nodes, links, and composites) in the application interface of CHS [70]. Since the hypermedia version server is an enhancement to CHS (see Figure 2), applications may implement different versioning policies tailored to the needs of the actual users via the application interface. CHS offers nodes, links, and composites that can be equipped with application-defined attributes. Objects can be accessed by their attribute values using the query language of the underlying database system of CHS. CHS maintains object histories. It stores the creation time and the author of each node, link, composite, and attribute and records each update to these objects with time and author information in an update history. However, CHS has no notion of versions and does not preserve previous states of updated objects. However, CoVer is intended for hypermedia documents and does not provide support for most of the artefacts produced during software development.

To make versioning not an all or nothing approach, in CoVer are defined two different types of objects, namely single-state objects (snobs), representing non-versioned objects, and multi-state objects (mobs), representing versioned objects. Each CHS object, i.e. node, link, or composite, is considered a snob and can be created by the usual CHS commands. It may be transformed into a mob by an explicit operation.

A mob represents a versioned object by gathering all states of the versioned object in its version set. A mob is implemented on top of CHS as a composite holding references to all states of the versioned object it represents. The states of a versioned object are called versions and are represented by individual nodes, links, or
composites. To preserve the states of versioned objects, versions of nodes, links, and composites can be frozen.

One of the key concepts in CoVer is the task tracking. Starting from a goal, users make coordinated changes to their hyper document. These coordinated changes guide version creation and serve version selection and identification. In addition, the inherent use of annotations in hypertext systems provides valuable context information for versioning. Moreover, maintaining the derivation history of hyper documents across document boundaries is another basis for version selection.

![Three layer architecture integrating CoVer](image)

**Figure 2. Three layer architecture integrating CoVer**

### 2.1.3 Stellation

Stellation [20] is another fine-grained CMS aiming to provide versioning
functionalities for source code. Instead of having the code organisation of the system dominated by its layout in source files for storage, programs and related development artefacts are stored as small, fine-grained units called fragments (see Figure 3).

```java
package stellation.util;
import java.util.Map;
import java.util.HashMap;

public class StringMap {
    public StringMap(Map m) {... }
    public StringMap() {... }
    public void put(String key, String value) {... }
    public String get(String key) {... }
    protected HashMap _map;
}
```

**Figure 3. Example of Stellation fragments for Java**

The basic feature of Stellation is what the author calls multidimensionality i.e., the ability to view and manipulate code using multiple overlapping viewpoints. Each viewpoint presents code in a different organisational structure, representing a different decomposition into concerns, or a different organisational or functional relationship between program artefacts. Multidimensionality is enabled by a combination of fine-grained storage, flexible metadata constructs for annotating program artefacts, mechanisms for creating versioned aggregate structures, and dynamic queries for creating and populating new viewpoints.

Instead of having the code organisation of the system dominated by its layout in source files for storage, programs and related development artefacts are stored as small, fine-grained units called fragments. The exact granularity of storage is
language specific: the smallest self-contained unit of code in the language. In Stellation, all storage is in terms of fragments.

The mechanism for combining groups of fragments into larger structures is the aggregation. Aggregation allows Stellation to represent dynamically generated collections of source code that is presented in a source-file like form. It also represents a way for representing relationships between software artefacts. For example, an aggregate can represent the relationship between a set of requirements, and the fragments of code that implement those requirements. Finally, aggregates form the basic data structure used by Stellation itself for providing a variety of facilities for supporting collaboration. For example, an aggregate can represent the set of artefacts placed under a lock for coordination between programmers.

### 2.1.4 SubCM

Volzer et al. [75] use composition to define hierarchical views of product structure, independently of the underlying artefact-repository structure. This approach has been adopted in SubCM, a generic lightweight tool to enable flexible configuration management of hierarchies of subsystems, designed to be used on top of more traditional file-based CM systems. The SubCM approach is lightweight because it requires no changes to be made to the system or repository structure, nor to existing CM practices.

SubCM automatically extracts configuration and change data from an underlying CM repository. To this aim, the user has to define a subsystem hierarchy by labelling a set of subsystems and their constituents. Subsystems can contain other subsystems and atomic configuration items. Moreover, the constituents can be shared among different subsystems. The hierarchy is represented by means of a directed acyclic graph, having atomic configuration items at the leaf nodes and references between subsystems as the edges. The user nominates a collection of system baselines, called the baselines reference basis (e.g., a branching structure with branches corresponding to the “is a modification of” relationship between different baselines of the system). The user decides which baselines to include in the
reference basis. Given a subsystem hierarchy and a baselines reference basis, the SubCM Tool inspects the CM repository and builds a subsystems version graph corresponding to each baseline in the reference basis.

Figure 4 shows an example of the subsystem version graphs generated and maintained by the SubCM Tool for a given baselines reference basis. The dashed arrows represent the relationship “is a modification of” between subsystems and the vertical arrows represent the relationships between the baselines and the corresponding subsystem versions.

Figure 4. Subsystem version graphs for a given baselines reference basis
2.2 Traceability Management Systems

Several research and commercial tools are available that support traceability between artefacts. DOORS [74] and Rational RequisitePro [66] are commercial tools that provide effective support for recording, displaying, and checking the completeness of traced artefacts using operations such as drag and drop [74], or by clicking on a cell of a traceability link matrix [66].

2.2.1 Molhado

Molhado [57] is an interesting SCM infrastructure that was specifically designed to support the development of object-oriented SCM systems. It provides both traceability management functionality and fine-grained version management supporting versioning of any fine-grained units at any structural levels in a software document, and providing version history and change information at the semantic level. Molhado was specifically designed to support the development of object-oriented SCM systems.

The key point of Molhado is its object-oriented approach to SCM in which all system objects are versioned and persistent.

Molhado structural SCM system has been built based on a primitive data model, called Fluid Internal Representation (IR) [13]. Main concepts of this representation model are summarised in the node-slot pattern (see Figure 5a). The basic unit is represented by the node that is used to represent any type of object. A slot is a location that can store a value in any data type. It can also be used to store a reference to a node. Slots can exist in isolation or attached to nodes, by means of an attribute. An attribute is a mapping between a node and a slot. It may have particular slots for some nodes and map all other nodes to a default slot. The data model can thus be viewed as an attribute table where rows correspond to nodes and columns correspond to attributes. The cells of this table represent the slots. Moreover, a third dimension for the table can be considered if we add the information on the version.
Unlike many SCM systems that focus on managing different versions of individual system objects, Molhado emphasises the evolution of the entire system. All system objects are versioned in a uniform, global version space. Similarly to ADAMS, a version in Molhado is global across the whole project and is a point in a tree-structured discrete time abstraction, rather than being a particular state of a system object. Differently from us, the state of the whole software system is captured at certain discrete time points and only these captured versions can be retrieved in later sessions, while we allow the retrieval of any past configuration of the system. The current version is the version designating the current state of the project. When the current version is set to a captured version, the state of the whole project is set back to that version. Each change applied to the current version of a project causes the creation of a temporary version, branching off the current project version. This temporary version is recorded only in case this is explicitly requested by the user. Note that, in order to record the history of an individual object, the whole project has to be captured.

Molhado distinguishes among three kinds of slots: constant (i.e., slots that can be
initialised and not modified), simple (i.e., modifiable), and versioned (i.e., slot that
can have different values in different versions). A sequence is a container with slots
of the same data type and a unique identifier. Sequences may be fixed or variable in
size and share common slots together. Figure 5b) shows a simple example of an
attribute table. The versioned slot associated with node “n1” via attribute “a1”
currently holds a reference to the node “n2”. Sequence “seq1” has two versioned
slots referring to “n1” and “n2” respectively. Slots in the table can also be simple or
constant slots.

Figure 6. Fine-grained version control in Molhado

Figure 6 and Figure 7 show an example of fine-grained version control
implemented in Molhado and the versioning of traceability links, respectively.
Similarly to ADAMS, Molhado provides versioning functionality for both a tree-
based and a graph-based fine-grained composition.
In the example illustrated in Figure 6, versions v2 and v3 branch off from the same version v1 (in v2 the node 4 was deleted and the content of node 5 was changed, while in v3 node 6 was inserted and node 3 was deleted). The values of versioned slots in the attribute table are changed to reflect modifications to the tree at these versions. Versioning for a directed graph is similar except that the attribute table does not have the “parent” attribute for nodes.

This fine-grained versioning management can be applied to XML-based documents (including specific sublanguages like SVG graphics and animations, and UML diagrams descriptions) and programs (represented by means of their abstract syntax tree). The complete project can be versioned by considering it as a named entity that represents the overall system structure of a software project. Since the
project is represented as a tree, it is versioned in accordance with the tree-based fine-grained versioning scheme above discussed.

Besides providing architectural configuration management functionalities, Molhado [57] also manages traceability links between architecture and code artefacts during software evolution. Molhado uses a single versioning mechanism for all software components and for the connections between them and can track changes at a very fine-grained level, allowing users to return to a consistent state of a single node or link. Traceability links can also be versioned considering their graph-based structure (see Figure 7).

### 2.2.2 RDD-100

RDD-100 [46] is a Requirements Driven Development (RDD) software suite that uses several mechanisms to aid the user in analysing and identifying requirements. These include a parser tool that can be defined and developed to help the user identify single or compound requirements.

RDD-100 provides the user the ability to link requirements with any element in the system’s hierarchy. These associations may be between requirements and other requirements, or between requirements and elements representing the physical, functional, or logical architecture, or between requirements and any other element needed to support the user’s business process rules. RDD-100 supports viewing requirements for all the views of requirements architectures (operational, functional, and physical) as well as the inter-relationships between these views. RDD-100 provides the user with the ability to see the links made between system design database data elements stored in RDD-100. The user can see the links graphically (through built-in or custom hierarchies) or outline/text (multi-element views) or in hardcopy (through Report Writer query templates). For large sets of requirements, RDD-100 can generate E-size plots of the hierarchies that facilitate requirements reviews. The information needed for traceability analysis includes not only requirements but also rationale, assignments, criticality, test/validation and many other issues to the requirement, allocation, and the system element to which
requirements are linked. RDD-100 categorises and links information through the ERA (Element-Relationship-Attribute) model. The user is allowed to categorise requirements in a specific manner, the rules for this categorisation can be captured and automated. RDD-100 provides the user with the ability to track changes to a specific requirement or other engineering element during the project’s life cycle, capturing baselines in ASCII files. The tool uses a master-subordinate structure: master copies of the System Design Database can be maintained and distributed under management control to the appropriate individuals. Each engineer can then act on the same published data, either on all the data, the data to which ownership has been assigned to the individual, or only the subset of data provided as a specific task. The tool also provides the ability to contrast and compare two different versions of any document.

RDD-100 also supports concurrent review, mark-up, and commenting through the ability to change the content of an element’s attributes, the ability to associate issues or comments with any specific element, and to re-integrate the changes and comments through the change integration process.

2.2.3 TOOR

TOOR [64] (Tracing Object Oriented Requirements) is a research tool in which traceability is not modelled in terms of simple links, but through user-definable relations that are meaningful for the kind of connection being made. This lets developers distinguish among different links between the same objects.

TOOR is based on Joseph Goguen’s broad view of requirements, which suggests that not only technical but also social factors play a significant role in software development [41].

TOOR needs the software engineer to create a project specification by declaring classes, subclasses, and attributes for each requirement and any other item he/she wants to trace. The tool automatically creates a template for each created class. Successively, objects are created by selecting a class from a graphical menu and then filling in the template. Relations are created similarly, as they are implemented
as objects. Objects and relations can be easily modified or deleted to reflect new situations, as well as the class declarations in the project specification.

Relations are the basis for traceability in TOOR. The tool provides three different trace modes: the Selective Tracing restricts focus to selected patterns of objects and relations, using a small language of regular expressions over an alphabet of object and relation identifiers; the Interactive Tracing allows interactive browsing through the objects related to a selected object, either backward and/or forward; and Non-guided Tracing allows going from one object to another, inspecting contents as desired. Moreover, TOOR can relate objects that are not directly linked using mathematical properties of relations such as transitivity.

2.2.4 Ophelia

The Ophelia project [72] is an EU funded initiative that aims to develop a platform to support software engineering in a distributed environment. The Ophelia project aims to provide support for integrating heterogeneous tools by defining a set of standard interfaces for accessing their generic functionality. These interfaces (called Module Interface Specifications) are described in terms of CORBA IDL definitions and are defined for such areas of project development as requirements, modelling, repository, bug tracking, project management, metrics and documentation management tools.

Ophelia also specifies several services that access various elements of the project, including knowledge management and traceability. The traceability module of the Ophelia project is founded on the fact artefacts are represented as CORBA objects. By considering all the artefacts of the project Ophelia is enabled to capture all the relationships present and not just those derived from requirements, for example relationships between defects and code, defects and models, documentation and code, code and unit tests. Moreover, the traceability layer can be seen as an essential part of project meta-data (see Figure 8). The traceability layer is organised in a graph structure that can be used as a starting point for browsing elements of the project.
Chapter 2. Related Work

Figure 8. Ophelia Traceability Layer

Unfortunately, the Ophelia Traceability Module is currently under beta development. Thus some of the functionalities are not available for use. As an example, the authors are planning to implement a simple mechanism of notifications, where objects signal a change in their status and this change is propagated along relationships present in the traceability layer to other elements of the project.

2.2.5 ArchEvol

An important issue of traceability management concerns the evolution of traceability links. Nistor et al. [58] developed ArchEvol an environment that
manages the evolution of architecture-to-implementation traceability links throughout the entire software life cycle. The proposed solution centres on promoting strong interconnections between the architecture, implementation, and the versioning sub-systems while still allowing their independent development. To this aim, ArchEvol maintains the mapping between architecture and code and ensures that the right versions of the architectural components map onto the right versions of the code (and vice versa), when changes are made either to the architecture or to the code. ArchEvol specifically builds upon three previously existing tools: ArchStudio [4], Eclipse [33], and Subversion [22] (see Figure 9).

Figure 9. ArchEvol Overview

ArchStudio provides the facilities for managing architectures, through an XML-based architecture description language, Eclipse provides the development environment supporting source code manipulation, and Subversion is used to store evolving versions of the architecture and implementation.

Since the architectural model evolves alongside its component implementation, the architecture description itself is versioned. To this aim, a specific version of the architecture consists of an architectural configuration made up of specific
component and connector versions. Besides recording the changes to the architectural model, versioning in such a manner allows the software engineer to determine for each version of the architecture the corresponding versions required for instantiation of all components and connectors.

2.2.6 OSCAR

OSCAR [12] is the artefact management subsystem of the GENESIS environment [6]. The GENESIS environment is intended to support distributed software engineering by providing a lightweight process-agnostic tool-set incorporating work-flow description and enactment, agent-based interaction between clients and the artefact management system. OSCAR has been designed to non-invasively interoperate with workflow management systems, development tools, and existing repository systems.

Each artefact in OSCAR possesses a collection of standard meta-data and is represented by an XML document containing both meta-data and artefact data that include linking relationships with other artefacts (see Figure 10).

OSCAR introduces the notion of “active” software artefacts that are aware of their own evolution. To support such an active behaviour, every operation on an artefact generates events that may be propagated by an event monitor to artefacts deemed to be interested in such events by their relationships with the artefact generating the event.

Figure 11 illustrates the data model used by OSCAR to manage software artefacts. At the highest level, a set of classes with various operations and attributes exist, mirroring the relationships between artefacts and their contents and providing methods to manipulate the properties of the artefact. The first class, Artefact, contains the basic artefact structure and its data, acting as a fly-weight for the other artefact-related classes. Secondly, the Property class represents a simple property of the artefact, in this case a string, though there are other property types available. The Version class contains the version information for an artefact. Finally, the Relationship class expresses a relationship.
Chapter 2. Related Work

Figure 10. OSCAR Architecture

Figure 11. OSCAR Artefacts Data Model
2.2.7 MILOS

MILOS [55] (Minimally Invasive Long-term Organizational Support) is an interesting Web-based process support system that improves the coordination and information exchange among virtual teams. MILOS allows the specification of hierarchical, object-oriented product models. Within process models, activities and their interrelationship are described. A process is defined by a description of the process goal, a set of conditions, process attributes, the products needed to plan and to execute the process, a set of alternative methods to reach the process goal, the products to be produced and resource allocations.

MILOS supports the execution of globally distributed software development projects in several ways. Project managers are able to define tasks and decompose them into smaller subtasks. They can schedule them by using the Web-based user interface of MILOS. In addition, the information flow between tasks can be specified: For each task, a user is able to define expected outputs. These outputs can then be used as inputs for other tasks. In contrast to standard workflow engines, MILOS allows on-the-fly plan changes and redefinitions of the information flow, notifying team members affected by those changes and updating the workflow engine accordingly.

MILOS-ASE (or M-ASE), is a branch of the MILOS project that focuses on agile methods support, web-based system development and knowledge management by building communities of practice [54]. M-ASE integrates structured and unstructured knowledge sharing tools. On the one hand, it allows several project teams to share a process centred experience base. Users are able to share knowledge by externalising it into Wiki³ pages. Both, the M-ASE Experience Base as well as

---
³ A Wiki is a website that allows visitors to add, remove, edit and change content, typically without the need for registration. It also allows for linking among any number of pages. This ease of interaction and operation makes a wiki an effective tool for mass collaborative authoring. The term
the M-ASE Project Support System use Wiki technology. Thus, knowledge sharing within a team as well as across team boundaries is supported. In addition, M-ASE indexes the Wiki pages as well as the database contents in the same text retrieval engine. Thus, a user is enabled to search for information in a single query that retrieves textual information and database contents. Finally, the built-in traceability component supports change notifications and helps the project participants to ensure that the project plan as well as the state of the enactment engine reflects the “real world” development process.

2.2.8 APER

Chen and Chou [19] have proposed a method for consistency management in the Aper process environment.

The method is based on maintaining different types of traceability relations between artefacts, including composition relations, and uses triggering mechanisms to identify artefacts affected by changes to a related artefact.

Aper is an Internet-based process environment. It is composed of the Aper language, the Aper compiler, the Aper server, the object management system (Aper OMS), and multiple Aper clients.

Aper clients communicate with the Aper server through Internet (see Figure 12). The Aper compiler compiles process programs to generate Java code which will be enacted (executed) by the Aper server. During process enactment, the Aper server interacts with Aper clients to assign work to developers, and stores and retrieves software artefacts.
Aper allows decomposing software artefact to allow establishing inter- and intra-product relationships. Trigger processes can be defined in relationship types to catch and handle events that are relevant to consistency management. Aper relationship unit is capable of defining all kinds of relationships and extending the built-in relationship types. Examples of built-in relationship types are Depend-On and Part-Of relationships. The Depend-On relationship defines dependency among artefacts or among sub-artefacts of different artefacts. Changing an artefact (or a sub-artefact) enacts a trigger process to change those dependent on it. The Part-of relationship defines decomposition relationships between an artefact and its sub-artefacts. If necessary, the relationships can be defined between a sub-artefact and its sub-artefacts. When an artefact is retrieved for manipulation (e.g., browsing, reusing, or updating), its sub-artefacts will also be retrieved.

2.2.9 Event Based Traceability

Cleland-Huang et al. [21] have developed EBT (Event Based Traceability), an approach based on a publish-subscribe mechanism between artefacts. EBT is composed of a requirements manager, an event server, and a subscriber manager connected using a standard communication mechanism (see Figure 13).

The requirements manager handles the requirements and is responsible for triggering change events as they occur. The subscriber manager places initial
subscriptions on behalf of the artefacts it manages, interacts with the process modelling domain to register for appropriate tasks, handles event notifications on behalf of the artefacts under its control, and supports the process of restoring an artefact and related traceability links to a current state.

The event server manages subscriptions, receives event messages, customises event notifications according to the process model and subscriptions, and forwards task directives in the form of event notification messages to the subscribers. The event server is primarily responsible for handling subscriptions, receiving change notifications, and forwarding customised event messages to the subscriber managers of dependent artefacts. Customised messages are needed to guide update activities during event resolution. The event server stores process knowledge in terms of appropriate actions to be taken by specific subscriber types in response to standard events.

When a subscription is placed, the centralised enactment component within the event server guides the subscriber in registering for specific task directives. When a change occurs on a given artefact having the publish role, notifications are sent to all the subscriber (dependent) artefacts.

Figure 13. System level model of EBT
2.2.10 PROSYT

Traceability has also been used in the PROSYT environment [24] to model software processes in terms of the artefacts to be produced and their interrelationships. Similarly to ADAMS, PROSYT is an artefact-based process support system. This artefact-based approach results in process models composed of simpler and more general operations than the operations identified using an activity based approach. PROSYT is capable of tolerating deviations from the process model during enactment and manage the inconsistencies between a process instance and the process model [25] by tracing the deviating actions and supporting the users in reconciling the enacted process and the process model, if necessary. It features effective mechanisms to help users in facing unexpected situations; it continues to handle process support and automation even during deviations, and supplies all the information needed to reconcile the process model and the process actually followed. Deviations can be tolerated as long as they do not affect the correctness of the system.

One of the key features of PROSYT is its ability to modify the level of enforcement adopted and the consistency handling policy at enactment-time, to adapt them to the different situations that may arise during process enactment. The process manager is also allowed to choose the level of enforcement adopted on a per-user basis; i.e., different users may be allowed to perform different kinds of deviations.

2.2.11 Sysiphus

Sysiphus [16] is an environment for distributed modelling and collaboration. Similarly to ADAMS and Molhado, Sysiphus manages system models (problem statements, requirements, architecture, detailed design, test cases), collaboration artefacts (informal comments about other models, issues, change requests, and action items), and organisational models (relationships among participants, and between participants and the system and collaboration models they create) in a
single and shared repository. This approach enables traceability and awareness services to be provided for any type of managed information.

Versioning functionality has been recently added to the Sysiphus environment by integrating the shared repository with Subversion [22].

Collaboration, and organisational models are represented as a single graph and functionalities for exploring, searching, filtering, and analysing the graph are available (see Figure 14). To reduce the amount of information displayed for complex traceability graphs, the element under consideration is displayed at the centre of a focused graph while its directly related elements are displayed around it. Moreover, search and filters functionality are also available. Similarly to ADAMS, artefacts exist independently from each other and can be created in any order, thus increasing flexibility with respect to the process and the development method used.

A notification service is also provided that sends emails to selected participants to raise their awareness of changes in the model. Notifications range from changes to documents, system model elements (e.g. use cases, components, classes), to issues and action items. To limit the proliferation of unnecessary notification, users can subscribe to specific type of changes they want to be notified about as well as the model element classes of interest. They can also subscribe to changes in documents or in document sections, resulting in notifications when any model element within the document hierarchy changes. Finally, changes in the collaboration model are

---

**Figure 14. Sysiphus traceability graph**

A notification service is also provided that sends emails to selected participants to raise their awareness of changes in the model. Notifications range from changes to documents, system model elements (e.g. use cases, components, classes), to issues and action items. To limit the proliferation of unnecessary notification, users can subscribe to specific type of changes they want to be notified about as well as the model element classes of interest. They can also subscribe to changes in documents or in document sections, resulting in notifications when any model element within the document hierarchy changes. Finally, changes in the collaboration model are
managed by sending notifications to the last modifier of a model element, when annotations are attached to it.

Sysiphus enables a template-based development. In fact, at the beginning of a project, an initial set of documents, consisting of the empty sections and their filters is presented to the software engineers [16].
Chapter 3

3 ADAMS Overview

ADAMS (ADvanced Artefact Management System) is an artefact-based Process Support System (PSS). It enables the definition of a process in terms of the artefacts to be produced and the relations among them.

ADAMS provides functionalities to manage resources, projects, and artefacts. Figure 15 presents the data model of ADAMS. In particular, the system enables the definition of roles within a project. Standard project level roles are the Project Manager, who manages the resources allocated on a project, defines the artefacts to be developed and the corresponding managers, allocate resources to artefacts, and define artefact dependencies, the Quality Manager, who manages standard artefact templates and review checklists, and the Traceability Manager, who is responsible for maintaining traceability links between software artefacts. Additional roles can be associated at artefact level, such as the Artefact Manager, who manages the evolution of an artefact and defines roles and permissions for software engineers working on it, and the Artefact Review Manager, who manages the review process the artefact undergoes in order to be accepted as a project baseline. Finally, a system level role is associated to the Administrator who manages the system itself and its resources, including user accounts, projects creation and allocation of human resources, and artefact types to be managed by the system.

ADAMS has a web-based architecture. The system is decomposed into several subsystems with a layered architecture (see Figure 16).
The user interface is implemented using JSP\textsuperscript{4} and HTML\textsuperscript{5} pages (see Figure 17),
however a graphical front-end has been also implemented by developing an Eclipse [33] plug-in\(^6\) that allows the software engineer to work on the artefacts directly from the IBM integrated development environment (see Figure 18).

---

\(^5\) HyperText Markup Language (HTML), is the predominant markup language for the creation of web pages. It provides a means to describe the structure of text-based information in a document and to supplement that text with interactive forms, embedded images, and other objects. HTML can also describe, to some degree, the appearance and semantics of a document, and can include embedded scripting language code which can affect the behaviour of web browsers and other HTML processors [80].

\(^6\) Eclipse is an open-source, platform-independent software framework for delivering what the project calls "rich-client applications", as opposed to "thin client" browser-based applications. So far this framework has typically been used to develop Integrated Development Environments (IDEs), such as the Java IDE called Java Development Toolkit (JDT) and compiler (ECJ) that comes as part of Eclipse (and which are also used to develop Eclipse itself). Eclipse employs plugins in order to provide all of its functionality on top of (and including) the rich client platform, in contrast to some other IDEs where functionality is typically hard coded. This plugin mechanism is a lightweight software component framework [80].
The current version of the Eclipse plug-in provides a subset of the cooperative functionalities. In particular, the versioning is enabled by using the HTTP client provided for the IDE. This allows the plug-in to connect to the ADAMS server and instantiate an HTTP session between the plug-in and the ADAMS web browser.

Figure 17. Web-based interface of ADAMS

Figure 18. Eclipse plug-in for ADAMS
3.1 Resource Management Subsystem

The Resource Management Subsystem provides administrative functionalities for user management, such as account management (see Figure 19), as well as functionalities concerning the human resources of an organisation, such as skill management.

Resources are organised in teams by the Administrator during the project definition. When the user is allocated to a project he/she becomes member of the project team and a role is assigned to him/her (see Figure 20). A user can be associated to many different projects and can play a different role for each of the assigned project.

![Figure 19. User account management in ADAMS](image)

3.2 Project Management Subsystem

The Project Management Subsystem provides project management functionalities, such as definition of the schedule, allocation of human resources to the artefacts.
The definition of the schedule is performed during the definition of the artefacts to be produced (see Figure 21) by specifying the starting and due date for the artefact.

Besides being available in the artefact card, this information is used to determine the schedule status of each artefact. Indeed, when the software engineers access their to-do list containing the set of artefact to produce, the system highlights late and overdue artefacts (see Figure 22).

<table>
<thead>
<tr>
<th>ID</th>
<th>First Name</th>
<th>Last Name</th>
<th>Login</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fausto</td>
<td>Pavan</td>
<td>fafas</td>
<td>Project Manager</td>
</tr>
<tr>
<td>2</td>
<td>Andrea</td>
<td>de Lucia</td>
<td>delucia</td>
<td>Project Manager</td>
</tr>
<tr>
<td>3</td>
<td>Rocco</td>
<td>Oliveto</td>
<td>oliveto</td>
<td>Project Manager</td>
</tr>
<tr>
<td>4</td>
<td>Gaetano</td>
<td>Morrone</td>
<td>loginAdams</td>
<td>Team Member</td>
</tr>
<tr>
<td>5</td>
<td>Domenico</td>
<td>Laurino</td>
<td>dom84en</td>
<td>Team Member</td>
</tr>
<tr>
<td>6</td>
<td>Francesco</td>
<td>Gnarr</td>
<td>fragna</td>
<td>Team Member</td>
</tr>
<tr>
<td>7</td>
<td>Luca</td>
<td>Castaldo</td>
<td>bombo</td>
<td>Team Member</td>
</tr>
<tr>
<td>8</td>
<td>Giuseppe</td>
<td>Carpanieri</td>
<td>Poppino1</td>
<td>Team Member</td>
</tr>
<tr>
<td>9</td>
<td>Matia</td>
<td>De Rosa</td>
<td>mattio</td>
<td>Team Member</td>
</tr>
<tr>
<td>10</td>
<td>Calabria</td>
<td>Giuseppina</td>
<td>pina</td>
<td>Team Member</td>
</tr>
</tbody>
</table>

**Figure 20. Project Team management in ADAMS**

Resources are assigned to the artefact to produce by the Project Manager who can allocate subsets of the project team to each artefact thus defining artefact teams (see Figure 23). It is worth noting that each resource can be assigned a different set of permissions according to the role they have in the development of the specific artefact.
Chapter 3. ADAMS Overview

Figure 21. Definition of schedule for an artefact

Figure 22. Visualisation of the to-do list in ADAMS
Chapter 3. ADAMS Overview

3.3 Quality Management Subsystem

The Quality Management Subsystem provides functionalities to manage quality at process level within an organisation. This entails the definition of the artefact types managed by the system (see Figure 24) and the standard templates and review checklists associated to them.

![Artefact Types management in ADAMS](image)

Each artefact type can be assigned multiple templates (see Figure 25). When a new artefact of that specific type is created, the quality manager can decide to use...
one of the template available or instantiate a customised template for the specific artefact type instance (i.e., the artefact).

![Figure 25. Template definition for artefact types](image)

The Quality Management Subsystem also support to the inspection process. Software inspection is a software engineering practice aiming at identifying defects, reducing rework, and producing high quality software systems [37][38]. The inspection tool is described in Section 5.4.

### 3.4 Artefact Management Subsystem

The Artefact Management Subsystem provides all the functionalities to manage the artefacts of a project and concurrent development of software engineers. It uses the revision control subsystem to create, modify, and delete artefacts, as well as to manage the artefact state and versions. An access control subsystem (not shown in Figure 16) is used by each subsystem anytime it needs to verify whether a human resource has the permission to perform an operation as well as by the user interface to enable or disable functionalities according to the user roles.

ADAMS enables fine grained management of software artefacts that can be hierarchically organised, thus providing a product-oriented work breakdown structure to the project manager. Traceability links between related artefacts can also be inserted and visualised through the traceability management subsystem and used
for impact analysis during software evolution. The event and notification management subsystem is used to enhance context-awareness within the software project. Through this subsystem software engineers are notified when changes are made to artefacts they are allocated to by concurrent developers. Software engineers can also subscribe events occurring to artefacts they would like to be notified about. Finally, the traceability layer can be used to propagate events occurring to an artefact to the artefacts depending on it. The functionalities provided by the artefact management subsystem and the subsystems it uses are the core of this thesis and will be discussed in detail in the following chapters.

3.5 Cooperative development subsystem

The Cooperative Development Subsystem includes tools used to enhance synchronous and asynchronous collaboration within ADAMS. Examples of synchronous collaboration tools are the internal, the UML collaborative editor (see Figure 26) presented in Section 5.2 and the distributed inspection meeting tool (presented in Section 5.2).

Figure 26. ADAMS collaborative editor for UML diagrams
Besides the asynchronous facilities provided by the Quality Management Subsystem, in fact, ADAMS supports the synchronous distributed inspection meeting, where all the inspectors can remotely meet, see the checklists of each other, and discuss about the artefact to produce the finally agreed checklist (see Chapter 5).

Examples of asynchronous collaborative tools are the internal e-mail tool and the rationale management tool (presented in Section 5.5).
Chapter 4

4 Fine-grained management of software artefacts

Software Configuration Management is a well established and common practice in the late phases of software development, notably during programming and integration. However, it is less commonly practiced in the early phases, i.e. analysis and design. One of the reasons for this behaviour is that there are usually not many versions (and hardly any configurations) of analysis and design documents. Thus, often making backup copies is considered sufficient. However, object-oriented development methods (e.g. the Unified Process) lead to a significant increase in the complexity and the number of versions of the documents in early phases [60]. Moreover, most of the tools providing versioning functionalities (e.g., CVS [10] or Subversion [20]), are specifically designed to manage files containing lines of text in a pretty-printed format, without considering the logical structure of the document contained in a file. Conversely, high level software documents are filled with elements that cause the automatic merge functionality fail (e.g., images, diagrams, tables). Moreover, in object-oriented development methods, analysis, design and implementation are considered as parallel activities. As a result, even simple modifications can affect several files, or parts of files, belonging to different development phases.

Fortunately, most of the documents produced during software development have a well-defined hierarchic structure. This structure is usually defined in the quality manual of an organisation and/or in the quality plan of a project and is essentially composed of chapters, sections, subsections, and paragraphs (we can further decompose paragraphs in sentences composed of words, made up of characters, but
the benefit of such a fine decomposition usually does not pay the overhead of such a complex structure), each of them providing a part of the whole document content. Even diagrams, usually included in such documents must comply with a well-specified meta-model (most of them have a graph-based structure, e.g., UML use case, class, statechart, activity, component and deployment diagrams). Source code is typically organised in packages, each of them containing classes composed of attributes and methods (composed of instructions).

All these kind of artefacts can be considered as an atomic entity. However, a finer-grained management, in which each single element of the document is managed as a different entity and the tool functionalities address both the atomic entity and the entities it is contained in, is more desirable, as it gives the possibility to define more detailed traceability links. In fact, considering each fine-grained element as an artefact, the software engineer is enabled to specify traceability links between entities within the same document, as well as between parts of two different documents, resulting in a more precise impact analysis. Moreover, such a fine-grained artefact management reduces the likelihood of conflicts arising when different software engineers apply changes to the same document. In fact, in case the document is decomposed in sub-artefacts, a conflict occurs only if changes concern the same sub-artefact. Finally, reuse of also components is enables, as the software engineer can include the same low level artefact in several composite artefacts, thus improving the consistency within the software project.

ADAMS adopts a fine-grained management of software artefacts. In particular, artefacts in ADAMS can either be an atomic entity (associated to one or more files) or they can be composed of an arbitrary number of atomic and further composed artefacts. The fine-grained management of artefacts (i.e., the intermediate work products) proposed in ADAMS, allows the software engineer to choose the granularity level of this decomposition according to the artefact type, the concurrency level needed for the human resources allocated on them, the number of software engineers that are likely accessing them at the same time, how meaningful is considering each piece as a single unit (e.g., to allow traceability link definition or responsibility individuation).
4.1 Artefact composition

In ADAMS, the composition of artefacts is realised by defining composition links between the composite artefact and each of the contained artefacts. In particular, the composition can be realised by selected the type of artefact for the parent and the children artefacts (see Figure 27). In order to reduce the number of artefacts retrieved by the query, a filter can also be specified during this phase. The artefacts matching the selected criteria are then proposed as parent and children of new compositional link. In case the number of artefacts in the filter is excessive, the software engineer can also select a subset of them to avoid the system proposing too many links (see Figure 28).

![Image](source.png)

Figure 27. Selection of parent and children artefact types and filtering

The resulting artefacts are then used to propose new compositional links that the software engineer can insert within the system (see Figure 29). It is important to note that the system automatically excludes from the candidate compositional links the artefacts currently locked. In fact, locked artefacts can cause problems when being involved in a new composition as their content is under modification. It is possible that the new document structure influences the content of each component. For this reason, the system avoids locked artefact to be involved in a change of their structure. Similarly, the system also prevents cycles in the document structures avoiding an artefact to be used in a compositional link as child of one of its
descendants. This is important to avoid the rebuilding process of the artefacts to be involved in an infinite loop.

Figure 28. Selection of a subset of recovered artefacts

Artefact hierarchies in ADAMS can be defined bottom-up (putting simple artefacts together to form a composite artefacts), top-down (starting from high level documents and incrementally adding sub-artefacts or decomposing parts of the composite artefact), or in any combination of the previous approaches, linking
together already defined trees. Such a composition process is dynamic, as the artefact manager can decide to change the artefact structure even when the development is already started. This gives the flexibility necessary in a highly changeable and evolving environment, like a software development project, and allows changing the grain of the decomposition as soon as the artefact structure becomes inadequate for the selected decomposition.

Composition of artefacts can also be performed through the graphical user interface of ADAMS. In fact, the hierarchical visualisation of artefacts supports the drag & drop of nodes and automatically creates the appropriate compositional links. The behaviour of the drag & drop operation can be specified by right-clicking and accessing the contextual menu “Drag Option” (see Figure 30).

![Figure 30. Drag & Drop options of ADAMS](image-url)

The available options are:

- Drag Before
- Drag Into
- Drag After

The “Drag Before” and “Drag After” options do not cause the creation of a compositional link, but they are simply used to establish an ordering between
children of the same parent (brothers). This ordering is important to define the structure of the final document as the rebuilding procedure take into consideration the ordering of the nodes as they appear in the hierarchical visualisation.

The “Drag Into” option, indeed, creates a new compositional link between the source artefact (the artefact selected before clicking the mouse left button) and the target artefact (the artefact selected before releasing the mouse button). It is important to note that the same checks performed by the system during the manual link creation (e.g., cycle prevention and locking verification) are performed during the creation of composition with the drag & drop functionality of the user interface. In case the link can not be created, the system notifies the software engineer with the appropriate message (see Figure 31).

**Artefacts for Project 35**

![Image of artefacts for Project 35](image)

**Figure 31. Cycle prevention check during the composition of an artefact**

Note that drag & drop is intended to move sub-artefacts between documents or sections. This means that, in case the source artefact was already part of an existing hierarchy (e.g., it was contained in another section of the same document or in a section of a different document or as a part of another artefact), the old compositional link will be deleted before the creation of the new link. In case the software engineer wants to preserve the structure of the source document, he/she can select the optional checkbox labelled “Copy”. With this option selected, the system does not delete any link the source artefact was involved in before the drag & drop operation. It is important to note, however, that the copy operation does not create a new artefact, but simply a new compositional link. This has the effect of creating a
shared artefact between the parent of the source artefact and the target artefact. Note that this is one of the benefits of the fine-grained management of software artefact. In fact, the shared artefact (e.g., an actor description as well as a glossary term) can be modified in one of the documents it is contained in and the effects of the modification are reflected to all the other containing documents.

The event notification mechanism of ADAMS takes into consideration the particularity of compositional links distinguishing the notification rules between compositions and other types of traceability links. In particular, the propagation of events across a hierarchical composition follows a bottom-up direction. This means that events are propagated only from children to parents. Even the graphical visualisation tool highlights the difference by means of a special symbol for compositional links.

4.1.1 Rebuild of a composed artefact

The composition of software artefacts in ADAMS is intended to define a hierarchical structure for a complex document and thus taking advantage on the benefits of a fine-grained management approach (definition of finer work products, more precise traceability management, fine-grain locking of the piece of artefact to access). However, the distribution of the content over an arbitrary deep and wide hierarchy of smaller sub-artefacts can cause the software engineers to spend a considerable quantity of time simply to put the pieces together before the delivery of the documents as well as anytime a complete version of the composite artefact is required. This time wasting activity can be avoided in case the structure of the document and the format of sub-artefacts can be automatically managed by the system.

The current version of ADAMS provide a specific support for two of the most common type of artefact used during a software development process, namely textual documentation (e.g., project documentation such as project plans, requirements specifications, design document, test documents, user manuals) and system models (in the current version of the system the automatic rebuild of UML
Use Case diagrams is provided).

Currently ADAMS supports the main formats adopted during the editing of this type of documentation:

- Microsoft Word (doc)
- Open Document Text (odt)
- Rich Text Format (rtf)
- Portable Document Format (pdf)
- Text Only (txt)
- HTML Documents

This means that the user can use one of these formats to generate the leaves of the composed artefact. The same documents are also available as exporting formats during the rebuild phase (see Figure 32).

![Figure 32. Rebuild of a composed artefact](image)

The rebuild functionality provided by ADAMS reconstructs the document starting from its leaf nodes and preserving the structure of the document decomposition as well as the ordering between the nodes. ADAMS uses the Open Document format to normalise the content of the document components, then merges all the pieces in a single open document and finally converts (if necessary) the merged document in the output format as required by the user.
The rebuild procedure can also use a template file to produce the cover of the document using dynamic information, including the project name, the artefact name, the document responsible, and the team allocated to the document. The revision history containing the list of all the versions of the reconstructed documents together with the comments added to each version is also reported into the document. Finally, a summary of the document structure reporting the versions used for each of the leaf node used in the reconstruction is attached to the document.

The index of the document is automatically generated during the rebuild procedure using the name of the artefacts for sections and subsections. The quality manager can also define the appearance for the auto-generated parts of the document by uploading a special template file. Templates are a mechanism used by ADAMS to upload a preliminary version of the artefact (see Figure 42). Similarly, automatic rebuild procedure uses a special kind of template to format the reconstructed document. This allows the quality manager to define different templates for different artefact types as well as different formatting rules for each different template.

During the rebuild procedure, the software engineer is asked to choose how the leaf nodes have to be managed. By default, the system creates a new section for each artefact (including leaves of the hierarchy). However, in some cases, it can be useful to have pieces of the document not included in a specific subsection. An example is represented by a section decomposed in subsections. Each subsection has a file associated with the content of the subsection. In case the document manager wants to add an header to the section (between the name of the section and the first subsection) the only possibility would have been to upload the content of the section header in the artefact referred to the section, and let the rebuild procedure to use both internal and leaf nodes. However, such a possibility is in contrast with the definition of composite artefact (i.e., the internal node content should reflect the actual content of its sub-tree and is not intended to integrate this with additional text). The solution adopted in ADAMS is to allow the software engineer to specify that leaf node content must not be considered as a subsection, but as a simple piece of the artefact. Figure 33 shows the structure of a Requirements Analysis Document structured using leaf nodes for lowest level section (a) and using leaf nodes simply
to store the content of the section they are contained in. Note that the former approach has the advantage of keeping the artefact structure simpler. In the other hand, the latter approach allows the software engineer to specify a header (represented by the artefact named “Intro”) to the section “Proposed System”. This double possibility leaves the software engineers the flexibility to adopt the approach that best fit their need, while allowing the composition of documents of arbitrary complex structure.

![Diagram of document organisation using (a) and skipping (b) sections for leaf nodes](image)

**Figure 33. Document organisation using (a) and skipping (b) sections for leaf nodes**

It is worth noting that the rebuild procedure can be executed starting from any composed artefact within the project. This allows the software engineer to reconstruct an entire document, as well as a single chapter, section or subsection of the document.

### 4.1.2 Visualisation of composed artefacts

Composite artefacts in ADAMS can be visualised using the hierarchical view or the graph-based view.

The hierarchical view (see Figure 34) allows the software engineer to browse the
artefact structure similarly to what append when browsing a file system directory. In fact, when the software engineer clicks on a composed artefact, the system explodes the next level of the hierarchy visualising the children of the selected artefact. It is worth noting that in ADAMS an artefact can have several parent nodes (this is the essence of the reuse). In this case, the same artefact can appear several times in a hierarchical view, as child of different composed artefacts (e.g., in Figure 34 the artefact Glossary is shared between the RAD document and the SDD document).

Figure 34. Hierarchical view of composed artefacts
Figure 35. Graph-based view of composed artefacts

In the graph-based view, the structure of composed artefact is represented by means of a direct acyclic graph (see Figure 35).

The graph-based view has been realised by integrating in ADAMS Grappa, an open source graph drawing package developed by AT&T Research Lab as a port of a subset of GraphViz to Java [43]. Grappa has been also customised with artefact type icons associated with the different types of artefacts managed in ADAMS, in order to provide enhanced visual information about the artefacts. Tool tips have been used to show immediate artefact information as soon as the user positions the mouse over an icon. The artefact used as query has been shown in red to point out its position within the traceability graph. Moreover, each different link type has been associated with a specific notation. Finally, the contextual menu shown on a right click of the mouse over an artefact of the graph has been customised. In particular, we added three new menu items to open the artefact card, to immediately send a feedback to the developers allocated to the artefact, and to subscribe events concerning the specific artefact, respectively (see Figure 35).

To show the graph, the software engineer selects a source artefact from the artefact list and defines the dependence types he/she would like to visualise (see Figure 36). In particular, he/she can decide whether to include or exclude specific types of link (e.g., compositions or traceability links) and nodes (e.g., standard
artefacts, rationale elements).

In the graph-based view the sharing of artefacts is clearer than in the hierarchical view. On the other hand, the ordering of the components can not be represented in a graph, while it is provided by the hierarchical view. To this aim, ADAMS provides a double view for composite artefacts, allowing the software engineer to choose the view that best reflects the needed information.

![Select the version for artefact ADAMS(5788)](image)

4.2 Traceability management

Software artefact traceability is the ability to describe and follow the life of an artefact (requirements, code, tests, models, reports, plans, etc.) developed during the software lifecycle in both forward and backward directions (e.g., from requirements to the modules of the software architecture and the code components implementing them and vice-versa) [42].

The usefulness of defining and maintaining traceability between software artefacts includes program comprehension, impact analysis, and reuse of existing software. In fact understanding the relationships existing within and across software
requirements, design, and implementation is a key factor for successful software projects.

Besides the possibility to define traceability links between any artefacts managed by the system, the fine-grained management of software artefacts enables the traceability manager to define dependences between components of the same artefact as well as between components of different artefacts.

**Figure 37. Traceability link definition form**

A traceability link represents a relationship between an independent artefact (or source artefact) and a dependent artefact (or target artefact). ADAMS allows the software engineer to specify traceability links between pairs of artefacts (see Figure 37). Traceability links can be organised in a traceability graph where the artefacts are identified by nodes and the traceability links are represented with edges of the graph. Within a traceability graph we identify traceability paths, i.e. sets of artefacts connected by traceability links. Obviously, an artefact along a traceability path could be impacted by each artefact appearing in the part of the path preceding it as well as it could impact on each artefact appearing in the part of the path following it.

Filters on the artefact type and/or on the name of the artefact can be specified to
guide the traceability manager during the identification of both the source and the target for the traceability link (see Figure 37). A stereotype can be used to identify the nature of the link.

ADAMS enables software engineers to manually define and manage traceability links between artefacts (see Figure 37). In order to use the manual traceability links management functionality, the software engineer responsible for maintaining the consistency of the traceability layer (the traceability manager), selects the source and the target artefacts among the already defined project artefacts.

The traceability layer can be organised in a graph having the nodes represented by artefacts and the edge represented by traceability links. In ADAMS, this graph can be visualised by using the graph-based view used to show the composition of artefacts (see Figure 35 and Figure 38.).

Figure 38. Traceability graph visualisation in ADAMS
Chapter 4. Fine-grained management of software artefacts

The graph-based view can be browsed to look at the state of previously developed artefacts, to download the file associated to the artefact, or to subscribe events on specific artefacts in order to receive notifications concerning their development.

Note that compositions and traceability links are visualised differently in Figure 38. As discussed before, in case the software engineer wants to focus only on the traceability links, he/she can decide to hide the compositions by unselecting the checkbox Artefact Composition before visualising the graph (see Figure 36).

4.2.1 Traceability links recovery

Traceability links evolve together with the evolution of the system, i.e. they can be added or removed, when new artefacts are added to the project, existing artefacts are removed or newer versions are checked-in.

![Recovering Functional Requirement onto Use case traceability links]

<table>
<thead>
<tr>
<th>Suggested Links</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Source Artefact</td>
</tr>
<tr>
<td>169</td>
<td>Doctor management (Requirement)</td>
</tr>
<tr>
<td>158</td>
<td>Medicine management (Requirement)</td>
</tr>
<tr>
<td>157</td>
<td>Patient management (Requirement)</td>
</tr>
<tr>
<td>158</td>
<td>Medicine management (Requirement)</td>
</tr>
<tr>
<td>165</td>
<td>Doctor management (Requirement)</td>
</tr>
<tr>
<td>108</td>
<td>Doctor management (Requirement)</td>
</tr>
<tr>
<td>ID</td>
<td>Source Artefact</td>
</tr>
<tr>
<td>167</td>
<td>Patient management (Requirement)</td>
</tr>
</tbody>
</table>

Figure 39. Suggested Links and False Positives
Since the number of managed artefact within a system that adopts a fine-grained approach tends to be very high, this task tends to be hard to manage. For this reason, ADAMS integrates a traceability link recovery tool (see Figure 39), called ADAMS Re-Trace [27] [28], that supports the software engineer during the traceability link definition and the identification of warning links, i.e., the links traced by the software engineer and missed by the tool (see Figure 40).

ADAMS Re-Trace uses Latent Semantic Indexing (LSI) [31] as an information retrieval method to calculate the similarity measure among the artefacts within a project. Given a source artefact $a_i$, the tool compares it against the other artefacts in the artefact space and ranks these artefacts according to their similarity with $a_i$. Moreover, the tool uses a similarity threshold to cut the ranked list and presents the software engineer only the subset of top artefacts that are deemed similar to $a_i$. It is worth noting that artefacts having a high similarity probably contain similar concepts, so they are likely good candidates to be traced on each other. In particular, ADAMS Re-Trace compares the links identified by the traceability recovery tool

![Figure 40. Warning Links](image-url)
with the links manually traced by the software engineer and highlights the disagreements between the tool results and the software engineer choices. To this aim, when the software engineer performs the traceability recovery function, the tool visualises the suggested links, i.e., the links not traced yet by the software engineer and retrieved by the tool and the false positives, i.e., links classified as false positive by the software engineer but retrieved by the tool (see Figure 39). The tool also visualise warning links, i.e., the links traced by the software engineer and missed by the tool (see Figure 40).

4.3 Versioning management in ADAMS

Versioning of composed artefacts in ADAMS is achieved by allowing the software engineer to decompose it into an order set finer-grained sub-artefacts. This decomposition can be applied to the components of the composite artefact, thus creating an arbitrary complex hierarchy of artefact. For each of the element of the hierarchy versioning is provided. ADAMS versioning management is based on the check-out/check-in paradigm adopted in most of the current versioning and configuration management tools. The software engineer repeatedly edits a given revision (usually working on a local copy of the file), until freezing it. Once a revision is frozen, it is stored permanently and no change can be applied to it. Usually, such a revision freeze is performed with a check-in or commit command. Different versioning management systems adopt different strategies to store the version content and implement concurrency policies. Each of them has pros and cons in terms of performance, disk space, reliability, or concurrency criteria.

ADAMS relies on a relational database to store the metadata related to the artefacts and their versions. The data content for each artefact is instead stored relying on the file system. The interface towards the file system is implemented using a subsystem of the Revision Control Subsystem that actually does nothing more than reading and writing the files of the artefact. This choice has been adopted to improve the performances and the simplicity of storing and retrieving functionalities. However, it is obviously possible to interface such a subsystem with
a wrapper for other versioning systems (e.g., CVS [10] or Subversion [22]) as well as to implement additional features such as caching and delta versioning.

Figure 15 illustrates the data model of ADAMS. Among the metadata stored with each artefact version, ADAMS records the timestamp of each check-in operation. Such a version timestamp is used to reconstruct the state of the project in a precise instant of the system evolution. The current state of the project is simply recovered by accessing to the state of the project at the request time (obviously the system distinguishes between the main development trunk and its branches, depending on the user request). However, it is possible to browse the artefact repository setting up a past timestamp and having the system to show the snapshot of the system in that precise instant of the development process. This can be useful for configuration management, but also provides support whenever it is necessary to recover any other intermediate (non baseline) version of the system.

Besides time stamping each artefact version, the system also records such information for traceability links, so that also this information is coherent with the past view of the project. Traceability links are also versioned such that it is possible to modify the nature of a link without loosing its history. Moreover, traceability links do not merely store the timestamp of their creation, but similarly to the artefacts they are provided a temporal interval starting with their creation and ending with their deletion. This allows to precisely determining the content of the project for a stated past timestamp without considering not yet created entities and excluding (logically) deleted entities.

Versioning allows the history management of the file associated to an artefact providing a mechanism to revert to previous versions of an artefact and manage concurrent accesses to the same artefact by different software engineers. However, versioning is only a part of the artefact lifecycle. ADAMS poses a great emphasis to the artefact lifecycle allowing the software engineer to flexibly manage the evolution of each artefact. Figure 41 illustrates the artefact life cycle in ADAMS. It is worth noting that, despite being influenced by each other, a composite artefact and the sub-artefacts composing it follow their own lifecycles.
When a new artefact is conceived, the software engineer responsible for its creation and management (the artefact manager) defines the schedule for the artefact and decides if branches are allowed during its production. In this phase, template and review checklists are usually associated by the quality manager according to the artefact type to be developed. One of these templates is selected and eventually customised for the specific instance of the artefact type (see Figure 42).

Once activated, thus producing its first version, the artefact is developed similarly to a standard versioning management system. Several draft versions of an artefact can be created and maintained by ADAMS.
Each time a new revision is produced, the software engineer, besides specifying CVS-like comments, can tag the artefact as completed, to produce a new baseline, or draft, in case he/she still needs to work on the artefact (see Figure 43). If branches are not allowed, each resource can lock the artefact (check-out) and work on it until a new version is uploaded (check-in). Otherwise, different branches can be produced and worked independently by each resource, which can also produce different versions of each branch. When all branches are closed, they can be merged in a new version of the artefact. Completed non-branch versions of an artefact undergo the revision process and are either approved and closed (baseline) or sent back to the draft state. Baselines can be reworked only after the artefact manager explicitly reopens the development phase causing an artefact transition to the draft state. When the project is closed all the artefacts are frozen and no more versions can be created for them until the project is eventually reopened.

According to the choices of the quality manager, the baseline production can require an informal or a formal review phase before being baselined. In this case, during the check-in operation the software engineer can decide to put the artefact in revision. An informal review of the artefact is performed by filling in a review
checklist that is attached to the artefact. In case the artefact satisfies the quality standards specified in the review checklist, a new baseline is created and the artefact is closed, so that no further changes can be applied to it unless the artefact manager decides to restart the working phase.

![Check-in form](image)

**Figure 43. Check-in form**

### 4.3.1 Versioning of composed artefacts

Composed artefacts and fine-grained artefacts follow the same lifecycle illustrated in Figure 41. A composite artefact may have files associated with it and versioning is provided for them. The usefulness of having a file associated with a composite artefact (whose content is actually provided by accessing its children content) is twofold. In case the artefact structure can be specified by means of a description file (e.g., a UML diagram can be represented by means of an XML-based document describing the diagram components together with information concerning colours, positions, and dimension of each element), the file associated with the composite
artefact is used by the system to automatically reconstruct the artefact starting from its children node content. It is worth noting that, being such a file versioned by the system, it is possible to modify the internal structure of the composite artefact (i.e., how to put its components together) whenever needed, by merely updating the file associated to the composite artefact. This also guarantees that the internal organisation of the previous version of the artefact is preserved into the previous versions of its description file.

However, for some type of composed artefacts (e.g., the high level software documents), the artefact content (e.g., the paragraph of the textual document) is maintained in the leaf nodes of the hierarchy. In this case, the file associated to the composite artefact can be used to store specific artefact version (e.g., the specific version built and delivered to the client) or a specific intermediate version manually recomposed by the software engineer (e.g., to solve some formatting problems due to unsupported document types).

As a consequence, the content of the composed artefact can be out of sync with respect to the content of the artefacts it is composed of. In fact, when a software engineer modifies a leaf node of the hierarchy producing an updated version of the associated file, the last rebuild of all the artefacts containing it still contain the information that the modified artefact had at the rebuild time. Even if the automatic document reconstruction or a manual merge can be performed to build the artefact again using the newest version of each leaf artefact, the content of the composed artefact is not synchronised until the software engineers freezes the new file in a new version for the composed artefact.

In order to inform the software engineer that a composed artefact version is not synchronised with the versions of the artefacts it is composed of, the hierarchical and the graph-based views of ADAMS show these artefacts with an out-of-synch decorator. In particular, an exclamation mark over the artefact icon is used in the graph-based view (see Figure 35 and Figure 38) and an exclamation mark icon is used in the hierarchical view instead of the usual icon (see Figure 34). Even the project to-do list provides this information (see Figure 17 and Figure 44).
Figure 44. Out-of-sync visualisation in the to-do list
Chapter 5

5 Collaborative development support

The support for collaborative development in ADAMS is provided through typical configuration management features. In fact, ADAMS enables groups of people to work on the same artefact, depending on the required roles. Different software engineers can access the same artefact according to a lock-based policy or concurrently, if branch versions of the artefact are allowed.

5.1 Concurrent development management

ADAMS adopts a pessimistic approach to manage concurrency. In fact, to edit a frozen revision the software engineer has to declare his/her intentions by performing a check-out operation, thus locking the artefact. Once all the changes are applied, the file is checked-in again, causing the production of a new revision. This approach prevents conflicts due to modifications applied by different software engineers to the same artefact. The drawback of such an approach is a serialisation of the changes to the artefact that results in a poor support to cooperative development.

In order to allow the concurrent development within a cooperative development environment, ADAMS supports the branching and merging of artefacts. Whenever an artefact has to be modified concurrently by different software engineers, each software engineer creates a new branch and starts working on it. When they finish working on the artefact the changes can be merged in a new version.
The concurrent development functionality of ADAMS can be applied to any type of artefact managed by the system. As an example, let us consider the Software requirements specification\(^7\), a high level project document. According to the quality manual defined for the organisation, the software engineers involved in the requirements elicitation and analysis phases of the software development lifecycle, start working concurrently on the document, each of them accessing the sections related to the functionalities they have been assigned on (see Figure 45).

Managing the requirements analysis document as a composition of finer-grained sub-artefacts allows the project manager to precisely assign the responsibilities for each document section and the software engineers to focus on the work product they have been assigned. This means that two different software engineers allocated on different functional requirements are allowed to work independently from each other without the necessity to take care of locking and unlocking mechanism as well as conflict resolutions issues. This is particularly important in a distributed software development environment, where the work breakdown structure is defined, tasks are assigned to the software engineers, and the intermediate work products (e.g., requirements and actors description, scenarios, use cases, sequence diagrams, test cases) are successively merged in a deliverable document.

\(^7\) A **Software Requirements Specification** (SRS) is a complete description of the behaviour of the system to be developed. It includes a set of use cases that describe all of the interactions that the users will have with the software. In addition to use cases, the SRS also contains nonfunctional (supplementary or quality) requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints). Recommended approaches for the specification of software requirements are described by IEEE 830-1998. This standard describes possible structures, desirable contents, and qualities of a software requirements specification [80].
Figure 45. Allocation of users to document sections

5.2 Synchronous collaborative UML diagrams editing

Besides complex documentation artefacts, another type of artefact that benefit from the hierarchic versioning management provided by the revision control subsystem is represented by diagram.

Fine-grained management of diagrams relies on the ADAMS versioning management for history management and concurrent editing of both the diagram and each of the included components. Composite artefacts and components are managed as different configuration items.
Figure 46. Entity objects of the use case meta-model in ADAMS

Figure 46 shows the meta-model supporting the fine-grained versioning of a use case diagram. It is important to note that each element of a diagram can be related to a description (e.g., the description of a use case can contain its pre and post conditions, the flow of events, and exceptional conditions). As a consequence, the diagram and each of its elements can have files associated, whose versioning is provided by the Revision Control Subsystem of ADAMS. The file of the composite artefact is used to store the corresponding diagram representation, while the descriptions of each diagram element are stored in different files. Such a separation is also valuable for the reuse of components across different diagrams.

It is worth noting that this approach also enables the software engineer to specify traceability links between entities within the same document, as well as between parts of two different documents or diagrams, resulting in a more precise impact analysis.

ADAMS prevents conflicts due to modifications applied by different software
engineers to the same diagram. To ensure a pessimistic access control to the diagram as well as each of the contained elements, locks are recursively propagated across the hierarchy. When a UML diagram is checked-out, both the diagram and the contained elements are locked. This means that in case UML diagrams sharing elements are checked-out at the same time, the shared elements can only be edited in the first checked-out diagram, while they are locked in the other diagrams. As a consequence, developers working on other diagrams can modify all diagrams elements except the shared elements previously locked in other diagrams.

Figure 26 shows STEVE [30], the collaborative editor for UML diagrams, a tool integrated in ADAMS to provide a visual UML editor. Relying on the fine-grained artefact management of ADAMS, the tool manages each model element as a specific artefact and uses traceability links to maintain relationships among model elements. Moreover elements can be shared across several diagrams as the tool guarantees the concurrency when diagrams containing the same elements are accessed concurrently.

The UML editor allows the developers to access and modify the same diagram concurrently, thus allowing a distributed team to discuss and model the system directly from within ADAMS, maintaining the versioning for each model element as well as for the information shared during the meeting.

Figure 47 shows an example of list of UML diagrams that a particular software engineer has the permission to access within a project for the development of an e-shop. New diagrams can also be created, depending on the permissions of the software engineer. In particular, the diagrams listed in Figure 47 are two use case diagrams, namely UCD Product Purchase and UCD Stock Reorder. Figure 48 shows an applet where the use case diagram UCD Product Purchase has been selected and a new use case is being inserted in the diagram. Existing use cases and actors present in other diagrams and/or documents can also be reused and inserted in the diagram, thanks to the fine-grained artefact management provided by ADAMS.

A Coordination Server is responsible for keeping all the views of the diagram consistent and managing the coordination among the software engineers working on the same diagram at the same time. This coordination is achieved by using a token:
each time a software engineer has to apply a change to the diagram, he/she takes the token thus preventing other developers to make changes to the diagram. The communication among the software engineers is also guaranteed by the possibility to exchange textual messages within a collaborative session.

When the user holding the token commits the changes to the diagram (by performing a check-in operation), the diagram description is sent back to ADAMS to create a new version of the diagram and of the modified elements. New and deleted elements are also managed accordingly. In particular, in case a new element is created and inserted in the diagram, a new artefact is created in ADAMS and linked to the composite artefact (i.e., the diagram). In case the element inserted in the diagram is reused from the ADAMS repository, the artefact is simply linked to the diagram by adding a new compositional link and a new version is created in case it has also been modified. Elements removed from the diagram are not removed from the repository, as they can still be used in other diagrams.
For example, suppose that a software engineer, namely Bob, checks-out the use case diagram *UCD Stock Reorder* in Figure 47 to add a new use case, namely *Verify Order Status* (see Figure 49). Suppose that another software engineer, namely Alice, would insert the same use case in the use case diagram *UCD Product Purchase*, while Bob is still working on the use case diagram *UCD Stock Reorder*. As shown in Figure 50, the *Verify Order Status* use case is locked in the use case diagram *UCD Stock Reorder* (its colour is grey) and Alice can only include it without making any change.

It is worth noting that diagram elements can also be shared with analysis and design documents. The lock applied to the UML diagram ensures that shared elements cannot be concurrently modified within the documents (and vice-versa). For example, a different perspective of the fine-grained artefact hierarchy of a UML diagram is shown in Figure 51a.
Chapter 5. Collaborative development support

Figure 49. UCD Stock Reorder use case diagram

Figure 50. Visualisation of a locked use case in the diagram
Figure 51 shows how software artefacts composing the use case diagram *UCD Product Purchase* have been locked when Alice checked-out the diagram. Consequently, the elements are also locked in each diagram including them (see *Verify Order Status* within *UCD Stock Reorder* in Figure 51a). Figure 51b shows the structure of the requirements analysis document after the check-out of the use case *Product Purchase*. Also the sections containing the descriptions of the locked use cases are marked with the lock symbol. This prevents from concurrently modifying the description of the use cases from different perspectives.

It is worth noting that the reuse can produce unexpected side effects. In fact, in case a diagram element is shared across different diagrams, changes (e.g., renaming or changing the description) of a shared element affect all the diagrams containing it. The software engineers working on affected diagrams could be interested in any applied change. As an example, in case Alice modifies the use case *Verify Order Status* in the *UCD Product Purchase*, the *UCD Stock Reorder* will be marked as out-of-sync, as it contains the changes element. The system visualises this situation by means of an exclamation mark close to the artefact name in the hierarchical view.
(see Figure 51b). Notifications of such events are also sent via e-mail by using the event and notification functionality of ADAMS.

5.3 Event notification management

The support for cooperation of ADAMS also includes functionalities to improve the context awareness among the members of the team. In fact, most of the problems experienced by people working concurrently on the same artefacts, especially in distributed settings, are related to limited communication and coordination that leads to context awareness problems. In order to face this problems up, ADAMS provides an event-based notification mechanism that allows the software engineers working on an artefact to be notified when something relevant (e.g., another branch is created by another worker) occurs on the artefact (see Figure 52). This provides a solution to the isolation problem for resources working on the same artefact in different workspaces: in fact it is possible to identify potential conflicts before they occur, since interested resources are notified as soon as an artefact is checked-out and potentially before substantial modifications are applied to it. Notification messages provide references to the artefact concerning the notification, provided as hyper textual links to the appropriate entity (see Figure 52).
Unfortunately, event based notification has revealed some scalability problems, as the number of messages generated can slow down the system and cause the developer to ignore notifications because they require a lot of time to be managed. Moreover, not all the users need to be notified for specific events and not all the events are perceived as equally important by different users.

For these reasons, ADAMS enables software engineers to subscribe events they would like to be notified about, and specify the artefacts they are interested in, allowing each user to define the best compromise between an adequate context-awareness level and an endurable quantity of messages received.

Events mainly concern the operations performed on artefacts and projects. A number of events are automatically notified without any need for subscription. Examples include notifying a software engineer he/she has been allocated to a project or an artefact. By default, all the artefacts allocated to a developer are intended as fully subscribed (all events concerning the artefact are notified), whereas all the other artefacts are intended as unsubscribed (see Figure 53). However each developer can override such behaviour specifying for each artefact the most appropriate notification level.

<table>
<thead>
<tr>
<th>Artefact Name: INTRODUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Level</td>
</tr>
<tr>
<td>No Event</td>
</tr>
<tr>
<td>Artifact Card modified, Artifact deleted</td>
</tr>
<tr>
<td>New Feedback</td>
</tr>
<tr>
<td>Baseline accepted, Baseline rejected</td>
</tr>
<tr>
<td>New Version submitted for Revision</td>
</tr>
<tr>
<td>New Draft Version created (Meaningful Changes)</td>
</tr>
<tr>
<td>New Draft Version created (Text Corrections)</td>
</tr>
<tr>
<td>New Complete Version created for a Branch</td>
</tr>
<tr>
<td>New Draft Version created for a Branch</td>
</tr>
<tr>
<td>Artifact unlocked, Artifact activated</td>
</tr>
<tr>
<td>Artifact Checked-Out, New Branch created</td>
</tr>
<tr>
<td>New Comment added/modified, Comment removed</td>
</tr>
<tr>
<td>New Resource allocated, Resource allocation removed</td>
</tr>
<tr>
<td>All Events</td>
</tr>
</tbody>
</table>

**Note:** A level event subscription implies the subscription of all the levels above it!

**Figure 53. Event subscription in ADAMS**
To support the developers during the customisation of the types of event they would like to be notified about, the ADAMS Event Management Subsystem subdivides events in a layered classification in order to cluster events sharing the same level of relevance into the same class. This approach simplifies the subscription phase because developers are only requested to choose the relevance class for which they need to be notified (see Figure 53).

The layered structure of the event classification ensures them that they will be notified as soon as an event belonging to the class subscribed as well as an event belonging to one of the upper level classes is generated. As an example, developers could choose to be notified as soon as an artefact they are allocated on is checked-out, in order to prevent the isolation problem, but they could only be interested on meaningful changes to that artefact. Such a distinction is possible due to the improvement of the granularity of change management introduced in ADAMS. Events triggered on an artefact are automatically propagated through the traceability layer.

This allows the software engineer to be notified in case of changes applied to artefacts their work depends on (e.g., a system architect responsible for the system decomposition can be notified in case a non-functional requirement driving an architectural design choice is changed). It is important to note that event propagation can significantly increment the number of generated notifications. To this aim, event propagation has been limited to meaningful changes only i.e., changes whose impact is considered relevant by the software engineer, as specified during the check-in operation by the selection of the specific impact level (see Figure 43). Moreover, each software engineer has the possibility to set the maximum number of traceability link a notification can cross to reach the subscribed artefact, thus limiting the number of notification received (see Figure 53).

It is worth noting that the event notification level is automatically tuned by the system that monitors the behaviour of each single software engineer whenever he/she receives a new notification. In particular, a software engineer that repetitively ignored notifications concerning an artefact, will realistically ignore later similar notifications, for this reason the subscription level is decreased accordingly.
Obviously the level can be re-established by the software engineer.

In order to reduce the number of unimportant changes notifications, each draft version of an artefact checked-in into ADAMS is tagged with an impact value determined comparing the new version with the previously stored version. For this reason developers are required to specify the impact of the changes implemented as soon as they check-in a new draft version of an artefact (see Figure 43), so that it is possible to distinguish between unimportant changes (e.g. text corrections, code comments, language improvements) and meaningful changes (e.g. requisite modifications, new functionalities, error corrections). This information, together with subscriptions, is used by the Event Management Subsystem to decide whether to send a notification for the newer version or not.

Besides such kind of passive user notification, ADAMS provides direct communication mechanisms between software engineers, such as e-mails. Moreover, cooperation between software engineers during iterative software processes is supported through the possibility of sending feedbacks concerning software artefacts (see Figure 54). Software engineers that make use of previously developed artefacts to produce new artefacts might send feedbacks to the input artefacts in case they discover problems within such artefacts.

![Figure 54. Feedbacks visualisation and creation](image-url)
Besides the possibility to access the feedbacks for a specific artefact (Figure 55), according to the subscription level, feedback can also be notified by the Event & Notification Management Subsystem (Figure 52). Similar to the feedbacks, the artefact comments are another communication channel available to the software engineer to share information concerning a specific software artefact. Differently from feedbacks, that are intended to highlight potential problems in the artefact, comments are mainly used to better understand the artefact they refer to. This is the reason why the notification level for these two types of events is completely different.

Figure 55. List of the artefact feedbacks

5.3.1 Event notification propagation

The traceability graph is also used to propagate event notifications, so that the developers can be aware of changes applied to artefacts their work depends on. This can contribute to anticipate potential changes thus reducing expensive later modifications. Event propagation also reduces the complexity of subscribing several events for notification, avoids worthless redundancies, and prevents from forgetting indirect but essential subscriptions. In fact, software engineers are not required to subscribe the same events for artefacts connected by traceability links, but it is sufficient to subscribe only the events occurring on the dependent artefacts as the system is able to notify them as soon as those events occur on the artefacts directly.
or indirectly linked.

Moreover, the links defined during the composition/decomposition of artefacts into hierarchies of fine-grained artefacts, are used to propagate events.

Obviously, the Event and Notification Management of ADAMS takes into considerations the differences between a composition relation between a composite and its component and a traceability link defined between two dependent artefacts.

It is worth noting that the propagation across the traceability graph and the composition hierarchy can cause the proliferation of the number of generated messages.

To avoid this proliferation, developers are allowed to specify whether they want to be notified about events concerning artefacts indirectly impacting their work, i.e., artefact not subscribed being source of an indirect traceability link with respect to artefact they subscribed. Moreover, with respect to the layered structure of the event classification, only events involving meaningful changes (or higher) are propagated across the traceability links. The software engineer can also specify the maximum number of links an indirect notification can cross (see Figure 53). Concerning the composition hierarchies, the system propagates events only in a bottom-up direction.

To this aim, the Event & Notification Management Subsystem uses the traceability graph to compute the set of dependant artefact directly or indirectly impacted by the event. Moreover, the ancestors of the impacted artefacts are also computed.

For each impacted artefact the set of subscribers (for a suitable event level) are notified by the Event Management Subsystem. To select the set of the artefacts involved because linked by a traceability link, the Event Management Subsystem computes a reachability matrix. Such a matrix contains n rows and n columns (one for each artefact). The cell $m_{ij}$ contains an integer value representing the minimum number of traceability links to cross in order to reach the artefact $a_j$ starting from the artefact $a_i$ and moving across the traceability graph. Note that the reachability matrix should be recomputed any time a new traceability link is inserted or existing links are modified or removed. However, the system record the status of the matrix by means of a Boolean value (similarly to the sticky bit) that states whether the
reachability matrix is up to date (i.e., no change in the traceability matrix has occurred since the last rebuild of the reachability matrix).

5.4 Review management

ADAMS provides functionalities to support the project team during the review of the software artefact produced. Depending on the type artefact to be revised, an informal review of the artefact or a formal inspection process can be required to create a new baseline.

One of the advantage of a fine-grained artefact management with respect to the review management is that the selection of the formality level required can be defined according to the type of artefact (leaf, internal node, root element). As an example, formal review are usually required for the composite artefact, while informal review can be sufficient for the leaves of the hierarchy. Once defined the review policy (e.g., in the quality plan of the organisation), the choice between the review process can be automated.

The inspection process implemented in ADAMS is illustrated in Figure 56. The process is composed of seven phases, namely Planning, Overview, Discovery, Refinement, Inspection Meeting, Rework, and Follow Up.

During the Planning phase the scheduling of the quality manager specifies which artefact version must undergo a formal review process, defines or selects an existing checklist, and compose the inspection team. All these functionalities are accomplished by using the process support, human resources, and quality management functionalities of ADAMS. There is no need to specify the artefact author as this information is available from the underlying configuration management subsystem. After this phase all the inspection participants receive a notification containing the details of the inspection and a new task appears in their to-do-list.

In the Overview phase, the artefact author explains the design and the logic of the software to the inspectors. To this aim, he/she produces a document that briefly describes the purpose and the scope of the artefact to be inspected and then deploys
it in ADAMS. The inspection mailing list managed by ADAMS is used to notify all the inspection participants.

**Figure 56. ADAMS Inspection Process**

During the Discovery phase, the inspectors analyse the artefact and take note of the candidate defects by highlighting all the cases where the artefact does not comply with the control checklist. The tool supports the inspector during this phase by recording the identified defect, its severity, its location within the software artefact in terms of page and line numbers (see Figure 57), or picture/table sequential number. The discovery phase can be performed in multiple sessions. The tool highlights the check item not yet compiled. Anytime in this phase, the moderator can visualise the inspector’s defect logs, the check items compiled or not compiled as well as a preview of the merged defect logs containing the inspection output produced by the inspection team (see Figure 58). This information can be used to decide whether to stop the detection phase and start the next phase. However, in case the moderator does not access this information, ADAMS notifies
him/her by sending an event as soon as all the inspectors have completed the discovery phase.

![Figure 57. Defect identification](image)

![Figure 58. Merged defect list with agreement information](image)

When the refinement phase starts, the tool sends an email containing the conflict list to the inspection team. This mail also aims at notifying the team members that the conflicts can be analysed in order to get an agreement. The main goal of this phase is to remove false defects and to build the consensus on the true defects. As
suggested by the empirical study presented in [48] the system considers a defect as a true defect when at least two reviewers recognise it. In this case the defect is not discussed. However, the minimum number of reviews required to automatically get an agreement can be differently chosen according to the inspection process constraints. It is important to note that this phase is not mandatory, as the moderator can decide to skip it in case he/she decides to resolve the conflicts.

![ADAMS checklist-based inspection collaborative tool](image)

**Figure 59. ADAMS checklist-based inspection collaborative tool**

A synchronous inspection meeting can be performed to discuss about unresolved conflicts (see Figure 59). The inspection team can visualise all the other member’s answers and comments and discuss altogether. According to the meeting discussion, the moderator can decide whether the artefact needs to be reworked by the developers (rejecting the revision) or a new baseline can be created for the artefact (accepting the revision). Even this phase can be skipped in case the number of
conflicts is manageable by the moderator or the time distance does not permit a synchronous discussion. Note that the tool does not require that all the participants join the inspection meeting. This enables the enactment of sub-teams meeting to resolve conflicts not involving other inspectors.

The defect log is then used during the Rework and Follow Up phases. In fact, the author uses it to fix the artefact defects and the moderator (or the verifier) checks the changes applied by the author to address the identified defects, thus deciding whether a new inspection process has to be enacted or the reworked artefact can be baselined. It worth noting that the system maintains the defect logs for each of the artefact version. Thus, in case the artefact undergoes several revisions, it is possible to access the defect logs for each version. This is used by the moderator (or the verifier) to control that the author addressed all the highlighted defects.

5.5 Rationale management

The rationale management tool enables the software engineers to address open issues that need to be investigated by the team (see Figure 60).

![Figure 60. Issue manager of the ADAMS Rationale Management Tool](image)
Options can be proposed to address issues and arguments can be linked to the options. Once a decision is reached the issue is closed and can be accessed in the close issue view (see Figure 61). During the issue resolution process, project criteria already defined can be linked to the rationale elements and new criteria can be defined.

Rationale elements are treated as artefact and relations among them as traceability links. Special stereotypes are used to manage specific relations between the rationale elements (e.g., meets / fails between options and criteria or addresses / resolves between options and issues). Rationale elements can also be related to standard artefacts developed within a project (e.g., to the functional requirements raising the issue or the design decision description). To this aim the software engineer can select the type of artefact and use a filter in order to reduce the number of links proposed by the system (see Figure 62).

The elements (standard artefacts and/or rationale elements) recovered according to the specified parameters are used to propose new links. The software engineer can uses stereotypes to specify the relation type (see Figure 63). Rationale elements can be also visualised in the graph-based view of ADAMS, by selecting the checkbox Rationale Items (see Figure 36). In Figure 64 rationale elements are visualised in the graph-based view of ADAMS.
Figure 62. Selection of the elements to link with rationale

Figure 63. Creation of links between rationale elements
Figure 64. Rationale Items in the graph-based view
Chapter 6

6 Evaluation and Experience

In this section an evaluation of ADAMS is presented. In particular, a comparison between the functionalities provided by ADAMS and the related work discussed in Chapter 2 is presented. Moreover, the experience of the tool as process support system and document management system within a university environment is reported and the results of a survey questionnaire assessing the functionalities, usability, reliability, and performances of the system are reported. Finally a discussion of the evaluation is presented.

6.1 Comparison with other tools

The comparison of ADAMS with other tools will focus only on a subset of the functionalities provided by ADAMS. Section 6.1.1 presents the characteristics selected for the comparison. Moreover, some of the approaches presented in Chapter 2 are not implemented in any available tool. The selection of tools addressed in this comparison is presented in Section 6.1.2.

6.1.1 Evaluated characteristics and tools

The characteristics used in this comparison are the following:
• **PM** (Project Management): The tool provides an adequate support during the software development process (e.g., project work breakdown structure, resource management, and task and activity management).

• **CM** (Configuration Management): The tool provides support for cooperative work by allowing the management of software configuration and/or versioning management of software artefacts.

• **HV** (Hierarchical Versioning): Versioning management functionalities allow decomposition of artefacts in hierarchies of fine-grained artefact.

• **AR** (Automatic Rebuild): The tool provides functionality to support the document editor during the process of rebuilding the composite artefact previously decomposed into a hierarchy of fine-grained artefacts.

• **TM** (Traceability Management): The tool allows the quality manager to define and manage traceability links between software artefacts.

• **TV** (Traceability Versioning): Traceability links are versioned.

• **TR** (Traceability Recovery): The tool support the software engineer during the traceability recovery tools (automatic or semi-automatic traceability link identification).

• **EN** (Event Notification): Events are triggered to notify software engineer when something relevant happens on software artefacts their work depends on.

• **EP** (Event Propagation): Events are propagated through the traceability graph.

• **NO** (Notifications Optimisation): The tool provides mechanisms to limit the proliferation of undesired notification.

• **AB** (Artefact Browsing): The tool allows the software engineer to browse the artefact repository.

• **AL** (Artefact Lifecycle): The tool allows the software engineer to follow evolution of the artefact through the entire life cycle.

• **QA** (Quality Assurance): The tool provides quality assurance functionalities.
The tools used in the comparison are:

- APER
- ArchEvol
- COOP/Orm / CoEdr
- CoVer
- EBT
- GENESIS / OSCAR
- MILOS
- Molhado
- Ophelia
- PROSYT
- RDD-100
- SubCM
- Stellation
- Sysiphus
- TOOR

### 6.1.2 Tool comparison

Table I shows a summary of the comparison of ADAMS with other tools for traceability management and fine-grained management of software artefact.

Versioning of hierarchical documents is provided by [20] [44] [51] [57] [75]. Both COOP/Orm and CoEd are limited to textual documentation. CoVer focus on the versioning of hypertext documents, while Stellation is limited to the fine-grained versioning of source code. Hierarchical versioning management in SubCM is used to provide configuration management of hierarchies of subsystems. Besides the tools here presented, other approaches aim at modelling versioned documents as abstract syntax graphs [79], or associate a specific version tree to every object [77]. The only tool providing fine-grained versioning functionalities for different types of artefacts, covering most of the artefacts produced during the software development process is
Molhado [57]. Besides provides a great flexibility to the software engineer that is allowed to tailor the granularity level of the artefact decomposition according to his/her needs, ADAMS also supports specific artefact types providing functionalities to automatically rebuild complex artefacts when needed.

<table>
<thead>
<tr>
<th></th>
<th>PM</th>
<th>CM</th>
<th>HV</th>
<th>TM</th>
<th>TV</th>
<th>TR</th>
<th>EN</th>
<th>EP</th>
<th>NO</th>
<th>AB</th>
<th>AL</th>
<th>QA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>APER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ArchEvol</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COOP/Orm / CoEdr</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoVer</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBT</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GENESIS / OSCAR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MILOS</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Molhado</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ophelia</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROSYT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDD-100</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SubCM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stellation</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sysiphus</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOOR</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Versioning of diagrams is provided by Molhado through the versioning of the XML representation of the diagram. However, this approach limits the advantages of the fine-grained approach adopted by the tool, as the sharing of diagram components among different diagrams is not enabled. Odyssey-VCS [61] is an interesting version control system providing the fine-grained versioning of UML diagrams. However, even if it implements an algorithm to merge and detect conflicts between two different versions of the same UML diagram element, synchronous
modelling functionality is not provided as well as support for different types of artefacts.

Among the tools providing traceability management functionality, some approaches also enable context awareness by managing event notification and using the traceability layer to propagate events [6] [19] [21] [55]. However, unlike ADAMS, these tools do not offer facilities to limit the proliferation of notifications. ADAMS gives developers the possibility of customising the set of events they would like to be notified about, thus avoiding undesired notifications on one hand and receiving notifications that are not planned by the traceability layer on the other hand. Moreover, it implements an automating tuning of the subscription level taking into consideration the software engineer behaviour to optimise the event notification mechanism.

Traceability link visualisation is mainly realised by means of a graphical interface showing the traceability graph. Among the tool managing traceability Cover, TOOR, Ophelia, and Sysiphus allows the software engineer to browse the traceability graph. Cover allows the software engineers to browse the artefact space, however the traceability layer is limited to the links between hypertext document. Thus the software engineer cannot create links unless an anchor exists between the documents, resulting in a rigid support for general type of artefacts. Ophelia provides interesting browsing functionalities, but it limits to provide a traceability layer to an existing configuration management tool. Moreover, the browsing of compositions is not provided. Even Sysiphus provides an interesting functionality to browse the repository focusing only on a specific portion of the document space.

None of the tool presented provide functionality to support the software engineer during the traceability recovery tools. Traceability recovery methods have been proposed in the literature [3] [35] [47] [49] [53] [56] [71] [78] [76] [82] [83], [84] but they have not been implemented in such tools.

Context awareness support in ADAMS is also provided through the possibility to visualise the resource involve in the development of an artefact, detailed information about the locks such as which artefacts are locked (even indirectly) who has checked-out the artefact and when he/she performed the operation as well as to
follow the evolution of each artefact during the software lifecycle. MILOS provides awareness on what is going on in the project, while OSCAR supports context awareness through the notion of “active” software artefacts that are aware of their own evolution. Coop/Orm support to the context awareness is limited to the possibility to share the version graph among the users editing or viewing the same document.

An artefact lifecycle management is provided by OSCAR, by means of the “active” artefacts. However, OSCAR does not support fine-grained management of software artefacts, thus the management of the lifecycle of complex artefacts is not precise as in ADAMS.

Sysiphus like ADAMS provides support for the quality management, by allowing a template-based development, however, none of the tool described includes a functionality to assess the quality of the artefact produced by providing a software inspection process.

Another distinctive feature of ADAMS with respect to these approaches is the support for the management of the entire life cycle of artefacts, including quality assurance features such as checklist-based inspection and review phase and project template management.

6.2 ADAMS Experience

From April 15th 2005 to July 30th 2006 ADAMS has been adopted as process support and artefact management system within the projects conducted by students of the Software Engineering courses of the Computer Science program at the University of Salerno (Italy). This experience involved about 170 students allocated in 21 software projects aiming at developing software systems with a distributed architecture (typically three tier) using mainly Java and web technologies and a relational DBMS (typically MySQL). In particular, two different experimentations were conducted: a first evaluation has been conducted in 2005 and involved about 140 students allocated on 17 projects. The results have been collected together with several suggestions and consideration to improve the system and, in 2006, a second
evaluation has been conducted. The second evaluation involved 28 students allocated on 4 projects.

Each project team included between six and eight undergraduate students with development roles and two master students with roles of project and quality management, respectively. The process model adopted for software development was incremental: the students performed a complete requirements analysis and high level design of the software system to be developed and then proceeded with an incremental development of the subsystems. The goal was to release at least one increment by the deadline for closing the projects. The project manager was responsible for coordinating the project, defining the project schedule, organising project meetings, collecting process metrics, and allocating human resources to tasks. The quality manager was responsible for defining process and product standards of the project, collecting product metrics, and organising checklist-based artefact reviews for quality control. The project and quality managers were also in charge of building the traceability matrix for the software artefacts developed in their project and to this aim they were also trained on the use of the traceability recovery tool of ADAMS.

The choice of the granularity level for each of the artefact to be produced was left to the team (i.e., to the project and quality managers). The average number of artefacts produces within each project was 192 (140 artefacts were produced for the smallest project and 325 artefacts for the largest project); on the average, for each artefact about 4 different versions were produced (in some cases, up to 22 different versions of the same artefact were produced). The main software documents produced were the Requirements Analysis Document (RAD), the System Design Document (SDD), the Object Design Document (ODD), the Test Plan, and the test execution documents, besides project and quality management documents [15]. Using the composition links, such documents were decomposed into a large number of sub-artefacts. In particular, for each project the average number of composition links traced by the software engineers was about 100. It is worth noting that the Requirements Analysis Document was the artefact with the highest number of sub-artefacts. This result suggests that a fine-grained artefact decomposition has been
preferred to a coarse-grained decomposition for those artefacts involving a high number of resources concurrently, such as the RAD. This choice aims at reducing the number of concurrent branches allowing the development of different sections of the same document by different team members.

Another important data concerns the number of dependences and undirected links traced by the software engineers (the average number of these links for each project is about 250). Analysing such data (the average number of traced links and artefacts within each project) we can estimate the average dimension of a traceability graph. In particular, on the average, each project has a traceability graph composed of about 200 nodes (artefacts) and 350 edges (traceability links).

### 6.3 Survey Questionnaire

At the end of the projects, the students evaluated ADAMS through a questionnaire. Each question refers to an attribute of four quality characteristics of interest, namely functionality, usability, reliability, and performance. Besides the characteristics here discussed, some quantitative evaluation questions have been provided to the subjects. The results of this last group of questions have been omitted in this Chapter, and can be found in Appendix A. In Appendix A we report the complete questionnaire with the results of each of the evaluation sessions. Here we present the results from both the 2005 and 2006 evaluation questionnaire.

The evaluation of functionality is focused on the attributes of usefulness (see Table II) and adequacy (see Table III) of the functionalities provided by ADAMS. Besides these two attributes, another group of questions investigates user considerations on specific functionalities (see Table IV).

The data here reported include the distribution of answer for each question and for each year of experimentation. This is useful to highlight the improvements achieved by implementing some of the suggestions collected at the end of the first evaluation period. Aggregate data are not reported since some of the functionalities provided by the system have been improved and new functionalities have been introduced based on suggestions and consideration of the users.
In Table II we notice that user’s perception of the tool usefulness has globally incremented during the second experimentation. In particular, the overall perceived usefulness of ADAMS (Q1.1.1) rose from 74.36% to 87.5%.

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2005</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Q1.1.1. Was ADAMS useful during the software development process?</td>
<td>75.00</td>
</tr>
<tr>
<td>Q1.1.2. Was the possibility to add comments to artefacts useful?</td>
<td>42.14</td>
</tr>
<tr>
<td>Q1.1.3. Was the possibility to send a feedback to the artefact manager useful?</td>
<td>42.45</td>
</tr>
<tr>
<td>Q1.1.4. Was the possibility to define different horizontal links useful?</td>
<td>81.25</td>
</tr>
<tr>
<td>Q1.1.5. Were negative traceability links useful to discover content overlapping among artefacts?</td>
<td>43.75</td>
</tr>
<tr>
<td>Q1.1.6. Was the functionality to visualise the users allocated to a specific artefact useful?</td>
<td>80.58</td>
</tr>
<tr>
<td>Q1.1.7. Was the functionality to visualise the graph composed by the artefacts of the project and the traceability links useful?</td>
<td>55.40</td>
</tr>
<tr>
<td>Q1.1.8. Was the possibility to subscribe specific artefacts useful?</td>
<td>51.80</td>
</tr>
<tr>
<td>Q1.1.9. Was the event hierarchy useful?</td>
<td>66.19</td>
</tr>
</tbody>
</table>

The only case in which the perceived usefulness decreased below an acceptable threshold is in question Q1.1.8. In this case, the negative evaluation of the subjects was affected by the fact that the subscription mechanism has been changed between
the two experimentations. The new version of the system does not need the explicit subscription for any artefact allocated to the software artefact, as they are automatically considered subscribed. Moreover, the adaptive tuning of the subscription implemented in the new version of ADAMS, helped in reducing the necessity to modify the event subscriptions. This caused the user perception of the usefulness of the subscription mechanism to decrease.

Table III.– Evaluation of Functionality: adequacy

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2005</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Q.1.2.1. Were ADAMS team management functionalities adequate?</td>
<td>56.25</td>
</tr>
<tr>
<td>Q.1.2.2. Did Project Management functionalities fit your needs?</td>
<td>93.75</td>
</tr>
<tr>
<td>Q.1.2.3. Did Resource Management functionalities fit your needs?</td>
<td>87.50</td>
</tr>
<tr>
<td>Q.1.2.4. Did the revision process fit you needs?</td>
<td>77.14</td>
</tr>
<tr>
<td>Q.1.2.5. Did the Fine Recovery functionality fit your needs?</td>
<td>59.38</td>
</tr>
<tr>
<td>Q.1.2.6. Did the Bulk Recovery functionality fit your needs?</td>
<td>90.63</td>
</tr>
<tr>
<td>Q.1.2.7. Did the Versioning Management functionality fit your need?</td>
<td>76.98</td>
</tr>
</tbody>
</table>

Table III is focused on the attribute of adequacy of the functionalities of ADAMS with respect to the needs of the subjects. Even in this case, the users declared an overall adequacy of the system functionalities. It is worth noting that this part of the questionnaire (except for the question Q1.2.4) was submitted only to the team managers (the project manager for questions Q.1.2.1 to Q.1.2.3 and the quality
manager for questions Q.1.2.5 to Q.1.2.7). Since the number of projects involved in the second experimentation is four, the distribution of answers for these questions is a multiple of 25%, and the comparison between the values obtained during the two experimentations is less meaningful.

The third part of the functionality evaluation questionnaire addresses user consideration about the tool usage and design decisions such as the role-base access control policy, the fine-grained approach, and the usefulness of branching and merging when a fine-grained approach is preferred (see Table IV). In particular, it is worth focusing the attention on questions Q.1.3.2 and Q.1.3.3 as they give an indication of the effect of fine-grained management of software artefacts onto user behaviour. In fact, the percentage of software engineers that used the branching and merging functionalities is very low (about 30% in the first experimentation and 7% in the second experimentation).

It is important to note that the results of question Q.1.3.2 is influenced by the answers of the software engineers that decided to use a coarse-grained management (that leads to the need for more branching and merging operations). To this aim, in order to better analyse the effects of fine-grained management with respect to the necessity of branching and merging, I have further investigated the questionnaire answer taking into consideration the answers of the software engineers that preferred a fine-grained management (Q.1.3.2a) and a coarse-grained management (Q.1.3.2b). In this case, the results confirm the conjecture that the need for branching and merging decreases when a fine-grained management is used. In fact, less than 19% of users in the first experimentation and less than 6% of users in the second experimentation used branching and merging even preferring the fine-grained management of artefacts (see question Q.1.3.2a).

Concerning the characteristics of usability (except for the intuitiveness attribute), reliability, and performance the questionnaire items have been phrased as statements for which the subjects expressed their agreement level using a Likert scale from A (“I totally agree”) to D (“I totally disagree”). In the tables here presented answers A and B (agreeing with the text of the question) are reported in the column labelled “Yes” while answers C and D (disagreeing with the text of the question) are
reported in the column labelled “No”. The detailed distribution of answer showing for each question can be found in Appendix A.

**Table IV. – Evaluation of Functionality: considerations**

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer (%)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2005</td>
<td>2006</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Q.1.3.1. Do you agree that a role-based policy to access the artefacts is preferable to a permission-based access control?</td>
<td>81.25</td>
<td>18.75</td>
<td>75.00</td>
</tr>
<tr>
<td>Q.1.3.2. Did you use branching and merging functionalities?</td>
<td>29.29</td>
<td>70.71</td>
<td>7.14</td>
</tr>
<tr>
<td>Q.1.3.2a. Did you use branching and merging functionalities? (users that preferred working at a fine-grained artefact management)</td>
<td>18.97</td>
<td>81.03</td>
<td>5.56</td>
</tr>
<tr>
<td>Q.1.3.2b. Did you use branching and merging functionalities? (users that preferred working at a coarse-grained artefact management)</td>
<td>36.59</td>
<td>63.41</td>
<td>10.00</td>
</tr>
<tr>
<td>Q.1.3.3. Did you preferred working at a fine-grained artefact management level (instead of a coarse-grained artefact management)?</td>
<td>41.43</td>
<td>58.57</td>
<td>64.29</td>
</tr>
<tr>
<td>Q.1.3.4. Do you think that the identification of negative links is an expensive activity?</td>
<td>68.75</td>
<td>31.25</td>
<td>75.00</td>
</tr>
<tr>
<td>Q.1.3.5. Is an iterative link tracing better than a “one-shot” approach?</td>
<td>56.25</td>
<td>43.75</td>
<td>100.00</td>
</tr>
<tr>
<td>Q.1.3.6. Do you agree with the proposed event hierarchy?</td>
<td>68.35</td>
<td>31.65</td>
<td>53.57</td>
</tr>
</tbody>
</table>

Table V and Table VI show the distribution of answers for the questions addressing the evaluation of the usability for the main functionalities of ADAMS. In particular, the project and resource management functionalities are considered
intuitive by 85% and 95% of the users respectively. Versioning management functionalities as well as the software artefact lifecycle management have been considered intuitive (Q.2.1.3. to Q.2.1.5).

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2005</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Q.2.1.1. Are Project management functionalities intuitive?</td>
<td>93.75</td>
</tr>
<tr>
<td>Q.2.1.2. Are Resource Management functionalities intuitive?</td>
<td>93.75</td>
</tr>
<tr>
<td>Q.2.1.3. Are creation and activation procedures intuitive?</td>
<td>80.00</td>
</tr>
<tr>
<td>Q.2.1.4. Are downloading e uploading procedures intuitive?</td>
<td>83.57</td>
</tr>
<tr>
<td>Q.2.1.5. Is the artefact revision procedure intuitive?</td>
<td>82.14</td>
</tr>
<tr>
<td>Q.2.1.6. Is the traceability link creation intuitive?</td>
<td>87.50</td>
</tr>
<tr>
<td>Q.2.1.7. Is the similarity measure between two artefacts intuitive?</td>
<td>71.88</td>
</tr>
</tbody>
</table>

The user interface is considered friendly by about 83% of the users in 2005 and about 89% of the users in 2006. Similar results have been achieved in 2005 for easiness of Input / Output between the users and the system (Q.2.2.1 to Q.2.2.2) and data retrieval (Q.2.2.3). While the former value has been confirmed in 2006, a significant improvement (more than 96%) has been achieved for the easiness of data retrieval in the new version of the system.

Reliability evaluation results are reported in Table VII and Table VIII. The
overall results are encouraging. Even the performances evaluation show good results in terms of adequacy of the performances with respect to the user needs and considering the limitation of a web application in terms of bandwidth restrictions. However, question Q.4.1. highlight some problems in terms of response time. This is mainly related to the time needed to access the work breakdown structure of the project. In fact, in case of project with a very high number of artefacts (the project ADAMS itself has been put under configuration management, resulting in about 2000 artefacts only for the code section and a hierarchy depth of 10 levels), the system suffered some scalability problems.

Table VI. – Usability evaluation: user interface

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer (%)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2005</td>
<td>2006</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Q.2.2.1. Providing input to the system is easy.</td>
<td>83.57</td>
<td>16.43</td>
<td>82.14</td>
</tr>
<tr>
<td>Q.2.2.2. Obtaining output from the system is easy.</td>
<td>83.57</td>
<td>16.43</td>
<td>96.43</td>
</tr>
<tr>
<td>Q.2.2.3. Searching for data is easy.</td>
<td>83.57</td>
<td>16.43</td>
<td>85.71</td>
</tr>
<tr>
<td>Q.2.2.4. Terms used for commands are clear.</td>
<td>90.00</td>
<td>10.00</td>
<td>96.43</td>
</tr>
<tr>
<td>Q.2.2.5. User Interface components layout is satisfactory.</td>
<td>91.43</td>
<td>8.57</td>
<td>89.29</td>
</tr>
<tr>
<td>Q.2.2.6. Understanding and memorising the User Interface components’ behaviour is easy.</td>
<td>90.71</td>
<td>9.29</td>
<td>100.00</td>
</tr>
<tr>
<td>Q.2.2.7. Understanding the main features is easy</td>
<td>91.43</td>
<td>8.57</td>
<td>92.86</td>
</tr>
<tr>
<td>Q.2.2.8. ADAMS User Interface is friendly</td>
<td>82.86</td>
<td>17.14</td>
<td>89.29</td>
</tr>
</tbody>
</table>
### Table VII. – Reliability evaluation: robustness

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2005 Yes</td>
</tr>
<tr>
<td>Q.3.1.1. The User interface prevents errors.</td>
<td>80.71</td>
</tr>
<tr>
<td>Q.3.1.2. The system allows the user to easily suspend or abort an operation at any time.</td>
<td>87.86</td>
</tr>
</tbody>
</table>

### Table VIII. – Reliability evaluation: error recovery

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2005 Yes</td>
</tr>
<tr>
<td>Q.3.2.1. In case an error occurs, the system allows an easy recovery.</td>
<td>76.43</td>
</tr>
<tr>
<td>Q.3.2.2. ADAMS support for error recovery is satisfactory.</td>
<td>89.29</td>
</tr>
</tbody>
</table>

### Table IX. – Performance evaluation

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2005 Yes</td>
</tr>
<tr>
<td>Q.4.1. Waiting time between request and response is tolerable.</td>
<td>60.00</td>
</tr>
<tr>
<td>Q.4.2. ADAMS Project and Resource Management Subsystems’ performances are satisfactory.</td>
<td>81.25</td>
</tr>
<tr>
<td>Q.4.3. ADAMS Traceability Management Subsystem’s performances are adequate.</td>
<td>84.38</td>
</tr>
<tr>
<td>Q.4.4. The ADAMS Artefact Management Subsystem’s performances are adequate?</td>
<td>84.17</td>
</tr>
<tr>
<td>Q.4.5. ADAMS Event &amp; notification Management Subsystem’s performances are adequate.</td>
<td>93.53</td>
</tr>
</tbody>
</table>
6.4 Discussion

This section describes the threats to validity that can affect our experience and discusses some of the results emerged during the evaluation of ADAMS. These are important for our study since we aim at concluding that the ADAMS system represents a useful support during traceability link identification and the effectiveness of a fine-grained management of software artefacts.

In the experience presented in this Chapter, we tried to simulate a real working environment, although we used undergraduate and master students. In particular, managers have been selected among last-year master students, so they have a very good analysis, development and programming experience, and they are not far from junior industry analysts. In addition, team members are students enrolled in a software engineering course, so they have sufficient knowledge of the software development life cycle. Unfortunately, in a real working environment there are other pressures so the results cannot be completely generalised to the industrial context and then the experience should be replicated with practitioners.

Concerning the questionnaire, it was mainly intend to get qualitative insights. It is worth pointing out that the number of “positive” versus the number of “negative” questions was out of balance. This could be a bias because it was easy for students to agree. Moreover, the default value for the answers was the first of the enumeration (i.e., Yes / I totally agree). This, together with the prevalence of “positive” questions could have affected the good results of the experimentation. For this reason, we plan to replicate the experience using a higher number of “negative” questions in the questionnaire and avoiding the use of a default value for the answers, thus forcing the user to select the answers from the list of possibilities provided.

The artefact repositories built by the students during the experience have a small/medium size but they represent a good benchmark as they cover many different projects with different goals. However, students mainly use the fine-grained approach during the Requirements Analysis Document (RAD) development, managing functional requirements, scenarios, use cases, participating objects, and sequence diagrams as different artefacts. On the other hand, the granularity of the
Object Design Document was quite coarse. This has been influenced by the fact that most of the document was composed of auto generated documentation for the interfaces of the application objects, that were provided as html documents by the javadoc utility.

A special consideration has to be made for source code artefacts. It is quite clear that a web based tool is not suitable for versioning of source code, as it requires the developer to access the system, authenticate, check the source file out, import it into the workspace (e.g., the Eclipse workspace), work on the source file, and then access the system again to perform the check-in of the modified file. Note that the number of source files to modify depends on the functionality to implement and on the dependence level (cohesion and coupling). These difficulties emerged during the experimentation and probably caused the developers to work with source code at a coarser granularity. However, we do not know if the fact that source code was not managed at a fine-grained level was also due to the project deadline (unfortunately, we did not ask this question to the students).

After the first experimentation, an analysis has been executed on the notifications received by a selected group of software engineers involved in different projects and having different roles within their teams, in order to evaluate the tool effectiveness (i.e., how many notifications were generated an how much useful they were considered by each receiver) and the software engineers behaviour (i.e., how much attention they paid to the received messages). Such an analysis revealed that, despite of the notifications filtering based on the subscription level and the possibility of classifying the relevance of a modification applied to an artefact version, the number of notifications is still too large to be managed by a single software engineer. Such a result was mainly spoiled by the proliferation of indirect notifications for a traceability graph with a high number of links.

---

8 Javadoc is a computer software tool from Sun Microsystems for generating API documentation into HTML format from Java source code.
Another consideration we can do concerns the software engineer’s behaviour. We noticed that in a first stage of the project, the classification of notifications was very good (nearly 100%), and the declared usefulness of received messages was satisfactory (about 50%). Subsequently, the behaviour get worse. We noticed that such a worsening occurs close to the delivery dates. In these periods, the number of generated notification greatly increases, due to the high number of releases produced by the team members. Such a proliferation of messages (in a critical period) is responsible of a reduction of the software engineering confidence towards the tool. In fact, even when the number of notifications became manageable, their behaviour was still inappropriate. Such a situation has also been found for software engineers being inactive or away for a short time. In these situations, their notifications were collected and it became hard to settle them. Generalising such a situation, we noticed that the intention in classifying notifications is changeable.

In order to face up these problems, between the two experimentations, we introduced two improvements to the event engine subsystem. A first improvement aimed at reducing the generation of unnecessary notifications. In particular, indirect notifications involving artefacts being distant from the source of the event notified, are quite unlikely useful for most software engineers. For this reason we added the possibility to specify, within a subscription, the maximum distance that a notification can reach across the traceability graph (see Figure 53). In this way, we limited the propagation of indirect notifications.

The evaluation of the software engineers behaviour indicated that the user confidence toward the tool decreases when the number of notification increases. In order to minimise the number of ignored messages while keeping a good context awareness level, we added a subsystem to the Event & Notification Management Subsystem that adaptively tunes the event notification level by monitoring each single software engineer behaviour whenever he/she receives a new notification. In particular, software engineers that repetitively ignore notifications concerning one or more artefacts will realistically ignore later similar notification. By this assumption, the event engine automatically decreases the subscription level accordingly, in order to reduce the number of notification he/she receives. Obviously, in such a tuning
process we take into account the actual working time (e.g. the last access date) of the involved software engineer. The results achieved with these improvements are quite encouraging. In fact, the number of notifications significantly decreased. In the second experimentation each software engineer received, on the average, about 95 notifications during the overall project development process (during the first experimentation the number of notification for each software engineer was more than 1000). Despite the minor number of notifications, the perceived context awareness level was considered adequate by the subjects of the experimentation.

Another issue raised by the experimentation, is related to the scalability of the system with respect to the number of software artefacts composing the work breakdown structure. ADAMS was used to manage its own evolution. This resulted in the creation of a project composed of thousands of artefacts. Indeed, the source code is composed of nearly 2000 artefacts with a hierarchy depth of 10 levels. The version of ADAMS used during the experimentation represented the project work breakdown structure by means of an XML document visualised by means of a XSL. However, this approach required the system to load the complete structure of the selected project into memory, resulting in a system resources wasting. Moreover, the dimension of the XML file transferred from the web server to the browser caused high response times due to the bandwidth limits especially for students connected at home with a dialup connection.

---

9 The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language that supports a wide variety of applications. XML languages or dialects may be designed by anyone and may be processed by conforming software. XML is also designed to be reasonably human-legible, and to this end, terseness was not considered essential in its structure.

10 The eXtensible Stylesheet Language (XSL) is a family of transformation languages which allows to describe the way a document encoded in the XML standard has to be formatted or transformed.
Chapter 7

7 Conclusions and future work

In this thesis we have presented ADAMS, an artefact based process support system that integrates project management and artefact management features. Besides coordinating distributed software engineers working on the same artefact and providing versioning facilities like configuration management systems, ADAMS also support context-awareness through event subscription and notification and artefact traceability. Traceability links are used to propagate events concerning changes to an artefact to the dependent artefacts, thus also increasing the context awareness in the project.

In particular, we focused on the fine-grained management of software artefacts provided by ADAMS, in which a complex artefact is decomposed in a hierarchy of sub-artefacts where each piece is managed as a different entity and versioning is provided for both the atomic entity changed and the entities it is contained in, giving the possibility to define more detailed traceability links. In fact, the software engineer is enabled to specify traceability links between entities within the same document, as well as between parts of two different documents, resulting in a more precise impact analysis.

Moreover, fine-grained artefact management reduces the likelihood of conflicts arising when different software engineers apply changes to the same document. In fact, in case the document is decomposed in sub-artefacts, a conflict occurs only if changes concern the same sub-artefact.

Finally, a composition of finer-grained sub-artefacts allows the project manager to
Chapter 7. Conclusions and future work

precisely assign the responsibilities for each document section and the software
engineers to focus on the work product they have been assigned. This is particularly
important in a distributed software development environment, where the work
breakdown structure is defined, tasks are assigned to the software engineers, and the
intermediate work products (e.g., requirements and actors description, scenarios, use
cases, sequence diagrams, test cases) are successively merged in a deliverable
document.

ADAMS has been implemented as a web-based system using Java technologies,
Apache Tomcat 5.5 as web server and MySq1 5.0 as Database Management System.
The user interface of the system is implemented by about 200 Java Server Pages.
The code for the application logic and data layer is composed of about 35K lines of
java code, spread among 250 servlets implementing the application logic subsystems
and 65 java beans providing data layer functionalities. The database is composed of
40 database tables.

In the last two years, ADAMS has been used as artefact management system in
the Software Engineering courses of the Computer Science program at the
University of Salerno (Italy). At the end of the projects, the students evaluated
ADAMS through a questionnaire. Each question referred to an attribute of four
quality characteristics of interest, namely functionality, usability, robustness, and
performance. The results of the evaluation questionnaire have been presented and
discussed.

We are investigating some of the main issue issues raised during the evaluation of
the system. In particular, we will focus on the variability in the choice of the
granularity decomposition. In our opinion the use of a coarse-grained approach for
specific types of artefact is greatly influenced by the functionalities provided by the
tool to the software engineers. As an example, the difficulties described in the
evaluation discussion about the coarse-grained source code management can be
addressed by allowing the developers to access the main functionalities of ADAMS
directly from the Eclipse plug-in. This will simplify the management of source code
and reduce to the minimum the overhead required by the web-based architecture of
the system.
Another issue raised by the experimentation regards the scalability of the approach. To address this issue, in the new version of ADAMS the user interface used to visualise the project artefacts has been implemented with the AJAX (Asynchronous JavaScript and XML) technology. When the software engineer accesses the project artefacts, the system provides only the root artefacts. Anytime the software engineer selects one of the artefacts visualised, an asynchronous call is sent to the web browser which recovers the children of the selected artefacts. This information is used during the visualisation. In this way, only the information about the artefacts effectively visualised are loaded by the system. This greatly improved the response time and the system resources usage as the hierarchies (possibly complex) of documents that are not accessed do not affect the performances resulting in a really scalable system.

Other directions for future work have emerged by the user suggestions provided during the first experimentations. In particular, the fine-grained management of software artefact will be available in the Eclipse plug-in not only for source code artefacts but even for any other type of artefact (e.g., diagrams and high level software documentation) produced during the software development life cycle. This is especially important to enable the developers to benefit on the advantages of the fine-grained approach reducing the overhead caused by an augmented number of configuration items. Moreover, the support for source code will be improved by integrating an optional optimistic approach and enhancing the fine-grained support for this type of artefact.
Chapter 7. Conclusions and future work
Appendix A

A. Survey questionnaires

In this appendix a full report of the survey questionnaire is presented. For each question the distribution of the user answer have been subdivided according to the year of the experimentation.

Questions have been classified according to four main characteristics of interest, namely functionality, usability, reliability, and performance. In some cases, these characteristics have been specialised according to an attribute. Besides the aforementioned characteristics, some quantitative data have also been collected and presented at the end of the appendix.

Table X. – Characteristics and Attribute evaluated

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1. Functionality</td>
<td>A1.1. Usefulness</td>
</tr>
<tr>
<td></td>
<td>A1.2. Adequacy</td>
</tr>
<tr>
<td></td>
<td>A1.3. User Considerations</td>
</tr>
<tr>
<td>C2. Usability</td>
<td>A2.1 Intuitiveness</td>
</tr>
<tr>
<td></td>
<td>A2.2 User Interface</td>
</tr>
<tr>
<td>C3. Reliability</td>
<td>A3.1 Robustness</td>
</tr>
<tr>
<td></td>
<td>A3.2 Error Recovery</td>
</tr>
<tr>
<td>C4. Performance</td>
<td></td>
</tr>
<tr>
<td>C5. Quantitative Data</td>
<td></td>
</tr>
</tbody>
</table>
C1. Functionality

7.1.1 A1. Usefulness

Q1.1.1. Was ADAMS useful during the software development process?

Q1.1.2. Was the possibility to add comments to artefacts useful?
Q1.1.3. Was the possibility to send a feedback to the artefact manager useful?

Q1.1.4. Was the possibility to define different horizontal links useful?
Q1.1.5. Were negative traceability links useful to discover content overlapping among artefacts?

![First Evaluation (2005) Diagram]


Q1.1.6. Was the functionality to visualise the users allocated to a specific artefact useful?

![First Evaluation (2005) Diagram]

Q1.1.7. Was the functionality to visualise the graph composed by the artefacts of the project and the traceability links useful?

Q1.1.8. Was the possibility to subscribe specific artefacts useful?
Q1.1.9. Was the event hierarchy useful?

Q1.1.10. Did ADAMS provide a useful support for documentation management during the software development process.
7.1.2  A1. Adequacy

Q1.2.1. Were ADAMS team management functionalities adequate?

Q1.2.2. Did Project management functionalities fit your needs?
Q1.2.3. Did Resource Management functionalities fit your needs?

Q1.2.4. Did the revision process fit you needs?
Q1.2.5. Did the Fine Recovery functionality fit your needs?

Q.1.2.6. Did the Bulk Recovery functionality fit your needs?
Q1.2.7. Did the Versioning Management functionality fit your need?

- First Evaluation (2005):
  - Yes: 23.02%
  - No: 76.98%

  - Yes: 25.00%
  - No: 75.00%

Q1.2.8. Did ADAMS Re-Trace provide a useful support during the traceability link definition process.

- First Evaluation (2005):
  - I totally agree: 28.13%
  - I agree: 0.00%
  - I disagree: 0.00%
  - I totally disagree: 71.86%

  - I totally agree: 0.00%
  - I agree: 100.00%
7.1.3 A3. User Considerations

Q1.3.1. Do you agree that a role-based policy to access the artefacts is preferable to a permission-based access control?

Q1.3.2. Did you use branching and merging functionalities?
Q1.3.3. Did you preferred working at a fine-grained artefact management level (instead of a coarse-grained artefact management)?

Q1.3.4 Do you think that the identification of negative links is an expensive activity?
Q1.3.5. Is an iterative link tracing better than a “one-shot” approach?

Q1.3.6. Do you agree with the proposed event hierarchy?
Q1.3.7. How was your approach to the traceability links recovery tool?

Q1.3.8. Which recovery functionality is the most useful?
C2. Usability

7.1.4 A1. Intuitiveness

Q2.1.1. Are Project Management functionalities intuitive?

Q2.1.2. Are Resource Management functionalities intuitive?
Q2.1.3. Are creation and activation procedures intuitive?

---

Q2.1.4. Are downloading and uploading procedures intuitive?
Q2.1.5. Is the artefact revision procedure intuitive?

Q2.1.6. Is the traceability link creation intuitive?
Appendix A. Survey questionnaires

Q2.1.7. Is the similarity measure between two artefacts intuitive?

7.1.5  A2. User Interface

Q2.2.1. Was providing input to the system easy.
Q2.2.2. Was Obtaining output from the system easy.

Q2.2.3. Was Searching for data easy.
Q2.2.4. Are Terms used for commands clear.

First Evaluation (2005)

- I totally agree: 9.29%
- I agree: 0.71%
- I disagree: 50.71%
- I totally disagree: 39.29%


- I totally agree: 3.57%
- I agree: 0.00%
- I disagree: 42.66%
- I totally disagree: 53.57%

Q2.2.5. Is User Interface components layout satisfactory.

First Evaluation (2005)

- I totally agree: 36.43%
- I agree: 8.57%
- I disagree: 0.00%
- I totally disagree: 55.00%


- I totally agree: 7.14%
- I agree: 3.57%
- I disagree: 21.43%
- I totally disagree: 67.86%
Q2.2.6. Was understanding and memorising the User Interface components’ behaviour easy.

Q2.2.7. Was understanding the main features easy
Q2.2.8. Is ADAMS User Interface friendly

First Evaluation (2005)


C3. Reliability

7.1.6 A1. Robustness

Q3.1.1. The User interface prevents errors.
Q3.1.2. The system allows the user to easily suspend or abort an operation at any time.

7.1.7  A2. Error Recovery

Q3.2.1. In case an error occurs, the system allows an easy recovery.
Q3.2.2. ADAMS support for error recovery is satisfactory.

C4. Performances

Q4.1. Waiting time between request and response is tolerable.
Q4.2. ADAMS Project and Resource Management Subsystems’ performances are satisfactory.

Q4.3. ADAMS Traceability Management Subsystem’s performances are adequate.
Q.4.4. Were the ADAMS Artefact Management Subsystem’s performances adequate?

Q4.5. Were ADAMS Event & notification Management Subsystem’s performances adequate.
C5. Quantitative Data

Q5.1. The overall number of artefact versions produced within your project is:

First Evaluation (2005)

- 18.75%
- 31.25%
- 12.50%
- 37.50%


- 75.00%
- 25.00%

Q5.2. How many different artefact types did you use?

First Evaluation (2005)

- 75.00%
- 12.50%
- 6.25%
- 6.25%


- 75.00%
- 25.00%
Q5.3. Considering the most used artefact type, which is the average dimension of the associated file?

Q5.4. On the average, how many comment did you add for each of your artefact?
Q5.5. On the average, how many feedbacks did you send?

Q5.6. How many traceability link were defined?
Q5.7. How many link did you trace with the tool (Trace Action)?

Q5.8. How many Warning Link highlighted quality problems within the related artefacts?
Q5.9. How many notification did you receive?

First Evaluation (2005)

- Less than 20%
- 20% - 50%
- 50% - 100%
- More than 100%


- Less than 20%
- 20% - 50%
- 50% - 80%
- More than 80%

Q5.10. On the average, how many of the notifications were useful?

First Evaluation (2005)

- Less than 20%
- 20% - 50%
- 50% - 80%
- More than 80%


- Less than 20%
- 20% - 50%
- 50% - 80%
- More than 80%
Q5.11. How many artefacts did you subscribe for event notifications?

First Evaluation (2005)


Legend:
- Less than 5
- 6 - 10
- 10 - 20
- More than 20
Figure Index

Figure 1. Coop/Orm hierarchical document with version graph ................................................. 23
Figure 2. Three layer architecture integrating CoVer ................................................................. 25
Figure 3. Example of Stellation fragments for Java ................................................................. 26
Figure 4. Subsystem version graphs for a given baselines reference basis ........................... 28
Figure 5. Node slot pattern in Molhado ....................................................................................... 30
Figure 6. Fine-grained version control in Molhado ................................................................. 31
Figure 7. Versioning for traceability links in Molhado ........................................................... 32
Figure 8. Ophelia Traceability Layer ......................................................................................... 36
Figure 9. ArchEvol Overview ................................................................................................... 37
Figure 10. OSCAR Architecture ............................................................................................... 39
Figure 11. OSCAR Artefacts Data Model .................................................................................. 39
Figure 12. APER Architecture ................................................................................................. 42
Figure 13. System level model of EBT ...................................................................................... 43
Figure 14. Sysiphus traceability graph ...................................................................................... 45
Figure 15. ADAMS Data Model .............................................................................................. 48
Figure 16. ADAMS architecture .............................................................................................. 49
Figure 17. Web-based interface of ADAMS ........................................................................... 50
Figure 18. Eclipse plug-in for ADAMS .................................................................................... 50
Figure 19. User account management in ADAMS .................................................................... 51
Figure 20. Project Team management in ADAMS ................................................................... 52
Figure 21. Definition of schedule for an artefact ..................................................................... 53
Figure 22. Visualisation of the to-do list in ADAMS ................................................................. 53
Figure 23. Allocation of resources to an artefact ...................................................................... 54
Figure 24. Artefact Types management in ADAMS ................................................................. 54
Figure 25. Template definition for artefact types .................................................................... 55
Figure 26. ADAMS collaborative editor for UML diagrams .................................................... 56
Figure 27. Selection of parent and children artefact types and filtering .................................... 61
Figure 28. Selection of a subset of recovered artefacts ............................................................. 62
Figure 29. Creation of the compositional links ............................................................. 62
Figure 30. Drag & Drop options of ADAMS ............................................................... 63
Figure 31. Cycle prevention check during the composition of an artefact .................. 64
Figure 32. Rebuild of a composed artefact ............................................................... 66
Figure 33. Document organisation using (a) and skipping (b) sections for leaf nodes... 68
Figure 34. Hierarchical view of composed artefacts ............................................... 69
Figure 35. Graph-based view of composed artefacts ................................................. 70
Figure 36. Selection of the type of links / artefacts to show in the graph .................. 71
Figure 37. Traceability link definition form ............................................................. 72
Figure 38. Traceability graph visualisation in ADAMS .............................................. 73
Figure 39. Suggested Links and False Positives ....................................................... 74
Figure 40. Warning Links ....................................................................................... 75
Figure 41. Artefact Life Cycle in ADAMS ............................................................... 78
Figure 42. Definition of templates for an artefact type .......................................... 79
Figure 43. Check-in form ....................................................................................... 80
Figure 44. Out-of-sync visualisation in the to-do list .............................................. 82
Figure 45. Allocation of users to document sections ............................................. 85
Figure 46. Entity objects of the use case meta-model in ADAMS ............................. 86
Figure 47. List of available UML diagrams ......................................................... 88
Figure 48. Adding a new use case ......................................................................... 89
Figure 49. UCD Stock Reorder use case diagram ................................................. 90
Figure 50. Visualisation of a locked use case in the diagram ................................. 90
Figure 51. Hierarchical views of the model elements (a) and RAD (b) artefacts .......... 91
Figure 52. Event Notification .............................................................................. 92
Figure 53. Event subscription in ADAMS ............................................................. 93
Figure 54. Feedbacks visualisation and creation ..................................................... 95
Figure 55. List of the artefact feedbacks ............................................................... 96
Figure 56. ADAMS Inspection Process ................................................................. 99
Figure 57. Defect identification ............................................................................ 100
Figure 58. Merged defect list with agreement information ...................................... 100
Figure 59. ADAMS checklist-based inspection collaborative tool .......................... 101
Figure 60. Issue manager of the ADAMS Rationale Management Tool ............... 102
Figure 61. Closed Issue archive ......................................................................... 103
Figure 62. Selection of the elements to link with rationale .................................. 104
Figure 63. Creation of links between rationale elements ..................................... 104
Figure 64. Rationale Items in the graph-based view ........................................... 105
Glossary

**Artefact**

Software Artefact

**Eclipse**

Eclipse is an open-source, platform-independent software framework for delivering what the project calls "rich-client applications", as opposed to "thin client" browser-based applications. So far this framework has typically been used to develop Integrated Development Environments (IDEs), such as the Java IDE called Java Development Toolkit (JDT) and compiler (ECJ) that comes as part of Eclipse (and which are also used to develop Eclipse itself). Eclipse employs plugins in order to provide all of its functionality on top of (and including) the rich client platform, in contrast to some other IDEs where functionality is typically hard coded. This plug-in mechanism is a lightweight software component framework [76].

**HTML**

HyperText Markup Language (HTML) is the predominant markup language for the creation of web pages. It provides a means to describe the structure of text-based information in a document and to supplement that text with interactive forms, embedded images, and other objects. HTML can also describe, to some degree, the appearance and semantics of a document, and can include embedded scripting language code which can affect the behaviour of web browsers and other HTML processors [76].

**Javadoc**

Javadoc is a computer software tool from Sun Microsystems for generating API documentation into HTML format from Java source code.

**JSP**

Java Server Pages (JSP) are a server-side technology that enables Web developers and designers to rapidly develop and easily maintain, information-rich, dynamic Web pages that leverage existing business systems. As part of the Java technology family, JSP technology enables rapid development of Web-based applications that are platform independent. JSP technology separates the user interface from content generation,
enabling designers to change the overall page layout without altering the underlying dynamic content [76].

**Software artefact**

A software artefact is a deliverable or an outcome of a software process. More generally, any piece of software developed and used during software development and maintenance can be referred to as a software artefact. Examples are requirements specifications, architecture and design models, source and executable code (programs), configuration directives, test data, test scripts, process models, project plans, project documentation. In the following, the terms software artefact and artefact will be used as synonyms anytime the meaning of the work is cleared by the context.

**SRS**

A Software Requirements Specification (SRS) is a complete description of the behaviour of the system to be developed. It includes a set of use cases that describe all of the interactions that the users will have with the software. Use cases are also known as functional requirements. In addition to use cases, the SRS also contains non-functional (or supplementary) requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints). Recommended approaches for the specification of software requirements are described by IEEE 830-1998. This standard describes possible structures, desirable contents, and qualities of a software requirements specification [76].

**UML**

In the field of software engineering, the Unified Modeling Language (UML) is a standardised specification language for object modelling. UML is a general-purpose modelling language that includes a graphical notation used to create an abstract model of a system, referred to as a UML model. UML is officially defined at the Object Management Group (OMG) by the UML meta model, a Meta-Object Facility meta model (MOF). Like other MOF-based specifications, the UML meta model and UML models may be serialised in XMI. UML was designed to specify, visualise, construct, and document software-intensive systems. UML is not restricted to modelling software. UML is also used for business process modelling, systems engineering modelling and representing organisational structures [76].

**Wiki**

A Wiki is a website that allows visitors to add, remove, edit and change content, typically without the need for registration. It also allows for linking among any number of pages. This ease of interaction and operation makes a wiki an effective tool for mass
collaborative authoring. The term wiki also can refer to the collaborative software itself (wiki engine), or to certain specific wiki sites, including the computer science site (the original wiki) and online encyclopaedias such as Wikipedia.

**XML**

The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language that supports a wide variety of applications. XML languages or dialects may be designed by anyone and may be processed by conforming software. XML is also designed to be reasonably human-legible, and to this end, terseness was not considered essential in its structure.

**XSL**

The eXtensible Stylesheet Language (XSL) is a family of transformation languages which allows to describe the way a document encoded in the XML standard has to be formatted or transformed.
Publications


Bibliography


and development in information retrieval, Sheffield, UK, July 2004, 25-32.


[27] CVS Home Page: http://www.cvshome.org


[48] F. Lanubile, T. Mallardo, and F. Calefato, "Tool Support for Geographically


[57] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao, "Infrastructures for


[73] StarTeam Home Page: http://www.starbase.com


[76] A. Von Knethen, and M. Grund, “QuaTrace: A Tool Environment for


